THE INITIATION AND MAINTENANCE OF SWIMMING IN HATCHLING *XENOPUS LAEVIS* TADPOLES

MICHAEL HULL

Doctor of Philosophy

Institute for Adaptive and Neural Computation
School of Informatics
University of Edinburgh
March 2013

ABSTRACT

Effective movement is central to survival and it is essential for all animals to react in response to changes around them. In many animals the rhythmic signals that drive locomotion are generated intrinsically by small networks of neurons in the nervous system which can be switched on and off. In this thesis I use a very simple animal, in which the behaviours and neuronal networks have been well characterised experimentally, to explore the salient features of such networks. Two days after hatching, tadpoles of the frog *Xenopus laevis* respond to a brief touch to the head by starting to swim. The swimming rhythm is driven by a small population of electrically coupled brainstem neurons (called dINs) on each side of the tadpole. These neurons also receive synaptic input following head skin stimulation. I build biophysical computational models of these neurons based on experimental data in order to address questions about the effects of electrical coupling, synaptic feedback excitation and initiation pathways. My aim is better understanding of how swimming activity is initiated and sustained in the tadpole.

I find that the electrical coupling between the dINs causes their firing properties to be modulated. This allows two experimental observations to be reconciled: that a dIN only fires a single action potential in response to step current injections but the population fires like pacemakers during swimming. I build on this hypothesis and show that long-lasting, excitatory feedback within the population of dINs allows rhythmic pacemaker activity to be sustained in one side of the nervous system. This activity can be switched on and off at short latency in response to biologically realistic synaptic input. I further investigate models of synaptic input from a defined swim initiation pathway and show that electrical coupling causes a population of dINs to be recruited to fire either as a group or not at all. This allows the animal to convert continuously varying sensory stimuli into a discrete decision. Finally I find that it is difficult to reliably start swimming-like activity in the tadpole model using simple, short-latency, symmetrical initiation pathways but that by using more

complex, asymmetrical, neuronal-pathways to each side of the body, consistent with experimental observations, the initiation of swimming is more robust. Throughout this work, I make testable predictions about the population of brainstem neurons and also describe where more experimental data is needed. In order to manage the parameters and simulations, I present prototype libraries to build and manage these biophysical model networks.

LAY SUMMARY

Many animal processes are repetitive actions, such as chewing, breathing and walking. Some rhythms, such as the heartbeat, are continuous, but others, such as locomotion, need to be switched on and off. Once a rhythm is started, it is sustained by networks of nerve cells which produce output signals to drive muscles. A simple example of such a network produces swimming in young frog tadpoles. The tadpole begins to swim in response to a brief touch, swimming can last several seconds and stops when the tadpole bumps into something. The swimming rhythm can be generated by a small population of neurons which can be switched 'on' and 'off' by input from short sensory pathways. Two symmetrical populations of 30 nerve cells, one on each side of the brain, play a central role in initiating and sustaining these rhythms. These particular nerve cells have distinctive features: they are electrically connected and this is essential for reliable swimming; they excite each other to sustain their rhythmic activity and they receive direct excitation in response to touch.

I build computational models of these nerve cells to explore the significance of these special features on the initiation of swimming in the tadpole. Modelling allows experiments to be performed that are not possible in living tissue. Firstly, by matching the models to experimental records, I narrow down the most likely locations for electrical connections. I find that their presence causes the activity of the nerve cells to synchronise and also has a dramatic impact on the behaviour of the individual neurons. Secondly, I investigate the effect of the long-lasting, self excitation in the model and show that when a brief 'touch' is given to a single side of the simulated network, the feedback within this population leads to a long-lasting rhythm, which can be stopped by activating the 'stop' pathway. Finally, I investigate how the two populations of nerve cells on each side of the brain need to be excited in order to start swimming-like activity. I find that by exciting the populations of nerve cells differently on each side, the tadpole is able to respond quickly and reliably to touch.

ACKNOWLEDGMENTS

I have been very lucky to work with my supervisor, Alan Roberts, for his patience, enthusiasm and commitment to science. His actions have taught me much more than I expected. It has been a pleasure to be part of the tadpole lab in Bristol and many thanks go to Steve Soffe who has been very supportive and patient with my endless questions over the past four years, and Edgar Buhl with whom it was a pleasure investigating the head skin initiation pathways.

In Edinburgh, many thanks to my supervisors David Willshaw and Mark van Rossum as well as David Sterratt for their advice on modelling. I would also like to thank Bob Meech, for his patience in explaining the subtleties of interpreting electrophysiological data; to Roman Borisyuk for his help with mathematics; and Wenchang Li and Crawford Winlove, for access to their experimental data. On the tool building front, many thanks to Andrew Davison and Eilif Muller for their support and encouragement. Many thanks to Carolina Doran, Bob Meech and my supervisors for reading drafts of this thesis. Finally, a special thankyou to Mum, Dad & my sister Lauren, who have been incredibly supportive over the past few years.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been specified for any other degree or professional qualification.

Edinburgh, March 2013

Michael Hull

CONTENTS

I INTRODUCTION 1

1 INTRODUCTION 3

1.1 How do animals convert a stimulus into a behavioural response? 3

1.2 Why are hatchling *Xenopus laevis* tadpoles a useful model system? 12

1.3 How *Xenopus* tadpoles will be used to address major questions? 21

II METHODS 27

2 MODELLING COMPONENTS OF A BRAINSTEM NETWORK 29

2.1 Overview 29

2.2 Models of membrane currents 38

2.3 Models of synapses 52

2.4 Specific methods for chapters 58

3 MANAGING COMPLEXITY IN SIMULATION TOOLS 61

3.1 General issues in computational science 61

3.2 Existing tools in computational neuroscience 67

3.3 Issues in modelling small, biologically realistic populations of neurons 70

3.4 mreorg 72

3.5 neurounit 76

3.6 morphforge 82

III RESULTS 93

4 A POPULATION OF ELECTRICALLY COUPLED DINS 95

4.1 Abstract 95

4.2 Introduction 96

4.3 Results 100

4.4 Discussion 117

5 SUSTAINED PACEMAKER ACTIVITY IN A POPULATION OF DINS 125
5.1 Abstract 125
5.2 Introduction 126
5.3 Results 129
5.4 Discussion 142
6 THE DECISION TO INITIATE SWIMMING 149
6.1 Abstract 149
6.2 Introduction 150
6.3 Methods 157
6.4 Results 160
6.5 Discussion 171

IV DISCUSSION 179
7 GENERAL DISCUSSION 181
7.1 Overview 181
7.2 Summary of what has been achieved in this thesis 183
7.3 Feedback for experimentalists 184
7.4 Managing simulation complexity 189
7.5 Further directions for modelling 190

V APPENDICES 193
A DIN MODEL PARAMETERS 195
B NEUROUNIT IMPLEMENTATION 199
B.1 Implementation details 200
B.2 Issues arising in parsing expressions involving units 208
C NEUROUNIT EXAMPLES 211
C.1 Level-1 211
C.2 Level-2 211
C.3 Level-3 212
D MORPHFORGE IMPLEMENTATION 215
D.1 Software architecture 215
D.2 Implementation details 236
E MORPHFORGE EXAMPLE SIMULATIONS 239

E.1 Current injection into a single compartment neuron with Hodgkin-Huxley (HH)-type channels 239
E.2 The effect of axonal radius on the speed of action potential propagation. 241
E.3 Simulation of three neurons using neurounit to define synapses 243
F SIMULATOR TEST DATA REPOSITORY 253
G SIMULATION PLATFORM 257
BIBLIOGRAPHY 259

LIST OF FIGURES

Figure 1.1 A simplified view of decision making. 5
Figure 1.2 The generation of antiphasic rhythm in the half-centre model. 10
Figure 1.3 Configurations of the half-centre model. 11
Figure 1.4 Anatomy of the stage 37/38 hatchling *Xenopus laevis* tadpole. 14
Figure 1.5 Tracings of swimming movements in stage 37/38 *Xenopus laevis* tadpoles. 15
Figure 1.6 The inputs and outputs of the network controlling behaviour in tadpole. 16
Figure 1.7 The layout of pattern generator neurons in the tadpole spinal column. 17
Figure 1.8 The ipsilateral head-skin initiation pathway. 17
Figure 1.9 An overview of the synaptic connections between interneurons in the tadpole swimming central pattern generator. 18
Figure 2.1 Ion channels control current flow across an excitable membrane. 31
Figure 2.2 Modelling ion flow across a neuron's membrane as an electrical circuit. 32
Figure 2.3 Extension of a single compartment Hodgkin-Huxley type model to use multiple compartments. 36
Figure 2.4 An example set of voltage-clamp recordings showing the *in situ* slow potassium current in Rohon Beard neurons. 38
Figure 2.5 The inferred steady-state current and conductance graphs for the slow potassium channel. 41
Figure 2.6 The kinetics of the model slow potassium channel and a comparison against voltage-clamp recordings. 42
Figure 2.7 Examples of the fast potassium currents seen in Rohon-Beard and dorsolateral neurons. 44

Figure 2.8 The steady-state currents and conductances of fast potassium currents. 45
Figure 2.9 The kinetics of the model fast potassium channel and a comparison against voltage-clamp recordings. 46
Figure 2.10 The hypothesised very fast, inactivating potassium current in a Dorsolateral (DL) neuron. 47
Figure 2.11 Comparison of the *Hull-12* (green) and *Dale-95* (blue) fast, non-activating potassium channel models. 48
Figure 2.12 Comparison of the *Hull-12* (green) and *Dale-95* (blue) slow, non-activating potassium channel models. 49
Figure 2.13 The kinetics of the sodium channel 50
Figure 2.14 The kinetics of the calcium channel. 52
Figure 2.15 The circuit diagram for the model of a postsynaptic receptor. 54
Figure 2.16 Example traces from the synaptic models. 55
Figure 2.17 Gap junctions between neurons. 57
Figure 3.1 An overview of `mreorg`. 73
Figure 3.2 The *Overview* page in `mreorg`. 74
Figure 3.3 The *Output* page in `mreorg` 75
Figure 3.4 The layers in morphforge and example classes from each layer. 83
Figure 3.5 The figure produced from running Listing 3.5. 87
Figure 3.6 A screenshot of a simple tool that was built on top of morphforge. 89
Figure 4.1 A population of electrically coupled dINs in the hatchling tadpole central nervous system. 99
Figure 4.2 Descending interneurons in the hindbrain and rostral spinal cord. 101
Figure 4.3 Distributions of gap junctions in the linear network of dINs. 103
Figure 4.4 Gap junction distribution and coupling coefficients between dINs. 106
Figure 4.5 Effects of uniform changes in densities of channels on model dIN membrane excitability, firing and action potential propagation. 110
Figure 4.6 Responses of final active dIN model to current injections. 111

Figure 4.7 Effects of electrical coupling on firing properties of network of 30 electrically coupled dINs. 113
Figure 4.8 Simplified circuit diagram of membrane and gap junction currents in the dINs. 114
Figure 4.9 The effects of electrical coupling on firing properties of network of 30 electrically coupled dINs. 116
Figure 5.1 A population of dINs with feedback NMDA synapses in the hatchling tadpole central nervous system. 127
Figure 5.2 Perfusing N-Methyl-D-aspartic acid (NMDA) onto the dIN population. 130
Figure 5.3 Maximum conductance of feedback NMDARs as a function of frequency. 133
Figure 5.4 The response of the dIN network with feedback excitation to brief sensory excitation. 135
Figure 5.5 Switching off the swimming network using a simple inhibitory pathway. 138
Figure 5.6 Generalising the feedback excitation rhythm generation mechanism. 140
Figure 6.1 Simplified behavioural responses of an animal to an environment. 151
Figure 6.2 Simplified circuit diagram of the head-skin initiation pathway and bilateral swimming central pattern generator in the tadpole. 153
Figure 6.3 Physiological recording from dINs during head-skin stimulation. 155
Figure 6.4 A simple generative model of spike times for a single trigeminal interneuron. 158
Figure 6.5 The xIN spike timing model. 159
Figure 6.6 The effects of electrical coupling on a population of dINs in response to synaptic input from a population of Trigeminal Interneurons (tINs). 161
Figure 6.7 The effect of electrical coupling on recruitment of the dIN population. 162

Figure 6.8 Modelling the effects of synaptic input distributions on initiation of a bilateral network. 164
Figure 6.9 The response of the bilateral network to short duration, symmetrically distributed AMPA-receptor (AMPAR)-mediated excitation. 166
Figure 6.10 The response of the bilateral network to short-duration, symmetrically distributed NMDA-receptor (NMDAR)-mediated excitation. 168
Figure 6.11 The response of the bilateral network to temporally spread, symmetrically distributed NMDAR-mediated excitation. 169
Figure 6.12 The response of the bilateral network to asymmetrical, AMPAR and NMDAR-mediated excitation. 170
Figure 6.13 A summary of the activity of the networks after 300 ms in the four bilateral initiation experiments 171
Figure 6.14 A hypothesis for a two sided-initiation pathway. 175
Figure A.1 Side view of the morphology of the multicompartmental dIN model. 195
Figure D.1 Using indirection to create simulation primitives. 217
Figure D.2 Morphforge synapse construction. 220
Figure D.3 Merging postsynaptic receptors. 221
Figure D.4 Overview of `PostSynapticTemplate` objects in morphforge. 222
Figure D.5 A simplified view of the architecture for recording in morphforge. 230
Figure D.6 Construction of simulator dependent `Record` objects in morphforge. 231
Figure E.1 The output figure produced from running Listing E.1 239
Figure E.2 The output figure produce from running Listing E.2. The graphs show the effect of axon diameter on action potential propagation velocity. 243
Figure E.3 The output figure produce from running Listing E.3 246
Figure E.4 The summary pdf document produced by Listing E.3 247

LIST OF TABLES

Table 2.1 Parameters used in synapse models between pairs of neurons 56

Table 2.2 Conductance densities of membrane channels used in the dIN parameter sweep and final model. 59

Table A.1 The parameters of the forward and backward rate constants used in the channel models of different currents 196

Table A.2 Final conductances and reversal potentials of the transmembrane currents used in the dIN model 197

Table B.1 Non-trivial terminals symbols used in `neurounit`. The Python regular expression syntax is used. 201

Table B.2 Basic units available in `neurounit` 206

Table B.3 Prefixes support by `neurounit` 207

Table G.1 The versions of software used in for simulations 257

Table G.2 Packages written during this thesis. 257

LIST OF LISTINGS

Listing 3.1 The GHK equation with units in code 77

Listing 3.2 Example of neurounit Level-1. 78

Listing 3.3 Examples of neurounit Level-2 79

Listing 3.4 Example of neurounit Level-3. 81

Listing 3.5 An example script using morphforge to simulate a single compartment neuron containing only leak channels in response to a current injection and plot the results 86

Listing 3.6 An example scenario description from the Simulator-TestData repository. 90
Listing B.1 An example equationset in which neurounit is able to infer the dimensions of all the symbols (v, g & i) 208
Listing B.2 An example equationset in which neurounit is unable to infer the dimensions of the symbols x & y 208
Listing C.1 Examples of valid neurounit strings for Level-1 211
Listing C.2 Examples of valid neurounit strings for Level-2. 211
Listing C.3 An example of neurounit Level-3, used to define a leak channel 212
Listing C.4 An example of neurounit Level-3, used to define an HH-type voltage-gated sodium channel 212
Listing C.5 An example of neurounit Level-3, used to define a voltage-gated Goldman-Hodgkin-Katz (GHK)-type calcium channel channel 213
Listing C.6 An example of neurounit Level-3, used to define a simple synapse model 213
Listing D.1 Constructing `Channel` objects via the `Environment`. 219
Listing D.2 Motivation for postsynaptic templates. 221
Listing D.3 Example of defining synapses using a `PostSynapticTemplate` object. 223
Listing D.4 Making the intention of code explicit. 223
Listing D.5 Defining the segmentation of a morphology for simulation using objects defined in morphforge 225
Listing D.6 Defining the segmentation of a morphology for simulation using user-defined objects. 225
Listing D.7 Example of defining channel distributions in `morphforge` 226
Listing D.8 Example of defining a custom `ChannelApplicator` for defining a particular channel distribution on a neuron 227
Listing D.9 Example of recording from different objects in a simulation 229
Listing D.10 Calcuating input resistance using `Trace` objects 232
Listing D.11 Plotting a `Trace` object with matplotlib 234

Listing D.12 Plotting the results of a simulation with `TagViewer` 235
Listing D.13 Building summaries of simulations in morphforge 236
Listing D.14 Example tag-selection strings, which could be used to select particular `Trace` objects after a simulation 237
Listing E.1 An example simulation using morphforge, in which an HH-type neuron is stimulated with a step current injection. 240
Listing E.2 An example of running three simulations in a single script using morphforge. 241
Listing E.3 An example simulation containing three neurons and two synapses. 243
Listing F.1 An example scenario file from the Simulator-TestData repository. The scenario is designed to test an alpha-type synapse. 255

LIST OF GRAMMARS

Figure B.1 The BNF grammar for `neurounits` (Part 1: Units & Quantity terms) 202
Figure B.2 The BNF grammar for `neurounits` (Part 2: Expressions & Functions) 203
Figure B.3 The BNF grammar for `neurounits` (Part 3: Eqnsets & Library definitions) 204
Figure B.4 The BNF grammar for `neurounits` (Part 4: Imports & Namespaces) 205
Figure D.1 The BNF grammar used to define a syntax for selecting objects based on a system of tags 238

ACRONYMS

4AP 4-Aminopyridine 37

aIN Ascending Interneuron 17

AMPA 2-Amino-3-(3-hydroxy-5-methyl-isoxazol-4-yl) Propanoic Acid 23

AMPAR AMPA-receptor 53

API Application Programmer Interface 69

ASCII American Standard Code for Information Interchange 77

AST Abstract Syntax Tree 79

BNF Backus-Naur Form 199

cIN Commissural Interneuron 17

CNS Central Nervous System 6

CPG Central Pattern Generator 4

CSA Connection Set Algebra 70

CSV Comma Separated Variable 78

DE Differential Equation 67

dIN Descending Interneuron 10

DL Dorsolateral 37

EPSP Excitatory Post-Synaptic Potential 52

GABA Gamma-Aminobutyric Acid 53

GHK Goldman-Hodgkin-Katz 50

HH Hodgkin-Huxley 11

HOC HOC 68

INCF International Neuroinformatics Coordinating Facility 80

IPSP Inhibitory Post-Synaptic Potential 51

LEMS Low Entropy Model Specification 69

MEA multi-electrode array 190

MHR Mid-Hindbrain Reticulospinal neuron 53

MLR Mesencaphlic Locomotor Region 6

MN Motoneuron 17

MODL MOdel Description Language 68

NMDA N-Methyl-D-aspartic acid 10

NMDAR NMDA-receptor 10

ODE Ordinary Differential Equation 79

PIR Post Inhibitory Rebound 10

PSP Post-Synaptic Potential 185

RB Rohon-Beard 16

SBML Systems Biology Markup Language 69

STG Stomatogastric Ganglion 12

TEA Tetraethylammonium 37

tIN Trigeminal Interneuron 25

TTX Tetrodotoxin

XML eXtensible Markup Language 69

XSLT eXtensible Stylesheet Language Transformation 69

Part I

INTRODUCTION

3

MANAGING COMPLEXITY IN SIMULATION TOOLS

3.1 GENERAL ISSUES IN COMPUTATIONAL SCIENCE

> "Increasingly, the real limit on what computational scientists can accomplish is how quickly and reliably they can translate their ideas into working code. "
>
> – Greg Wilson [Wilson, 2006]
> *Where's the Real Bottleneck in Scientific Computing?*

Dramatic developments in computational hardware over the last 30 years mean that vast amounts of data can now be acquired, processed and stored inexpensively. In parallel, a solid base of high-quality open-source platforms and libraries have been developed by research communities around the world. In science this has facilitated the collection and analysis of large quantities of data and today, the expectations of what can be understood quantitatively have ballooned: the climate, medicine, the origin of the universe, and the brain. These quantitative models are often analytically intractable dynamical systems and it is difficult to use traditional mathematical techniques to reason about their behaviours. Instead simulations are used to gain insight into their properties; computational modelling has become a widespread tool used across many fields of science: from subatomic collisions to the formation of galaxies, from rockets to climates, from tadpoles to neocortex.

The software ecosystem is constantly evolving on top of rapidly changing hardware platforms. In contrast to the traditional sciences, scientific computing is a young discipline: the problems of the field are slowly being identified and the best practises are starting to emerge [Merali, 2010]. Surprisingly, the difficult issues around modelling

often do not lie in the numerics of computation because the core code required for simulation kernels is often well studied and highly optimised [Press et al., 2007]. If modelling studies can be summarised in a few hundred words and a few equations and in general the simulations themselves do not take much time to run, why is modelling time-consuming - what limits productivity of individual modellers and what are the bottlenecks in collaborating and extending existing work?

I argue that most modelling is not spent at the cutting edge of a model; implementing new sets of equations, adding new features or trying to understand the numerics of a problem. Instead time is spent on more mundane problems such as managing files and data; converting between formats; interfacing between tools; finding mistakes in code and learning new library interfaces. The problems do not come from our inability to understand complex algorithms or conceptual issues of a model but instead the mundane day-to-day implementation issues: how do we effectively make a computer do what we want? Modelling has many similarities with software development, particularly the open source model, in which small geographically distributed groups of people, often working on loosely related problems, work together to produce technically complex software. What can we learn from the experience of the wider software development community?

The software community has recognised that the surrounding *softer* issues, for example data-management, reproducibility and sharing are central to effective sustained progress in complex software projects. Traditionally the focus of a software project was primarily the resulting executable code and the surrounding *scaffolding* such as documentation, testing, versioning, distribution and coding style was considered an optional nicety. The model has dramatically shifted: Python builds documentation directly into the language, test-driven-development proposes writing tests before any functionality [Beck, 2002], and toolchains are developed so that most software can be built, tested and documented with a single command. Although these issues are not as scientifically stimulating as exciting new algorithms, they must be solved to facilitate collaborative and sustained workflows. Both software projects and computational models require many smaller pieces of software to operate smoothly together. In order to build more advanced models, it is essential to get a handle on complexity: as the number of components, n, in a project increases, the possible

number of interactions between them scales as $O(n^2)$ [Brooks, 1995]. How are large software projects successfully developed if our brains can't reliably process such complexity [Parnin and Rugaber, 2012]?

Across all approaches to software development, a central tenet is effective partitioning [Sommerville, 2004]. Effective languages and libraries encourage interfaces to structure large codebases by encapsulating functionality in modules [Woodfield et al., 1981]. This allows developers to understand software both at a conceptual level, as the interactions of abstract components, but also allows them to focus their attention on a small, isolated part of the program to solve implementation issues [Sommerville, 2004].

In large projects, developers go to great lengths to reduce the user intervention required for often repeated, mundane tasks. Many commonly used tools were developed following the recognition of the difficulties surrounding the management of large amounts of data in software projects, for example the development of filesystems, compilers, development environments and version control tools. By automating a simple task, we make it easy to repeat and can be more confident that it has been performed correctly, which frees developers to work at a higher level. Experienced software developers know that mistakes are made. It is estimated that every 1000 lines of code delivered by industry contains 10-50 bugs [McConnell, 2004] and science is surely not immune to similar mistakes (e. g. [Krug and Kresnow, 1999; Miller, 2006; Gronenschild et al., 2012]). Successful development approaches take this into account. One approach is to avoid writing new code and instead reuse existing, tested libraries. Another approach is to pinpoint and isolate *mistake hotspots*, specifically tasks that are repetitive but conceptually well defined, and delegate these to libraries to remove human intervention as far as possible.

Experienced software developers are acutely aware of their own shortcomings in reliably reading and writing complex code. Tools for automatically quantifying complexity in code have existed for decades [McCabe, 1976]. Rules of thumb about where errors commonly occur have emerged, for example, functions that take more than five parameters or contain more lines than can fit on a screen which can be automatically detected by tools [McConnell, 2004]. Refactoring, simplifying the interfaces of code without affecting functionality, is an area of active research. Coding styles have

emerged because unpredictable program layouts are another form of complexity to be assimilated. The issue here is not that a function with more than five parameters, or without proper indentation is impossible for a developer to understand, but that more mental resources will be spent doing so. Each of these examples taken alone may only have a small effect, but their net result is a significant increase in the unnecessary complexity that a developer contends with.

Modelling is rarely a *hole-in-one* and often may require many cycles of iteration in conjunction with experimental work, suggesting where data is lacking and experiments needed, and incorporating new information as it becomes available. Modelling can be used to perform experiments that are impossible in reality, such as investigating the effects of a channel by removing it. In many cases, the exact issues that need to be investigated may not be known from the outset; instead, for a model to be useful it needs to be quickly and easily manipulated in order to answer specific questions. This scenario presents different problems beyond just solving the sets of equations. Interestingly, many similar issued have been raised in agile programming, a development process used in industry to develop software with shifting requirements [Olague and Etzkorn, 2008]. Specifically in computational modelling, the following particular classes of problems have been identified within the communities and potential solutions already proposed, both at the level of the individual modeller and to facilitate effective group collaboration:

MANAGEMENT OF SIMULATIONS An advantage of building a model, is that it can be used to investigate the effects of changing parameters on output, but how do we store these different simulation setups and results effectively? [Gil et al., 2007; Hudson et al., 2011]. If a modelling framework requires a new file for each experimental setup for every set of parameters, then even a small parameter sweep in which two parameters are tested at three different values and three experiments are run per set of values will require 27 input files. If we naïvely duplicate files, this experiment will become very difficult to maintain: if new experimental data becomes available and a parameter is revised, for instance, then it will need to be changed in 27 places.

DEPENDENCY MANAGEMENT Often, several tools are used in conjunction with each other and several processing steps produce the final output. For example, a set

of parameters, which are used by a simulator to produce a file containing results, which is then plotted using a graphing library. If the original parameter set changes, how can it be ensured that the output graphs are up to date? Mature, general purpose tools solve this problem by automatically tracking the dependencies between inputs and outputs (e. g. make [Mecklenburg, 2004]). To integrate with these dependency tracking tools, our own tools have to be designed with such automation in mind and it is particularly important that the command line is well supported [Groth and Gil, 2008].

REUSE OF COMPONENTS: A central tenet in software development is to avoid duplication of the same information [Kernighan and Ritchie, 1988; Fowler et al., 1999; Hunt and Thomas, 1999]. Similarly, parameters for a model need to be referenced from a single, authoritative place, if not it is inevitable that future changes to the model, for example due to new experimental data, will be made in some places and not others. This may lead to false conclusions being drawn because of running two different versions of a model. It is essential that our modelling tools facilitate component reuse, and more subtly, it must be easy to override particular parameters in a component without *copying and pastin*g, so we can investigate explicit changes.

REPEATABLE WORKFLOWS AND REPRODUCING RESULTS: How do I know which parameters produced a set of output graphs? How do I reproduce a graph I saw six months ago? One approach to this problem is to use version control systems (e. g. Sumatra [Davison, 2012]). Modelling often involves systems with many interacting components. If I change part of my model, how do I know that the effects do not knock-on to other parts of the model and whether previous results from that model are still valid? In software development, a comprehensive *test suite* is often created, which can be run to ensure that changes made in one place does not have unexpected side-effects in other parts of the code [Beck, 2002; Sommerville, 2004]. For this to be effective, it is essential that running this process is simple and has lead to the development of automated *continuous integration* tools. Many general tools have been written which can be adapted to neuroinformatics specific toolchains [Zaytsev and Morrison, 2012].

In modelling it is harder to write tests to check all output; yet we still need to be able to see effects of changing one part of a model on all previous simulations to ensure that our model still behaves within constraints as expected. Our tools need to make it easy to remain in control of a *library* of simulations during model creation, and after publication it should be easy for someone reading a modelling paper to quickly reproduce the graphs.

REINVENTION OF THE WHEEL Many tasks initially seem trivial, for example, extracting the spike times from an analogue voltage signal, but after the edge cases have been handled and bugs have been ironed out, many people may have wasted time solving the same problem. To avoid this we need common platforms and object models that are open, modular and easy to extend, so that parts can be reused across different projects, for example Bruederle et al. [2010].

COMMUNICATION OF MODELS How do we reliably communicate the details of a complex model? Following consistent notations allows our brains to assimilate a problem quicker. A recent literature survey finds that very similar concepts are communicated using a range of different conventions in computational modelling papers [Nordlie et al., 2009]. On a more practical level, how do we reuse models other people have written? Mistakes can be made in manually transcribing parameters from models into papers. One possibility is to generate documentation directly from the models [Nordlie and Plesser, 2010]. This has parallels to software documentation tools (e. g. tangle & weave [Knuth, 1992], doxygen [van Heesch, 1997], sweave [Leisch, 2002]).

In order to develop more complex models, unnecessary complexity needs to be trimmed so that scientific questions remain the focus of attention. Development of interpreted languages, such as Matlab and Python, around low-level numerical algorithms written in C and FORTRAN has already dramatically increased productivity of computational scientists. To overcome today's modelling bottlenecks, we can learn from the experience of the wider software community. Specifically, we need to remove user intervention from mundane activities, standardise interfaces and libraries, develop tools and toolchains to manage simulations and data which support repro-

ducible workflows and develop modelling platforms which encourage the reuse of model components and algorithms.

3.2 EXISTING TOOLS IN COMPUTATIONAL NEUROSCIENCE

As in many fields of computational science, simulating the activity in neuronal networks is achieved by modelling the individual components of the network as Differential Equations (DEs) which are solved over a period of time given an initial starting state. This is a widely used approach for many problems in computational science, and highly optimised, general purpose libraries have been written for efficiently solving large sets of these equations (e. g. [Hindmarsh et al., 2005]). Although models can be written using these libraries directly, there has been a trend towards using specialist neuronal simulators. These act as intermediate layers between the DE solvers and the context of the biological problem, for example NEURON [Carnevale and Hines, 2006] and GENESIS/MOOSE [Bower and Beeman, 1998] for simulations of multicompartmental neurons, and NEST [Gewaltig and Diesmann, 2007] and BRIAN [Goodman and Brette, 2008] for large networks of single compartment neurons [Brette et al., 2007]. These simulators offer a more natural interface for building biologically-based models by allowing modellers to express concepts such as *neuron population, cell segmentation* or *ion diffusion* in a single, high-level statement rather than by manipulating the underlying equations directly. By specialising the generic DE solvers, these tools can be highly optimised for solving the forms of equations occurring in neuronal simulations [Hines, 1984], can support specialist features such as events & delays and can automatically parallelise a simulation over a compute cluster. By hiding the numerical details from the modeller, it is simpler to write and manage complex models. The opaque interface provided by the simulator can be invaluable in debugging models: a simulator such as NEURON provides a thoroughly tested platform to build upon, in contrast to code written by an individual scientist, which is unlikely to have had any testing. This encapsulation dramatically reduces the complexity that a modeller has to consider and allows a great jump in productivity over coding from scratch. The kernels of modern simulators are efficient and support a diverse range of model types.

Over the last few years, Python has emerged as the *de facto* programming language in computational neuroscience [Davison et al., 2009]. Python combines powerful, dynamic typing, elegant architecture and clean syntax. Python is a general purpose programming language, with a large, open-source community providing powerful libraries from databasing to distributed processing to graphical user interfaces. Python provides an excellent basis for computational science by providing a high-level interface to mature, efficient numerical and plotting libraries [Oliphant, 2007; van der Walt et al., 2011]. It is simple to interface to existing code in languages such as C and FORTRAN. Python is a flexible language and which allows high-level concepts to expressed concisely, making it ideal for *plumbing components together*. All major neuroscience simulators now provide Python interfaces. [Eppler et al., 2008; Ray and Bhalla, 2008; Hines et al., 2009; Cornelis et al., 2012].

Both NEURON and GENESIS use custom languages for model specification. NEURON supports high-level simulation configuration through HOC (HOC), and the specification of individual components through MOdel Description Language (MODL) [Hines and Carnevale, 2000; Carnevale and Hines, 2006]. This allows the simulator to provide a high-level interactive workspace for the modeller to define the structure of a simulation, whilst leveraging existing compilers to generate efficient code for the core loops of the computation. NEURON was designed as a modelling environment for humans and unfortunately it is harder to use NEURON as a library. A Python interface to NEURON exists [Hines et al., 2009], which gives access to the HOC interpreter, but has shortcomings; (a) it is difficult to clear a simulation and restart from scratch, making it difficult to run multiple simulations in a single script; (b) it is difficult to define new simulation object types (e. g. channel & synapse models) directly from Python.

To promote model sharing, central repositories have been created such as ModelDB [Peterson and Healy, 1996]. The difficulties in sharing models in computational neuroscience are well documented and best formats for sharing complex models remain unclear [Cannon et al., 2007]. In general, approaches for defining models range between fully declarative to imperative. Declarative descriptions provide a cleaner, focused interface, reducing a complex model to set of parametrisable components, which are defined with values such as *conductance density* for a channel *or opening time constant* for a synapse, while imperative descriptions offer more flexibility be-

cause features from the host language, such as `if` statements, `for` loops and libraries, can also be used.

One example of a declarative format is NeuroML [Goddard et al., 2001; Gleeson et al., 2010], which provides a set of primitives, for example `DoubleExponentialSynapse`. Models are implemented by defining an eXtensible Markup Language (XML) file which specifies the parameters to these primitives, for example, `max_conductance`, `rise_time`, `decay_time` and `reversal_potential`. To use these components in a simulation, simulator-specific code is generated from this file, for example via an eXtensible Stylesheet Language Transformation (XSLT). An advantage of this approach is that it allows the user to specify a model once in a high-level fashion, and use it across multiple simulators. One drawback is that models need to be written in XML, which can make parametrisation difficult, and the indirection involved also necessitates a form of dependency tracking. In another XML-based model description language, Systems Biology Markup Language (SBML), a toolchain has been proposed for managing a set of patches that should be applied to an original XML file in order to change parameters and produce the *working model* [Saffrey and Orton, 2009]. Another problem with declarative frameworks in general is the requirement that a model must be expressed as one of the components predefined by the framework. Solutions are being proposed to these problems in the NineML language and NeuroML V2 via Low Entropy Model Specification (LEMS), which allow the specification of the parametrisable modelling components in terms of more general mathematical primitives, for example, *state variables*, *operating regimes* and *parameters*.

PyNN, a high-level simulator independent Application Programmer Interface (API) for building networks of point neuron models, takes an alternative approach to model specification. Network models are defined directly in Python and interoperability between simulators supported by allowing a model setup to be specified once and the simulator used to run the simulation chosen with a single `import` statement. PyNN avoids the dependencies on external files by predefining the types of neuron and synapse models that can be used. By defining simulations within a scripting language, more flexibility is afforded to the modeller in setting up complex network connectivity than would be available by defining the simulation in a declarative format, al-

though the clean separation between concept and implementation is no longer enforced.

A novel approach to defining network connections, proposed in the Connection Set Algebra (CSA) [Djurfeldt, 2012], is to supply a set of mathematical primitives which are powerful enough to represent algorithmic components for defining connectivity, allowing a complex description in a small number of symbols. More generally, how do we represent components of models that are best expressed as algorithms? The problem is rooted in the historical design of programming languages; traditionally functions are considered separately from data. Passing data around a program is commonplace but passing algorithms around has been more involved (requiring for example, function pointers [Kernighan and Ritchie, 1988]; polymorphism [Stroustrup, 1997] or architectural patterns [Gamma et al., 1994]). Modern languages are blurring this distinction and function objects, *functors,* can be treated in the same way as data structures, capable of being passed as arguments to other functions, greatly lowering the barriers to reusing algorithms.

3.3 ISSUES IN MODELLING SMALL, BIOLOGICALLY REALISTIC POPULATIONS OF NEURONS

In modelling small biologically realistic populations of neurons, beyond the more general issues surrounding computational modelling discussed above, we are also interested in the following topics:

DIMENSIONS & UNITS Quantities in neuroscience are expressed in a variety of units; for example conductance densities can be specified in $\mathrm{pS}/\mu\mathrm{m}^2$, $\mathrm{mS}/\mathrm{cm}^2$ or indirectly, for example 'a neuron with surface area of $1200\,\mu\mathrm{m}^2$ with an input resistance of $300\,\mathrm{M}\Omega$'. Although the conversions between these quantities are not complex, we want to avoid making these tedious conversions by hand. We want to be able to express the parameters used in our model in an explicit, flexible form so it is simple to verify that the values used in a simulation are the same as those measured experimentally.

DEFINING MORPHOLOGIES Neuronal morphologies are used in a variety of contexts; for example, electrophysiological simulations, and connectome reconstruction. Declarative formats for storing morphologies exist, for example SWC and MorphML [Crook et al., 2007], but currently no standard Python object-model has emerged. Our format should be interoperable with existing formats and make common operations simple.

VARIATION & STOCHASTICITY Even within a homogeneous population of neurons, we would expect variation amongst the members and the parameters used in a model usually reflect an average of those measured experimentally. It is important that our tools make it easy to reintroduce this variability seen in biology between components and also allow investigation into the effects of changing particular parameters explicitly

CHANNEL DISTRIBUTIONS There is strong evidence that channels are not uniformly distributed over neuronal membrane; for example the axon hillock and the nodes of Ranvier have been shown to have higher densities of sodium channels, which has already been modelled in studies (e. g. [Schmidt-Hieber et al., 2008]). We would like to be able to express these distributions concisely.

PARAMETER SWEEPS Often, exact parameters are unknown, and we need to try out parameters over a range to understand their effects. It must be very easy to perform parameter sweeps, ideally presenting a simple interface from within a single script to avoid potential complex data-management issues arising due to intermediate files.

INTERACTIVE TOOLS It is useful to explore a model, in order to get intuition into how it works, to understand how crucial particular parameters are to behaviour of a system. The less the number of steps required between a change being made, and the result appearing, the more valuable this process will be and ideally we would like to be able to build interactive tools on top of our toolbox.

REUSABLE COMPONENTS We would like to be able to define composite objects at a high level which can be reused. For example, given the definition of a neuron, including its morphology and channel distributions, or a small subnetwork con-

taining neurons, and their associated connectivity, it should be possible to embed this in several simulations directly without the need to replicate the code.

To tackle some of these issues; I present three pieces of software designed to make it easy to build and manage models for computational neuroscience: (a) `mreorg`, a tool for managing large numbers of Python simulations graphically; (b) `neurounit`, a library for specifying and working with units in computational neuroscience in Python; (c) `morphforge`, a high-level API for simulating multicompartmental neuron models in Python. These are briefly described in the next sections.

3.4 MREORG

Modelling is often an iterative process, and as the complexity of a model increases, managing a large number of associated scripts on the filesystem can quickly become unwieldy. `mreorg` is a tool for managing a set of Python simulations and their results in order to make it easy to monitor the behaviour of an evolving model (Fig. 3.1). `mreorg` automatically captures and stores output from Python simulations including generated graphs and text, and allows the user to later browse simulation results. It differs from Sumatra in that focus is not on tracking the history of the model [Davison, 2012], rather `mreorg` assumes that we are only interested in the results of the current state of the model . The central goal of `mreorg` is to make it as easy as possible to re-run and visualise the new output from a set of simulation experiments when the model changes.

`mreorg` is a tool consisting of two components: (a) a low-level Python library and (b) high-level web-based interface to managing large numbers of simulations. The low-level library leverages Python's dynamic typing and powerful introspection to allow behaviours of scripts to be controlled via environmental variables. For example, certain environmental variables control whether figures are displayed on the screen, or are instead saved to image files by dynamically changing the behaviour of matplotlib at runtime (i. e. *monkey-patching*). This allows the same script to produce interactive figures on the screen (normal behaviour) or be used as part of a batch file without any modification. It is unintrusive and only requires one line (`import mreorg`) to be added near the top of the script to use it.

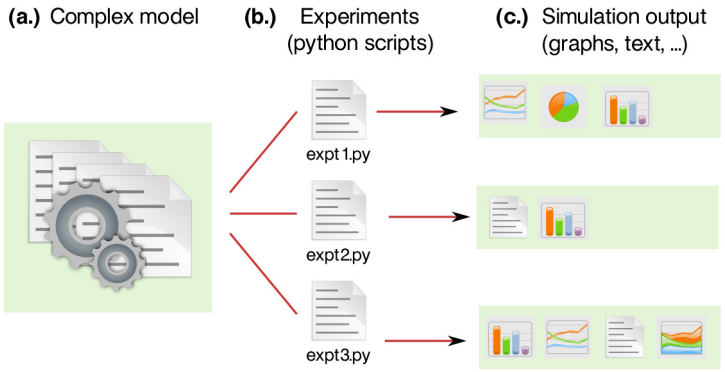


Figure 3.1 – An overview of `mreorg`. The use-case is that a model (a) is tested in many simulation experiments (b), which each produce a set of text and graphical outputs (c). A the goal of `mreorg` is to minimise the effort needed by the user to re-run and visualise new output from the simulation experiments as the model evolves over time.

The high-level tool makes it easier to maintain large numbers of scripts, when changes are being made and provides a simple way to run them in parallel. The tool consists of a django-based *web interface*, and a *backend* that runs the simulations. Using the web interface, a set of Python simulation files on a filesystem can be marked as *Monitored*. Several pages are then available, including an *Overview* page (Fig. 3.2), which summarises the current state of each simulation in a table, an *Output* page (Fig. 3.3), which displays all the output graphs captured from all simulations, and a *Details* page (not shown), which gives all the output (figures, standard-out/error) associated with a particular simulation run. The *Overview* page shows each simulation file as a row in a table including the date and time taken to run the simulation. The colour of the row makes it easy to see the current state of the file (Fig. 3.2), and simulations can be placed in a queue to be run by clicking on *'Run'*. Because the tool is interactive, it is possible to manually control runs of simulations that may take a long time to run.

Unlike Sumatra [Davison, 2012], mreorg does not track Python package dependencies. The goal of this software has been to make it very simple to ensure everything is up to date, quickly to track down errors in batch runs, and easily inspect large numbers of output graphs. More detailed information about `mreorg` can be found in Appendix G.

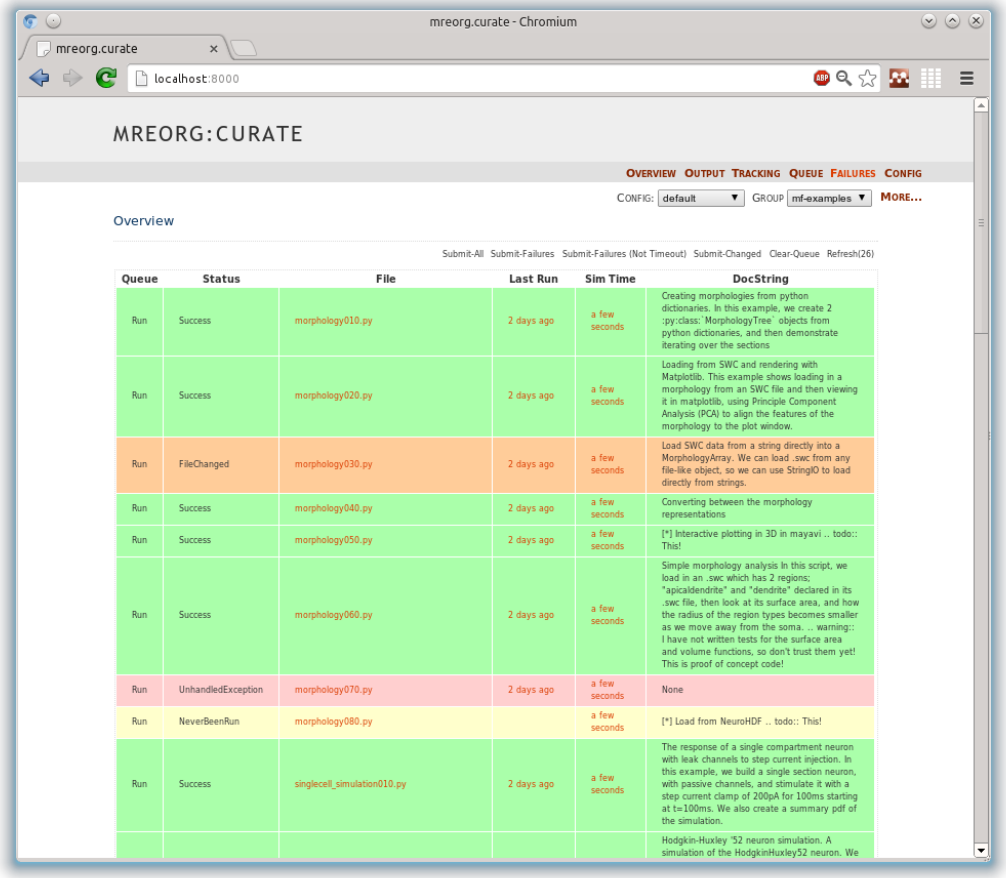


Figure 3.2 – The *Overview* page in `mreorg`, which shows each simulation file as a row in a table including the date and time taken to run the simulation and the docstring. Simulations can be placed in a queue to run by clicking on them. The colour of the row makes it easy to see the status of the simulation file, for example: *green*: simulation successfully ran; *red*: error running the simulation; *blue*: simulation timed out (not shown); *orange*: the file contents have changed since the last run. Simulation files can be grouped and in this screenshot the group *mf-examples* is shown.

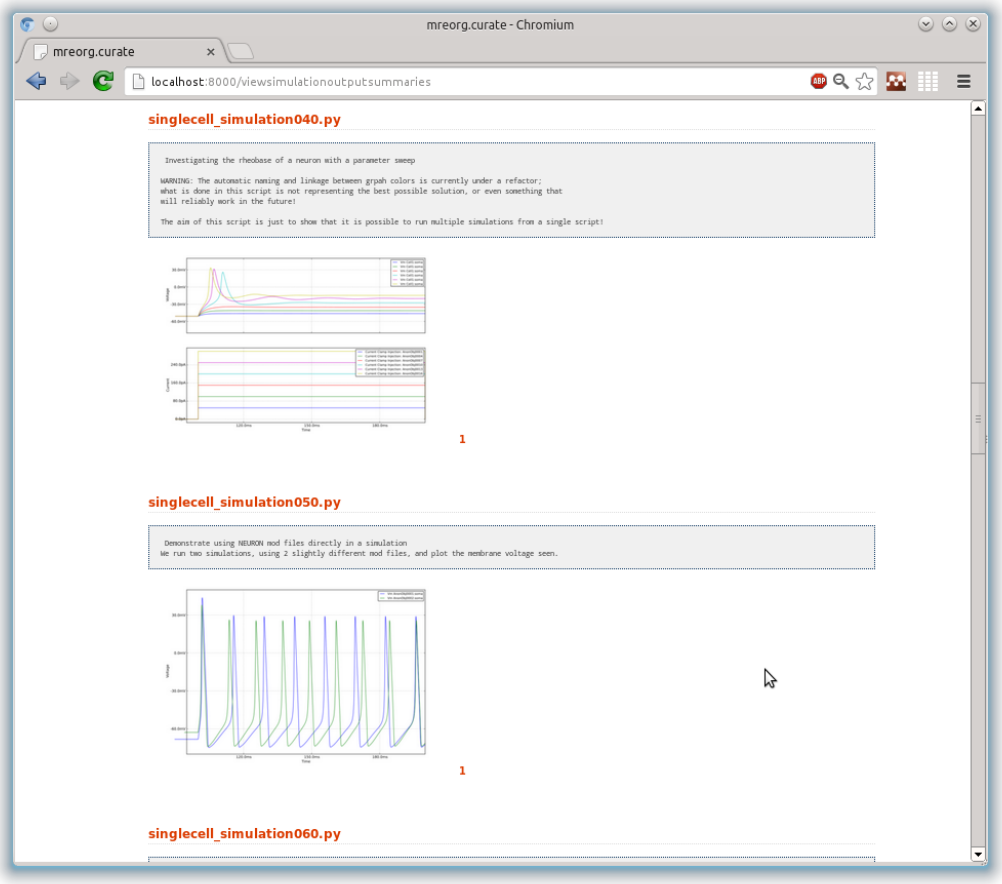


Figure 3.3 – The *Output* page in `mreorg`, which allows all the output graphs from a simulation run to be browsed offline. In this example, the output of two simulations, *singlecell_simulation040.py* & *singlecell_simulation050.py* are shown. This page shows a summary for the simulations (taken from the file's docstring) as well as the output graphs. Clicking on the name of the simulation gives more detailed information, such as the standard output & error and how long the simulation took to run (not shown).

3.5 NEUROUNIT

> "Root Cause: Failure to use metric units in the coding of a ground software file"
>
> *– Mars Climate Orbiter Mishap Investigation Board Phase I Report* [Stephenson et al., 1999]

The Mars Climate Orbiter crashed in 1999 because of a confusion over SI *Newton*s and imperial *pound-force* values. Over $500 million and five years of development was lost because a multiplication factor of 4.45 was missed [Stephenson et al., 1999].

Although conversions between units are mathematically trivial, they can be a source of difficult-to-find bugs. In neuroscience, the scales are small and often particular units of measurements chosen which are suitable for experimentalists. Simulations of biologically realistic neurons often involve many parameters, and it is essential that models are simple for humans to read to find mistakes. The values are not hard to calculate on a single occurrence, but in complex models, errors involving units are more likely to be found if the units of parameter values are specified explicitly and flexibly in a form that is easy to read. To illustrate, if a neuron has a surface area neuron of $1000\,\mu m^2$ and is voltage-clamped at $-30\,mV$, how much calcium current flows in the steady state, assuming $10\,mM$ intracellular and $100\,nM$ extracellular concentration of calcium, a permeability of $0.01\,cm\,s^{-1}$ and assuming that all the channels are fully open (i. e. Eqn. 3.1)?

$$\begin{aligned} I_{ca} &= P_{Ca} 2\nu F \frac{[Ca^{2+}]_i - [Ca^{2+}]_o e^{-\nu}}{1 - e^{-\nu}} \times m^2 \\ \text{where} \quad \nu &= \frac{2V_m F}{RT} \end{aligned} \tag{3.1}$$

Although the value is simple to calculate, unambiguously writing this in code succinctly is more difficult. In the implementation given in Listing 3.1, it is difficult to detect a mistake without resorting to pen and paper. Even if it is correct, when our simulations do not behave as expected, we are likely to scour this code to determine whether it is correct.

```
V_in_mV = -30
F_in_C_per_mol=96485; R_in_J_per_Kmol=8.314; T_in_K=300
pca_in_cm_per_sec=0.01; so_in_mM=10; si_in_mM=100e-3; area_in_um2=1000
nu = (2 * V_in_mV * 1e-3 * F_in_C_per_mol) / (R_in_J_per_Kmol * T_in_K)
ica = pca_in_cm_per_sec * (0.01) * 2.0 * nu * F_in_C_per_mol * (si_in_mM - so_in_mM * exp(-nu)) * ↙
    ↳ (1e-3) / (1-exp(-nu)) * area_in_um2
```

Listing 3.1 – An implemention of the GHK equation in a C-type language (possibly incorrect). It is difficult to quickly ascertain whether this is a correct implementation of Eqn. 3.1.

Computers are excellent at menial, well defined tasks and libraries already exist which perform dimension checking and inference in scientific calculations (e. g. [Petty, 2001; Alexandrescu et al., 2001; Karlsson, 2005; Dale, 2011]). Modern interpreted programming languages such as Python support mechanisms that allow dimensions to be explicitly embedded within the data itself rather than being specified in variable names, in comments, by convention or in documentation. Libraries such as `python-quantities` overload mathematical operators so that a voltage measurement divided by a current measurement will automatically have dimensions of resistance for example. In computational neuroscience, some simulator packages provide tools for checking unit consistency (e. g. `modlunit` in NEURON), and more modern simulators, for example BRIAN, provide built-in support for units [Goodman and Brette, 2008].

Two issues surrounding dimensions remain unresolved: firstly, there is no standard notation for defining simple quantities with associated dimensions as text strings and secondly, we lack suitable libraries that allow sets of equations with units to be specified and transparently handle dimensional inference *behind the scenes*. The first is needed to allow integration of data from different software packages and to allow tools and simulators to converge and interoperate by moving towards more standardised interfaces. The second is essential for building complex models of equations that both humans and machines can easily read.

I propose a grammar for defining units, quantities and systems for equations involving dimensions. The proposed syntax is designed to be unambiguous, but human readability is the central priority. Simple American Standard Code for Information Interchange (ASCII) strings are used because they have no dependencies, are human readable, implementation independent, and can be used within a variety of contexts,

for example in the header of a Comma Separated Variable (CSV) file. A prototype library is available in Python (see Appendix B & G), but a similar library could be implemented using standard libraries available in most mainstream programming languages. Three layers have been defined to support increasingly complex use cases:

LEVEL-1 supports the definitions of units and quantities involving units, for example: `'mV', '10pA/cm2', '0.1e-2mm', '14 centimeter second'`. LEVEL-1 is designed to be simple to parse using simple tokenising routines from standard libraries and predefined lookup tables. Such strings could be used in a simulation configuration files or in the header of a CSV file for example. Using `neurounit` and `python-quantities`, the code in Listing 3.1 can be rewritten as Listing 3.2.

```
from neurounit import Q1
# Define quantities with units using Neurounit Level-1:
V=Q1('-30 mV')
F=Q1('96485 C/mol'); R=Q1('8.314 J/K mol'); T=Q1('300K')
pca=Q1('0.01 cm/s'); so=Q1('10 mM'); si=Q1('100 nM')
area=Q1('1000 um2')

# use python-quantities to calculate the result:
# (note: 'nu' and 'ica' are internally recording the units)
nu = (2*V*F)/(R*T)
ica = pca * 2.0 * nu * F * (si - so * exp(-nu))/(1-exp(-nu)) * area


print ica.rescale('pA')
# Displays: "array(-0.4966447845995212) * pA"
```

Listing 3.2 – Example of neurounit LEVEL-1. We define a series of variables using neurounit LEVEL-1, which we then use to perform a calculation. Since the units are part of the objects (V, F, R, etc), the correct units are automatically propagated to `ica`.

LEVEL-2 supports the calculations of quantities involving units, for example using the mathematical operators `'+-*/'`, and also the use of constants and functions. LEVEL-2 only supports a single expression[1]. In LEVEL-2 (and LEVEL-3), all numerical constants in expression must appear within curly braces,`{...}`, for example `'100mV + 3V'` would be written as `'{100mV}+{3V}'`. This is to prevent ambiguous statements such as `'1 pA / F'` which could refer to either the *one pico-amp-per-farad* or *one pico-amp divided by Faraday's constant*. (These two cases

1 Although the result of one expression could be used in another in a script, for example, 3.2

would instead by written as `{1 pA/F}` or `{1pA}/F`). A built-in library is provided, which contains basic mathematical functions (e. g. `sin()`, `exp()`) and physical constants (e. g. `e`, `pi`, `F`). This allows us to make derivations of parameter values more clear (Listing 3.1).

```
from neurounit import Q2
area_circle = Q2('4 * 3.141 * {10um}**2')
area_sphere = Q2('4 * std.math.pi * {15 micrometer}**2')
leak_conductance = Q2('(1/{300 MOhm}) / {590 um2}')
current_flow = Q2('({1mV}+{200 uV}) * {400 nS}')
```

Listing 3.3 – Examples of neurounit LEVEL-2

LEVEL-3 supports the grouping of differential and normal equations that should be solved together, for example, in a model of a transmembrane current specified by HH equations (see Section. 2.1.1). In `neurounit`, a group of equations, possibly including Ordinary Differential Equations (ODEs), can be grouped together as an equation-set (`eqnset`), in which the symbols are shared among the equations. LEVEL-3 also supports the definition of libraries (`library`), which contain only function definitions and constants, and can be accessed through using a Python-like syntax *(*i. e. using *namespaces* and `import` statements*)*. Both `eqnset` and `library` are specified as blocks, as shown in Listing 3.4. Equation-sets allow the definition of time-evolving state variables, intermediate variables, parameters and functions. LEVEL-3 allows synaptic and channel kinetics to be defined in an unambiguous, easy to read fashion. The syntax is general purpose, and neuroscience-specific information is specified through metadata, such as, which variables represents transmembrane voltages or currents. LEVEL-3 syntax is more complex to read than LEVEL-1 & LEVEL-2 and involves more comprehensive parsing and interpreting phases. `neurounit` internally constructs an Abstract Syntax Tree (AST), which can then be used to produce various outputs, for example, MODL, summary pdf-documents via LaTeX and Python functors.

A discussion surrounding parsing strings with units, design decisions about the proposed syntax and some of the implementation issues are given in Appendix B.

LEVELS 1 & 2 of `neurounit` are used extensively in morphforge (Section 3.6) to specify parameters and LEVEL-3 can also be used to define channel and synapse models (Section E)

Neurounit was developed following discussions at an International Neuroinformatics Coordinating Facility (INCF) NineML taskforce meeting about how to specify the parameters involving units for models. The implementation of LEVELS-1 & 2 provided a more standardised syntax for specifying parameters morphforge, the extension to LEVEL-3 was an experiment to investigate how the syntax would scale. NineML supports a wider range of features that `neurounit` does, for example *regimes* and *transitions*. In the future, it would be interesting to extend NineML to include some of the syntax from `neurounit`.

```
from neurounit import L3
k_chl_def = """
library myneurolib {

   # We define a function for calculating rate constants:
  RateConstant5(V:(V), a1:(s-1) ,a2:(V-1 s-1), a3:(), a4:(V), a5:(V)) = \
     (a1 + a2*V)/(a3+std.math.exp((V+a4)/a5))
}
eqnset chlstd_hh_k {
   from myneurolib import RateConstant5
   i = g * (v-erev) * n**4
   ninf = n_alpha_rate / (n_alpha_rate + n_beta_rate)
   ntau = 1.0 / (n_alpha_rate + n_beta_rate)
   n' = (ninf-n) / ntau
   n_alpha_rate = RateConstant5( V=v,a1=n_a1,a2=n_a2,a3=n_a3,a4=n_a4,a5=n_a5)
   n_beta_rate = RateConstant5( V=v,a1=n_b1,a2=n_b2,a3=n_b3,a4=n_b4,a5=n_b5)

   <=> PARAMETER g
   <=> PARAMETER erev
   <=> PARAMETER n_a1, n_a2, n_a3, n_a4, n_a5
   <=> PARAMETER n_b1, n_b2, n_b3, n_b4, n_b5
   <=> OUTPUT i:(mA/cm2) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }
   <=> INPUT v: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} } }
}
"""

# Parse the string:
k_chl_lm = NeuroUnitParser.File(k_chl_def)
k_chl = k_chl_lm.eqnset['chlstd_hh_k']

# Now, we can do various things with the eqnset:
k_chl.to_nmodl(...)
k_chl.to_latex(...)
k_chl.to_functor(...)
```

Listing 3.4 – Example of neurounit LEVEL-3. We demonstrate using a library and an equationset. The library is defined, which defines a function RateConstant5 which calculate the forward and backward rate constant at a particular voltage, V, given a set of parameters, A, B, C, D & E (see Chapter 2). We next define a potassium channel in an equationset called chlstd_hh_k. After this string has been parsed by neurounit into an object, it can be used in a simulation, written to a LaTeX document or used as a Python functor.

3.6 MORPHFORGE

3.6.1 *Overview*

morphforge is a high-level, simulator independent, Python library for building simulations of small populations of multicompartmental neurons, in which membrane voltage is calculated from the sum of individual ionic currents flowing across a membrane. The use-case of the API is to allow the modeller to quickly construct simulations of small populations of neurons and synaptic connections, with particular focus on:

a) allowing simulation specification, with visualisation and analysis in a minimal, clean, human readable language;

b) reduction of complex simulation toolchains to a single Python script;

c) promoting reproducible research through automatic documentation generation;

d) encourage the reuse of components such as morphologies, neurons and channels such so that specific and stochastic variation in parameters is simple;

e) transparent handling of different units;

f) allowing the use of established formats, (e. g. MODL files), but also simplify the definition and sharing of new channel types easily, including the possibility to support other libraries and standards easily (for example PyNN, NineML, NeuroML, CSA)

Morphforge is not a simulator itself; it is a high-level interface to simulators (currently NEURON) and provides a set of high-level primitives for specifying morphologies, channel distributions and network connections in Python. Morphforge is not designed for large-scale simulations and a design choice was taken to prioritise the interface to the modeller over simulation speed. Morphforge provides a single interface to building models of multicompartmental neurons: an entire *in silico* experiment, including the definition of neuronal morphologies, channel descriptions, stimuli and plotting of results at publication quality can be written in a single short Python script,

without the need for external files. Reusability is central to the library, and consideration has been given to how to reuse both algorithmic and parametrisable components. The design of the API allows models to be written in a declarative style, but because the full Python object model is exposed, complex objects can also be built. It is possible to define new, simulator-independent objects, although the architecture aims not to restrict what can be done to the lowest common denominator. For example, it is possible to use HOC and MODL files in morphforge simulations using the NEURON backend directly.

Morphforge consists of four layers which each define a set of classes which work together as an object model (Fig. 3.4). The higher layers depend on the lower levels, but lower levels do not need the higher ones, for example, `Morphology` objects are used by the `simulation` layer, but can also be used without it, for example for anatomical reconstructions.

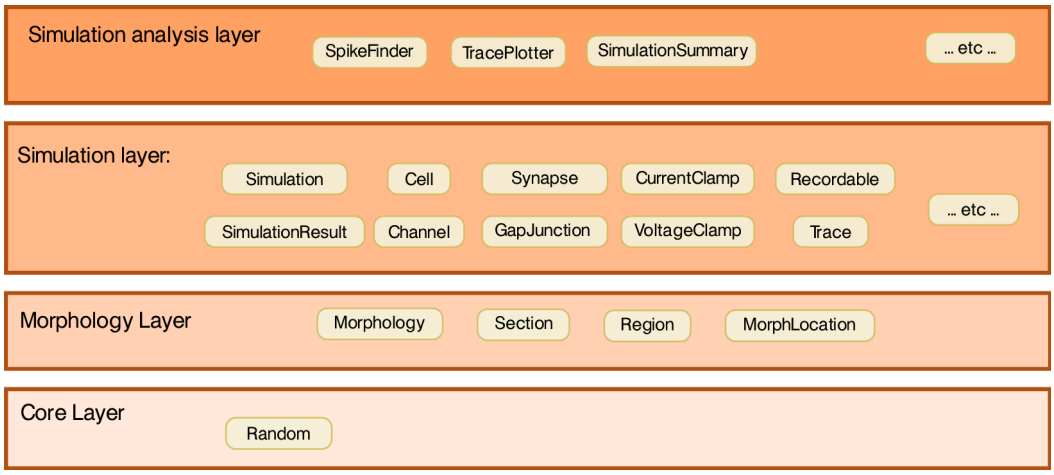


Figure 3.4 – The layers in morphforge and example classes from each layer. The details of each layer will be described in the following sections.

The `core` layer provides a single point of access to control random number seeding, simulation settings and locations on the filesystem access as well as the plug-in infrastructure and utility functions. The `morphology` layer provides classes that represent neuronal morphologies as a tree of cylinders and functions for their creation, import, export, rendering, traversal & manipulation. The `simulation` layer defines a simulator-independent object-model on top of the morphology objects for defining multicompartmental neurons with complex channel distributions. Primitives for de-

fining network connectivity and an interface for recording values during a simulation are also provided. It provides a library system for allowing components (e. g. channel & synapses) to be defined once and reused with different parameters, as well as an extensible high-level object model for representing analogue signals with units. Finally, the `simulationanalysis` layer provides functions for analysing the output of simulations such as spike detection, a visualisation system for easily viewing the outputs of simulations and infrastructure for automatically generating summaries of simulations including the details of components such as channels and synapses.

In the following sections, I will briefly introduce the `morphology` and `simulation` layers. Further details about the high-level architecture, design decisions and implementation of the `simulation` and `simulationanalysis` layers can be found in Appendix D. Example simulations are given in Section E. Further example scripts and API documentation can be found in the morphforge documentation (see Section G).

3.6.2 *`morphology` layer*

Morphforge defines a simple object model for representing the morphologies of neurons, independently of electrical properties. As with many existing formats, morphologies are represented as a tree of joined conical frustra (for example see [Carnevale and Hines, 2006; Crook et al., 2007]). The object model is minimal and it is simple to traverse the tree structure using Python iterators. Morphologies can be imported from SWC & NeuroML files or constructed in code. Morphologies can be rendered as 2D projections using matplotlib and in 3D using MayaVI. The object-model consists of just four classes: `Morphology`, `Section`, `Region` and `MorphLocation`.

The `Section` objects, which correspond to the individual conical frustra, contain information about their length and diameters as well as their connections to other `Sections`. The `Morphology` object, is a container for `Sections`, and represents the morphology of a single neuron. Within a `Morphology`, a group of `Sections` can be defined as a `Region`, and particular locations on the morphology can be specified as `MorphLocation` objects. `Region`, and `MorphLocation` objects are used by the `simulation` layer, for specifying channel distributions and synapse locations.

3.6.3 *simulation layer*

General structure of a simulation

The simulation layer provides an object model for building simulations of multicompartmental neurons and networks entirely within Python. The object-model in the simulator layer is designed to be both flexible and allow complex simulations to be written concisely. A script to run a simulation normally performs the following steps:

a) Instantiate a `Simulation`

b) Define types of `Channel` and `Synapses`

c) Populate the `Simulation` with `Cells`

d) Specify the biophysics of the `Cells`, (e. g. using `Channels`)

e) Connect the `Cells` with `Synapses`

f) Specify experimental stimuli (e. g. `CurrentClamp`))

g) Define which values should be `recorded` during the simulation

h) Call `Simulation.run()`, which runs the simulation and returns a `SimulationResult`

i) Retrieve the recorded values from the `SimulationResult` for plotting or analysis

A simple example script is given in Listing 3.5 that simulates a single neuron with passive channels in response to a step current injection. No external file dependencies are needed, and running the code will cause Figure 3.5 to be displayed on the screen.

LINE [4] A `Simulation` is created, which will use the NEURON backend and last for 200 ms.

LINES [7,8] We create a `Cell` object. We use a dictionary to define the morphology of the cell, which is a single `Section` (cylinder) of length 20 µm and diameter 20 µm. The `Section` has the id of *soma*, which will be used later.

LINES[11-14] A leak channel is defined with a conductance and reversal potential.

```
from morphforge.stdimports import *
from morphforgecontrib.stdimports import *
# Create the environment & simulation
sim = NEURONEnvironment().Simulation(tstop=200*units.ms)

# Create a cell:
cell = sim.create_cell(name="Cell1",
                 morphology=MorphologyBuilder.get_single_section_soma(area=1000*units.um2) )

# Define a leak channel
lk_chl = sim.environment.Channel(StdChlLeak,
         name="LkChl",
         conductance=0.25*units.mS/units.cm2,
         reversalpotential=-51*units.mV)

# Apply the leak channel to the cell, and set the capacitance:
cell.apply_channel(lk_chl)
cell.set_passive(PassiveProperty.SpecificCapacitance, 1.0*units.uF/units.cm2)

# Create the stimulus
cc = sim.create_currentclamp(
   name="Stim1", cell_location=cell.soma,
      amp=200*units.pA,
   dur=100*units.ms,
   delay=100*units.ms)


# Define what to record:
sim.record(cell, what=StandardTags.Voltage, name="SomaVoltage", cell_location=cell.soma)
sim.record(lk_chl, what=StandardTags.CurrentDensity, cell_location=cell.soma)
sim.record(cc, what=StandardTags.Current)

# run the simulation
results = sim.run()

# Display the results:
TagViewer(results, figtitle="The response of a neuron to step current injection", timerange=(95, ↙
     ↳ 200)*units.ms)
```

Listing 3.5 – An example script using morphforge to simulate a single compartment neuron containing only leak channels in response to a current injection and plot the results

LINES[17] The leak channel is *applied* to the cell. (By default it is applied everywhere on the neuron with a uniform density, but this can be specified. This is discussed further in Section D.1.1).

LINES[18] The capacitance of the cell is set.

LINES[20-21] A step current-clamp is created which injects 200 pA for 100 ms starting 100 ms after the simulation starts into the soma of *cell1*.

LINES[29-31] We specify that we want to record the voltage from the neuron at the soma, the current density flowing through the leak channels at the soma and the total current flowing through the current-clamp.

LINES[34] Until this line, we have built an object model in memory, When a `Simulation` is `run()`, morphforge runs the simulation, and returns the result. (Internally, this step can be complex, for example, in this example, morphforge will generate and compile suitable MODL files, spawn an instance of NEURON in a new process and link it with the compiled MODL code. Next, NEURON will run and solve the equations. Morphforge will capture the output, and finally make it available in the original process through the `SimulationResult` object stored in the variable `results`).

LINES[37] Plot the output results. The `TagViewer` inspects the results object, and works out what should be placed on which axes, and deals with the units of the output automatically. The figure displayed by this call is shown in Fig. 3.5.

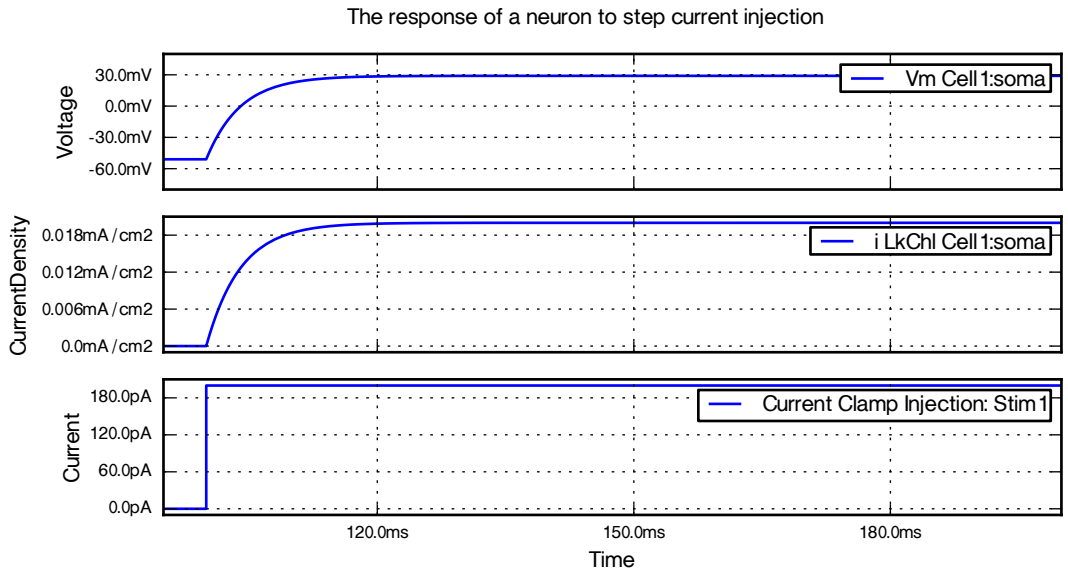


Figure 3.5 – The figure produced from running Listing 3.5. Three axes have been automatically created, for *voltage*, *current density* and *current*, and the values recorded on lines 29-21 automatically displayed with the correct units. The `TagViewer` object is discussed further in Section D.1.2.

Further examples are given in Appendix E and details about the architecture and implementation are given in Appendix D.

3.6.4 *Building on the object model*

One advantage of building an object model over a standalone program is that it can be used as a basis to build more complex tools. I built a tool that allowed us to interactively explore the effects of different parameters on a single tadpole neuron. The tool allowed us to visualise the effects of the forward and backward rate curves functions on the simulated voltage-clamp curves interactively (not shown) and to explore the effects of the different parameters of the model on the responses to current-clamp experiments (Figure 3.6). The tool uses the interactive plotting library Chaco.

Each type of channel (sodium, fast and slow potassium, calcium and leak) in the neuron has a tab, which allows the reversal potential and conductance to be adjusted, as well as a set of interactive graphs for each gating particle for that channel. These graphs show the forward & backward rate constants and the resulting steady state and time constant graphs as a function of voltage The curves are approximated as piecewise line segments, and the connecting points can be adjusted by dragging with the mouse. The graphs are interconnected and moving a point on either the time constant or steady state activation graphs for example will automatically adjust the forward and backward rate equation graphs in real-time, and vice versa. Changing any parameters in the simulation will cause the simulation to be run in NEURON and the results to be displayed in the centre panel. The entire tool is less than 700 lines of code.

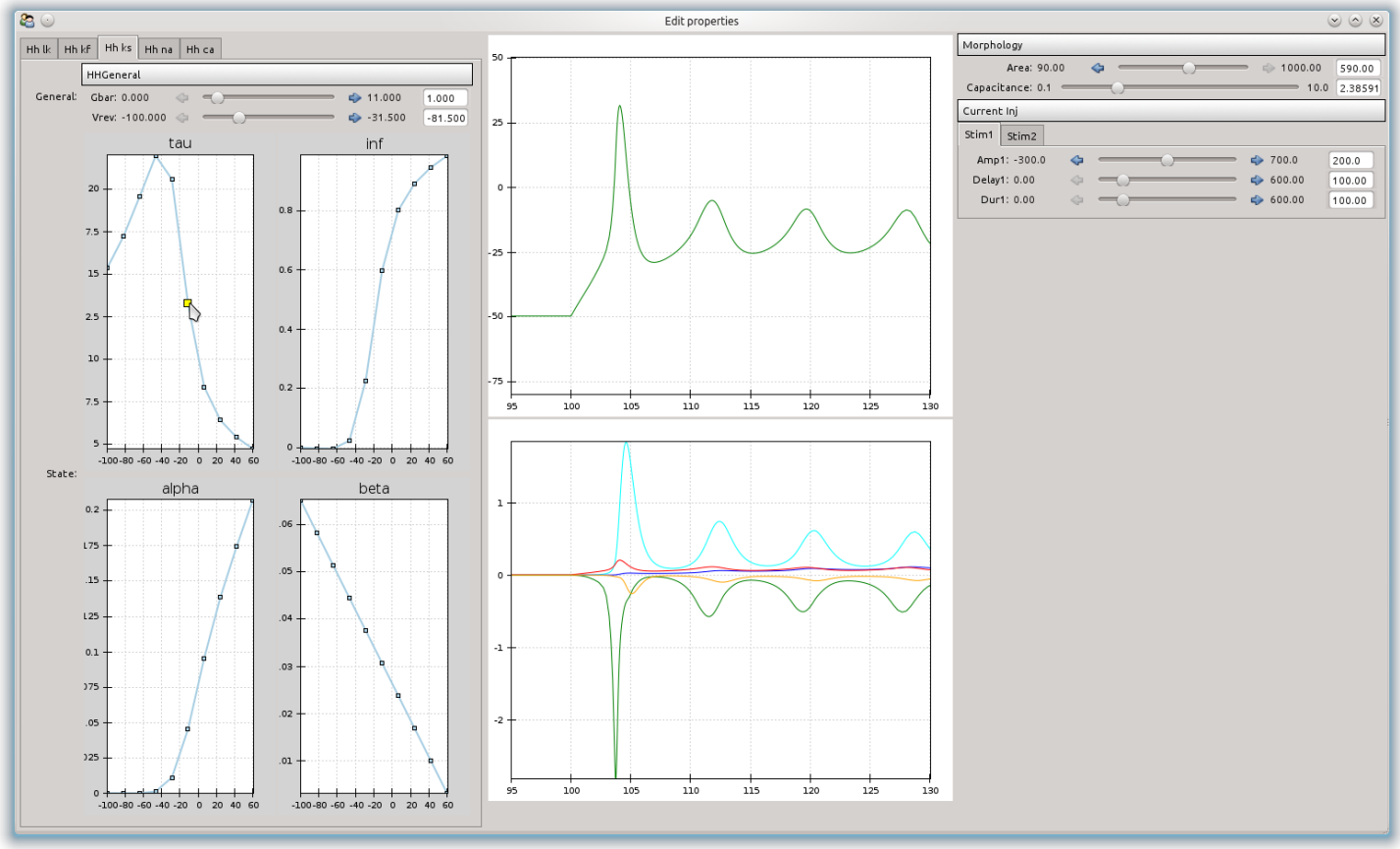


Figure 3.6 – A screenshot of a simple tool that was built on top of morphforge. The tool allows parameters of a simulation to be interactively changed, including passive properties & stimulation protocols (top-right), the conductance densities and reversal potentials of each channel (top-left). For each state variable, for each channel, the forward and backward rates, steady state and time constants as functions of voltage are shown in plots. The curves are approximated by line segments, which can moved interactively using the mouse in order to adjust the channel kinetics. Changing any of the parameters of the model runs the simulation with the new parameters and automatically updates the graphs.

3.6.5 *Testing morphforge*

Testing is an integral part of software development and many proposals have been made for how to test different aspects of software systems in general. One particular difficulty with testing scientific software is that the exact purpose of the code can be difficult to define ahead of time. The approach used to develop morphforge was a combination of defensive programming [McConnell, 2004] in conjunction with high-level functional testing. Morphforge has been tested using a new Simulator-TestData repository, which defines a set of scenarios in a simple, consistent, human & machine readable text file.

Each scenario file describes the setup for a simulation; the start of 'scenario-001' is shown in Listing 3.6. The description is given as a text string. The specification allows parameters to be used in the description (for example `<A>`, `<VS>`, `<C>`, `<GLK>`, `<EREV>` & `<I>`), and defines what should be recorded (for example the voltage of `cell1` as `$V`). The file defines which units to use for all the parameters and recorded values, and also defines the values that should be used for each parameter for parameter sweeps (not shown).

```
scenario_short= scenario001
title = Responses of passive cell to step current injection
description = """
In a simulation lasting 350.0ms
Create a single compartment neuron 'cell1' with area <A> and initialvoltage <VS> and capacitance <C>
Add Leak channels to cell1 with conductance <GLK> and reversalpotential <EREV>
Inject step-current of <I> into cell1 from t=100ms until t=250ms
Record cell1.V as $V
Run the simulation
"""
```

Listing 3.6 – An example of a scenario description from the Simulator TestData repository. (Only the first part of the file is shown)

The repository is designed for testing simulation results in two ways. Firstly, results can be compared against a set of hand-calculated results, which are given as a table in the scenario file. Secondly, result traces from different simulators can be compared against each other. More details about the format and a complete example scenario file are given in F.1.

The Simulator-TestData repository was written to check that no mistakes had been made in unit conversion internally within morphforge. The predominant primitive

objects in morphforge, including current & voltage clamps, channels, synapses and gap junctions, were tested using 10 scenarios, which include tables of hand-calculated expected values. A set of HOC and MODL files were written to ensure that the results from code written directly in NEURON matched those from morphforge.

3.6.6 *Further work*

Morphforge has developed incrementally out of the requirements for the modelling for this thesis. However, I have tried to develop it as a reusable platform for more general use cases. Morphforge is not *production-ready* code, it is a prototype object model for modelling and simulation small networks of multicompartmental neurons. Some decisions were taken because they were simple to implement, rather than being the optimum design decision and improvements can be made.

EFFICIENCY OF THE `trace` CLASS: Currently, there is no sharing of the time values amongst `Trace` objects. This means that if 300 traces are recorded from a simulation, 300 distinct `numpy.arrays` will be created with exactly the same data, which is not efficient for large simulations. There is no architectural reason why this needs to be the case and was done for simplicity. Moreover, further work on morphforge would ideally interface to the classes in NEO[], to provide maximum interaction with electrophysiological data-formats.

SYNAPSES: As was discussed in Section D.1.1, in some cases it is possible to share postsynaptic receptors among several synapses, provided they are linearly superposable. Morphforge currently allows postsynaptic receptors to be shared, however, there are some limitations, for example it is not possible to record internal values, such as current, from shared receptors using the interface described in Section D.1.1. One possibility that would need further development is that the modeller would not specify that receptors are shared, and instead, morphforge could make optimisations after `Simulation.run()` was called. At this point, the object model could detect instances of the same type of receptor onto the same postsynaptic neuron, and, provided that they were linearly su-

perposable and not being recorded from, merge the postsynaptic objects automatically, in order to allow much faster simulations.

ION FLUXES By defining the ionic species of currents, NEURON is able to calculate ion fluxes across the membrane and therefore keep track of changes to intracellular and extracellular ionic concentrations, which can be used by other channels, for example in models of calcium-dependent potassium channels. This is not directly supported by the morphforge object-model, but nor does it it limit it and incorporating MODL using this feature should work as expected.

PYNN INTERFACE morphforge provides an interface for building small populations of multicompartmental neurons. By contrast, `PyNN` provides an interface for specifying simulations of larger populations of neurons. Since they both use Python object models, it should be simple to provide a morphforge-backend to `PyNN`. One benefit of this would be a standardisation of connectivity primitives between software packages, as well as providing a single platform for building multiscale models.

ERROR HANDLING morphforge has been written incrementally as a tool. The code is written defensively [McConnell, 2004] and will cause exceptions to be raised in the case of ambiguous circumstances. morphforge is a prototype, and the software is not polished, which means that sometimes the resulting error stack traces can be hard to diagnose.

SUMMARY GENERATION Morphforge contains infrastructure for automatically generating summaries of the simulation setup. The summaries contain details about the morphologies of the neurons, synapses between them and stimuli (e. g. Fig. E.4). Diagrams are automatically drawn showing the connections between individual neurons or between populations of neurons. The `Channel` objects and `PostSynapticTemplate` objects are able to document their kinetics in a summary. More work is needed to improve the quality of the output diagrams and to display summaries with more appropriate levels of detail depending on the complexity of the simulation.

Part III

RESULTS

B

NEUROUNIT IMPLEMENTATION

I propose a grammar for defining units, quantities and systems for equations involving units and have defined three layers to support increasingly complex use-cases:

LEVEL-1 Simple units and quantities (e. g. `mV`, `10pA/cm2`, `0.1e-3 centimeter second`)

LEVEL-2 Expressions involving quantities (e. g. `(1/{300 MOhm})/{590 um2}`)

LEVEL-3 Systems of equations and libraries (e. g. see Appendix *C*).

It is difficult to write the grammar for neurounit as a single, context-free grammar because of the way in which whitespace needs to be handled and because in many cases the meaning of alphanumeric tokens cannot be ascribed immediately during parsing. (For example, whether an `F` denotes a variable name, Faraday's constant or the unit Farad.) Instead, parsing of expressions is performed in four phases: (*1*) preprocessing to normalise the input string; (*2*) parsing of the normalised string using a Backus-Naur Form (BNF) grammar; (*3*) resolution of the individual unit-symbol; and (*4*) inference of the dimensions of symbols by analysing the ASTs These are described in the following text.

B.1 IMPLEMENTATION DETAILS

B.1.1 *Phase 1: Preprocessing the input*

In life, when writing quantities involving units, care must be taken with the spacing between the characters. For example, `1 m s` (*1 meter second*) is not the same as `1 ms` (*1 millisecond*). In the proposed grammar, this whitespace is also important. In some languages, whitespace between the tokens does not carry meaning except to separate them and can be ignored after the tokenising phase. This simplifies the parsing phase, because whitespace no longer appears in the grammar definition. Unfortunately, since whitespace has meaning in `neurounit` syntax, it cannot simply be ignored. Instead, to reduce the complexity of the BNF grammar, a preprocessing step is used to normalise the string.

Whitespace is only kept if it lies between an alphanumeric character and an alpha character. (Two numbers separated by a whitespace is not valid). The keywords `and` `or` and `not` are converted into the symbols `&`, `|` and `!` respectively, and for LEVEL-3 strings, any empty lines are removed, and lines ending with a backslash are joined with the following line. For LEVEL-3 strings, we also add semicolons to the ends of lines. These simplifications reduce the complexity of the following parsing phase.

B.1.2 *Phase 2: Context-free grammar*

During this stage, the input string is converted into a set of tokens, and then these tokens are parsed by a grammar. A context-free grammar can be defined as a 4-tuple $G = (V_N, V_T, S, P)$ where V_N are the non-terminal symbols, V_T are the terminal symbols, S is the starting symbol and P is the set of production rules [Grune et al., 2000]. In `neurounit`, the three levels all use the same V_N, V_T and P, and only differ by changing the starting symbols, S. The terminal symbols consist of the simple single characters, (`<>()[]{};:+-/=,!&|`) a set of keywords (`if`, `else`, `not`, `and`, `or`, `from`, `import`, `as`, `library` and `eqnset`) and more complex regular-expression-based symbols, given in Table B.1. The full syntax is described by Grammars B.1-B.4.

Table B.1 – Non-trivial terminals symbols used in `neurounit`. The Python regular expression syntax is used.

TERMINAL SYMBOL	DEFINITION
IO_LINE	r"""<=> [^;]*"""
ONEVENT_SYMBOL	r"""==>>"""
INTEGER	r"""[-]?[0-9]+"""
FLOAT	r"""([-]?[0-9]+\.[0-9]*([eE][+-]?[0-9]+)?)| ([-]?[0-9]+([eE][+-]?[0-9]+))"""
WHITESPACE	r"""[\t]+"""
ALPHATOKEN	r"""[a-zA-Z_]+"""
NO_UNIT	r"""NO_UNIT"""
NEWLINE	r"""\n+"""

⟨*alphanumtoken*⟩ ::= ALPHATOKEN>
| ⟨*alphanumtoken*⟩ ALPHATOKEN
| ⟨*alphanumtoken*⟩ INTEGER

⟨*quantity_expr*⟩ ::= ⟨*quantity_expr*⟩ PLUS ⟨*quantity_term*⟩
| ⟨*quantity_expr*⟩ MINUS ⟨*quantity_term*⟩
| ⟨*quantity_term*⟩

⟨*quantity_term*⟩ ::= ⟨*quantity_term*⟩ TIMES ⟨*quantity_factor*⟩
| ⟨*quantity_term*⟩ SLASH ⟨*quantity_factor*⟩
| ⟨*quantity_factor*⟩

⟨*quantity_factor*⟩ ::= ⟨*quantity*⟩
| LBRACKET ⟨*quantity_expr*⟩ RBRACKET

⟨*quantity*⟩ ::= ⟨*magnitude*⟩
| ⟨*magnitude*⟩ ⟨*unit_expr*⟩
| ⟨*magnitude*⟩ WHITESPACE ⟨*unit_expr*⟩

⟨*magnitude*⟩ ::= ⟨*FLOAT*⟩
| ⟨*INTEGER*⟩

⟨*unit_expr*⟩ ::= ⟨*unit_term_grp*⟩
| ⟨*unit_term_grp*⟩ SLASH ⟨*unit_term_grp*⟩
| ⟨*parameterised_unit_term*⟩ SLASH ⟨*parameterised_unit_term*⟩
| ⟨*unit_term_grp*⟩ SLASH ⟨*parameterised_unit_term*⟩
| ⟨*parameterised_unit_term*⟩ SLASH ⟨*unit_term_grp*⟩
| ⟨*parameterised_unit_term*⟩

⟨*parameterised_unit_term*⟩ ::= LBRACKET ⟨*unit_term_grp*⟩ RBRACKET
| ⟨*LBRACKET unit_term_grp*⟩ SLASH ⟨*unit_term_grp*⟩ RBRACKET

⟨*unit_term_grp*⟩ ::= ⟨*unit_term*⟩
| ⟨*unit_term_grp*⟩ WHITESPACE ⟨*unit_term*⟩

⟨*unit_term*⟩ ::= ⟨*unit_term_unpowered*⟩
| ⟨*unit_term_unpowered*⟩ INTEGER

⟨*unit_term_unpowered*⟩ ::= ALPHATOKEN

⟨*empty*⟩ ::= ⟨⟩

⟨*whiteslurp*⟩ ::= ⟨*empty*⟩
| ⟨*WHITESPACE*⟩

⟨*white_or_newline_slurp*⟩ ::= ⟨*empty*⟩
| WHITESPACE
| NEWLINE
| ⟨*white_or_newline_slurp*⟩ WHITESPACE
| ⟨*white_or_newline_slurp*⟩ NEWLINE

Grammar B.1 – The BNF grammar for `neurounits` (Part 1: Units & Quantity terms)

⟨*assignment*⟩ ::= ⟨*lhs_symbol*⟩ EQUALS ⟨*rhs_generic*⟩

⟨*time_derivative*⟩ ::= ⟨*lhs_symbol*⟩ PRIME EQUALS ⟨*rhs_generic*⟩

⟨*rhs_symbol*⟩ ::= ⟨*localsymbol*⟩
| ⟨*externalsymbol*⟩

⟨*lhs_symbol*⟩ ::= ⟨*localsymbol*⟩

⟨*open_funcdef_scope*⟩ ::= ⟨⟩

⟨*function_def*⟩ ::= ⟨*lhs_symbol*⟩ LBRACKET ⟨*function_def_params*⟩ RBRACKET EQUALS ⟨*open_funcdef_scope*⟩ ⟨*rhs_generic*⟩

⟨*function_def_param*⟩ ::= ⟨*localsymbol*⟩
| ⟨*localsymbol*⟩ COLON LCURLYBRACKET RCURLYBRACKET>
| ⟨*localsymbol COLON LCURLYBRACKET <unit_expr*⟩ RCURLYBRACKET>

⟨*function_def_params*⟩ ::= ⟨*whiteslurp*⟩
| ⟨*function_def_param whiteslurp*⟩
| ⟨*function_def_params*⟩ COMMA ⟨*whiteslurp*⟩ ⟨*function_def_param whiteslurp*⟩

⟨*rhs_term*⟩ ::= ⟨*function_call_l3*⟩
| MINUS ⟨*rhs_term*⟩
| LSQUAREBRACKET ⟨*rhs_generic*⟩ RSQUAREBRACKET IF LSQUAREBRACKET ⟨*bool_expr*⟩ RSQUAREBRACKET ELSE LSQUAREBRACKET ⟨*rhs_generic*⟩ RSQUAREBRACKET
| LBRACKET ⟨*rhs_term*⟩ RBRACKET
| ⟨*rhs_term*⟩ PLUS ⟨*rhs_term*⟩
| ⟨*rhs_term*⟩ MINUS ⟨*rhs_term*⟩
| ⟨*rhs_term*⟩ TIMES ⟨*rhs_term*⟩
| ⟨*rhs_term*⟩ TIMESTIMES INTEGER
| ⟨*rhs_term*⟩ SLASH ⟨*rhs_term*⟩
| ⟨*rhs_variable*⟩
| ⟨*rhs_quantity_expr*⟩
| ⟨*quantity*⟩

⟨*function_call_l3*⟩ ::= ⟨*rhs_symbol*⟩ LBRACKET ⟨*func_call_params_l3*⟩ RBRACKET

⟨*func_call_params_l3*⟩ ::= ⟨*rhs_term*⟩
| ⟨*func_call_param_l3*⟩ ⟨*whiteslurp*⟩
| ⟨*func_call_params_l3*⟩ COMMA ⟨*whiteslurp func_call_param_l3*⟩

⟨*func_call_param_l3*⟩ ::= ⟨*alphanumtoken*⟩ EQUALS ⟨*rhs_term*⟩

⟨*rhs_generic*⟩ ::= ⟨*rhs_term*⟩

⟨*bool_term*⟩ ::= ⟨*rhs_term*⟩ LESSTHAN ⟨*rhs_term*⟩
| ⟨*rhs_term*⟩ GREATERTHAN ⟨*rhs_term*⟩

⟨*bool_expr*⟩ ::= ⟨*bool_term*⟩
| ⟨*bool_expr*⟩ AND ⟨*bool_expr*⟩
| ⟨*bool_expr*⟩ OR ⟨*bool_expr*⟩
| NOT ⟨*bool_expr*⟩
| LBRACKET ⟨*bool_expr*⟩ RBRACKET

⟨*rhs_variable*⟩ ::= ⟨*rhs_symbol*⟩

⟨*rhs_quantity_expr*⟩ ::= LCURLYBRACKET ⟨*quantity*⟩ RCURLYBRACKET

Grammar B.2 – The BNF grammar for `neurounits` (Part 2: Expressions & Functions)

⟨*text_block*⟩ ::= ⟨*white_or_newline_slurp*⟩
| ⟨*text_block*⟩ ⟨*block_type*⟩

⟨*block_type*⟩ ::= ⟨*eqnset_def*⟩
| ⟨*library_def*⟩

⟨*open_eqnset*⟩ ::= ⟨*empty*⟩

⟨*eqnset_def*⟩ ::= ⟨*eqnset_def_internal*⟩

⟨*eqnset_def_internal*⟩ ::= EQNSET ⟨*open_eqnset*⟩ WHITESPACE ⟨*namespace*⟩ LCURLYBRACKET ⟨*eqnsetcontents*⟩ ⟨*white_or_newline_slurp*⟩ RCURLYBRACKET ⟨*white_or_newline_slurp*⟩ SEMICOLON ⟨*white_or_newline_slurp*⟩

⟨*complete_eqnset_line*⟩ ::= ⟨*white_or_newline_slurp*⟩ ⟨*eqnsetlinecontents*⟩ ⟨*white_or_newline_slurp*⟩ SEMICOLON

⟨*eqnsetcontents*⟩ ::= ⟨*white_or_newline_slurp*⟩
| ⟨*complete_eqnset_line*⟩
| ⟨*eqnsetcontents*⟩ ⟨*complete_eqnset_line*⟩

⟨*eqnsetlinecontents*⟩ ::= IO_LINE
| ONEVENT_SYMBOL ⟨*event_def*⟩
| ⟨*import*⟩
| ⟨*function_def*⟩
| ⟨*assignment*⟩
| ⟨*time_derivative*⟩

⟨*open_library*⟩ ::= ⟨*empty*⟩

⟨*library_def*⟩ ::= ⟨*library_def_internal*⟩

⟨*library_def_internal*⟩ ::= LIBRARY ⟨*open_library*⟩ WHITESPACE ⟨*namespace*⟩ LCURLYBRACKET ⟨*librarycontents*⟩ ⟨*white_or_newline_slurp*⟩ RCURLYBRACKET ⟨*white_or_newline_slurp*⟩ SEMICOLON ⟨*white_or_newline_slurp*⟩

⟨*complete_library_line*⟩ ::= ⟨*white_or_newline_slurp*⟩ ⟨*librarylinecontents*⟩ ⟨*white_or_newline_slurp*⟩ SEMICOLON

⟨*librarycontents*⟩ ::= ⟨*white_or_newline_slurp*⟩
| ⟨*complete_library_line*⟩
| ⟨*librarycontents*⟩ ⟨*complete_library_line*⟩

⟨*librarylinecontents*⟩ ::= ⟨*import*⟩
| ⟨*function_def*⟩
| ⟨*assignment*⟩

⟨*open_event_def_scope*⟩ ::= ⟨*empty*⟩

⟨*event_def*⟩ ::= ⟨*alphanumtoken*⟩ LBRACKET ⟨*function_def_params*⟩ RBRACKET ⟨*white_or_newline_slurp*⟩ LCURLYBRACKET ⟨*open_event_def_scope*⟩ ⟨*on_event_actions_blk*⟩ RCURLYBRACKET

⟨*on_event_actions_blk*⟩ ::= ⟨*white_or_newline_slurp*⟩ ⟨*on_event_actions*⟩

⟨*on_event_actions*⟩ ::= ⟨*empty*⟩
| ⟨*on_event_action*⟩
| ⟨*on_event_actions*⟩ ⟨*on_event_action*⟩

⟨*on_event_action*⟩ ::= ⟨*empty NEWLINE*⟩
| ⟨*alphanumtoken*⟩ EQUALS ⟨*rhs_term*⟩ ⟨*whiteslurp*⟩ SEMICOLON

Grammar B.3 – The BNF grammar for `neurounits` (Part 3: Eqnsets & Library definitions)

⟨*import*⟩ ::= FROM WHITESPACE ⟨*namespace*⟩ WHITESPACE IMPORT WHITESPACE ⟨*import_target_list*⟩
| ⟨*FROM WHITESPACE <namespace*⟩ WHITESPACE IMPORT WHITESPACE ⟨*localsymbol*⟩ WHITESPACE AS WHITESPACE ⟨*localsymbol*⟩

⟨*import_target_list*⟩ ::= ⟨*localsymbol*⟩ ⟨*whiteslurp*⟩
| ⟨*import_target_list*⟩ COMMA ⟨*whiteslurp*⟩ ⟨*localsymbol*⟩ ⟨*whiteslurp*⟩

⟨*namespace*⟩ ::= ⟨*alphanumtoken*⟩
| ⟨*namespace*⟩ DOT ⟨*alphanumtoken*⟩

⟨*localsymbol*⟩ ::= ⟨*alphanumtoken*⟩

⟨*externalsymbol*⟩ ::= ⟨*namespace*⟩ DOT ⟨*localsymbol*⟩

Grammar B.4 – The BNF grammar for `neurounits` (Part 4: Imports & Namespaces)

B.1.3 *Phase 3: Unit-group symbol resolution*

The next step is to resolve unit-terms strings, such as `'millivolt'` and `'pA'`. In the previous phase, the string has been decomposed into tokens, including `ALPHATOKEN`s (tokens only containing ASCII letters), which might include for example `V` and `F`. Phase-2 provides information to determine whether these tokens represent variable names, constants or units. In Phase-3, the `ALPHATOKEN`s that represent unit terms are resolved. In `neurounit`, this is done with another BNF grammar (not given) and some explicit code to resolve corner-cases. Both short and long forms of values are supported, (e. g. `mV` and `millivolt`). The following supported prefixes and identifiers are given in Table B.2. A unit-term can be either a long or short form of a dimension, a long prefix and a long dimension or a short prefix and a short dimension.

Table B.2 – Basic units available in `neurounit`

LONG-FORM	SHORT-FORM
meter	m
gram	g
second	s
ampere	A
kelvin	K
mole	mol
candela	cd

LONG-FORM	SHORT-FORM
volt	V
siemen	S
farad	F
ohm	Ohm
coulomb	C
hertz	Hz
watt	W
joule	J
newton	N
liter	L
molar	M
watt	W

B.1.4 *Phase-4: Inference of dimensions using ASTs*

Phase-3 provides a set of syntax trees, which define the equations in terms of the symbols. In the Phase-4, dimensional inference occurs, in which `neurounit` inspects the trees and automatically infers the units for symbols which have not had their

Table B.3 – Prefixes support by `neurounit`

LONG-FORM	SHORT-FORM
giga	g
mega	m
kilo	k
centi	c
milli	m
micro	u
nano	n
pico	p

units specified explicitly. For example, in Listing B.1, because we have specified that v is in millivolts, and `g` is in Siemens, `neurounit` will infer that the dimensions of `r` must be Amperes and `g2` must be Siemens. At this stage, `neurounit` does not worry about scaling factors (powers of ten) between quantities, only the base dimensions. Later, when the syntax trees are used for other tasks, for example to generate MODL the relevant scaling factors will be applied accordingly.

In `neurounit`, dimensions are inferred by applying a series of heuristics to the nodes in the AST. For example, at an AST node representing addition, if one of the arguments or the result has a particular dimension, for example Siemens, then all both the arguments and result must also have that dimension, since it does not make sense to add a value in volts to a value in Siemens for example. Different nodes have different heuristics, and the heuristics are also applied to functions and their parameters. In this way, `neurounit` is able to propagate units around the AST trees.

In some cases, it will not be possible to infer all the dimensions. In Listing B.2 for example, there is no way to infer the dimensions of `x` and `y` without further information. The dimensionality of a symbol can be provided with a statement such as the `<=> PARAMETER` statement in Listing B.1. If `neurounit` unit is unable to infer the dimensions of a symbol at the end of Phase-4, an exception is raised.

```
eqnset e1 {
   v = 1 mV
   i = v*(g+g2)
   <=> PARAMETER g : S
   <=> PARAMETER g2
}
```

Listing B.1 – An example equationset in which neurounit is able to infer the dimensions of all the symbols (v, g & i)

```
eqnset e2 {
   v = 1 mV
   x = v * y
}
```

Listing B.2 – An example equationset in which neurounit is unable to infer the dimensions of the symbols x & y

B.2 ISSUES ARISING IN PARSING EXPRESSIONS INVOLVING UNITS

Whilst defining this grammar, several salient issues were raised, which are briefly discussed here.

OPERATOR NOTATION & DIVISION Conventionally, no multiplication sign is written between a magnitude and its dimension, for example, we write $4\,\mathrm{ms}$ rather than $4 \times \mathrm{m} \times \mathrm{s}$ [des Poids et Mesures, 2006]. Similarly, when quantities are written in `neurounit`, no multiplication sign is used, and there is no ambiguity. However, in the case of division of units, the case is more complex and existing tools use different notations. For example, defining the gas constant, `R` = 8.3144 $\frac{\mathrm{J\,mol}}{\mathrm{K}}$. A human, and MODL, would interpret `8.314 J/K mol` as `8.314 J/(K * mol)`, but a `C`-style language[1] as `8.314 (J/K)*mol` because multiplication and division have equal precedence [Kernighan and Ritchie, 1988]. The situation is more ambiguous with multiple division operators, for example `'1m/s/s'`. One approach, as taken by `GNU` units is to introduce a high-priority division operator `|`. `Neurounit` interprets such strings as a human or MODL would, by changing the precedence of space (multiplication) and divide and explicitly disallowing more than one division sign in a units definition, unless parentheses are used. This change in precedence only applies to unit-terms, and C-type precedence is

1 Assuming that space is interpreted as multiplication

used in calculation between quantities, so `{4 m} / {2 s} / {2 s}` is valid and results in $1\,\mathrm{m\,s^{-2}}$.

LIBRARIES & CONSTANTS: `library` allow constant values and functions to be reused between `eqnsets`. A `library` can only contain functions and constants, although one constant can be defined in terms of other constants. It is possible to `import` functions and constants from a `library`, using Python style syntax, for example `from math import pi`.

FUNCTION CALLS It is useful to be able to define functions to avoid duplicating code. `Neurounit` supports the definition of functions within `eqnset` or `library` blocks. Parameters only return a single value, which is denoted by the equals sign in their definition, for example: `area_rect(x,y) = x*y` defines a function called `area_rect` which takes two parameters, `x` & `y` and returns their product. Functions take Python-style named parameters and in function calls, all parameters must be explicitly specified by name, unless the function only takes a single parameter. Functions are defined in `eqnset` or `library` blocks. Briefly, function calls do not have *side-effects*, because functions can only *see* their arguments, other functions and *constants* in the block they are defined in. It is possible to nest function calls, for example `area_square(x)=area_rect(x=x,y=x)`, which would return the square of `x`, but recursive functions are not allowed. To help with unit inference, it is possible to define the expected dimensions of parameters given to functions.

`Neurounit` is a prototype library, and several issues remain to be solved. One issue is that the error reporting is limited when an invalid `eqnset` or `library` is given. Specifically, it can be difficult to diagnose the exact location of either a syntactic mistake, or a mistake due to a units mismatch. The `neurounit` library needs better diagnostics, and isolation of where there is a mismatch in the units expected for a particular symbol, as well as a history of how they have been inferred. Another issue surrounds the flexibility of functions and units. In the current prototype, it is slightly convoluted to write the equation for the electrotonic length of a neuron. This can be calculated from $\lambda = \sqrt{r_m/r_a}$, the dimensions of $r_m/r_a$is in m^2 [Koch, 1999]. Currently, `neurounit` is unable to propagate the dimensions correctly through the square root

function. The `sqrt` function expects a dimensionless arguments and return value, so this expression has to be written as `lambda = sqrt((rm-ra)/{1m2})*{1m}`. This is a symptom of a more general problem with the current implementation, that it is not possible to define 'template' functions, as can be done in C++ (or generics in Java). Fortunately, this does not limit what can be done with the syntax, and only make a few edge case expressions more verbose since it is always possible to explicitly *factor* the dimensions in and out of the arguments and the return value, by multiplying by the appropriate unit constants, as was done in the this case.

C

NEUROUNIT EXAMPLES

C.1 LEVEL-1

```
m == meter
s == second
millisecond == ms
millisecond-1 == ms-1
amp == coulomb/s
m/s2 == m/s s == m/(s s)
volt == amp ohm == A Ohm == W / A == J / A s == N m / A s == kg m2 / C s2 == N m / C == J /C == kg ↙
     ↳ m2/A s3
m / s2 == (m/s)/s == m/(s s)
1.0m == 1meter
100.cm == 1m
101cm != 1m
1 mM == 1 mol/m3
1uM == 1e-3 mol/m3
3 millivolt == 3 kiloamp microohm == 3.0 mA Ohm == 3 W / kA == 3 pJ / A ns == 3e-3 N m / A s == ↙
     ↳ 0.003 kg m2 / C s2 == 0.003 N m / C == 3 J / kilocoulomb == 3 g m2/A s3
3 millivolt != 1 kiloamp microohm
3 millivolt != 3 kiloamp ohm
```

Listing C.1 – Examples of valid neurounit strings for LEVEL-1. The '==' and '!=' are not part of neurounit LEVEL-1 but denote that two expressions are equivalent.

C.2 LEVEL-2

```
{1.0m} / {2s} == {0.5 m/s} == {50 cm/s} == {0.5 mm/ms}
3+4/2==5
3+6/2+1==7
5 m/s2 == 5 m/s s
6 s/m == 6 ms/mm
1/{1s} == 1Hz
1000L == 1m3
({1m/s} - {5m s-1}) / {8ms} == -500. m/s2 == -0.5 km s-1 s-1
{500mN} == {1 kg} * {1 m/s2} / 2.
1 == 1 / 2 * 2
0.125 == 1 / 2 / 4
std.physics.R == 8.314472 J mol-1 K-1
```

Listing C.2 – The '==' and '!=' are not part of neurounit LEVEL-2 but denote that two expressions are equivalent.

'

C.3 LEVEL-3

```
eqnset chlstd_leak {
   i = g * (V-erev)

   <=> PARAMETER g, erev
   <=> OUTPUT i:(mA/cm2) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }
   <=> INPUT V: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
}
```

Listing C.3 – An example of neurounit LEVEL-3, used to define a leak channel

```
library my_neuro {
   from std.math import pow
   RateConstant5(V:{V},a1:{s-1} ,a2:{V-1 s-1}, a3:{},a4:{V},a5:{V} ) = (a1 + ↙
        ↳ a2*V)/(a3+std.math.exp( (V+a4)/a5) )
}

eqnset chlstd_hh_na {

   from my_neuro import RateConstant5
   from std.math import exp
   i = g * (v-erev) * m**3*h

   minf = m_alpha_rate / (m_alpha_rate + m_beta_rate)
   mtau = 1.0 / (m_alpha_rate + m_beta_rate)
   m' = (minf-m) / mtau

   hinf = h_alpha_rate / (h_alpha_rate + h_beta_rate)
   htau = 1.0 / (h_alpha_rate + h_beta_rate)
   h' = (hinf-h) / htau

   m_alpha_rate = RateConstant5( V=v,a1=m_a1,a2=m_a2,a3=m_a3,a4=m_a4,a5=m_a5)
   m_beta_rate = RateConstant5( V=v,a1=m_b1,a2=m_b2,a3=m_b3,a4=m_b4,a5=m_b5)
   h_alpha_rate = RateConstant5( V=v,a1=h_a1,a2=h_a2,a3=h_a3,a4=h_a4,a5=h_a5)
   h_beta_rate = RateConstant5( V=v,a1=h_b1,a2=h_b2,a3=h_b3,a4=h_b4,a5=h_b5)

   <=> PARAMETER g, erev
   <=> PARAMETER m_a1, m_a2, m_a3, m_a4, m_a5
   <=> PARAMETER m_b1, m_b2, m_b3, m_b4, m_b5
   <=> PARAMETER h_a1, h_a2, h_a3, h_a4, h_a5
   <=> PARAMETER h_b1, h_b2, h_b3, h_b4, h_b5
   <=> OUTPUT i:(mA/cm2) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }
   <=> INPUT v: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
}
```

Listing C.4 – An example of neurounit LEVEL-3, used to define an HH-type voltage-gated sodium channel. A simple library is defined, and the RateConstant5 function imported from it in the channel definition

```
eqnset cachl {
      from std.math import exp
      from std.physics import F,R

      n_alpha = ( {10mV} - V ) / ( 100* exp( ({10mV}-V)/{10mV} ) - {1} ) * (1 ms-1 V-1)
      n_beta = 0.128 * exp( V / {-80mV}) * (1 ms-1)
      n_tau = 1/(n_alpha+n_beta)
      n_inf = n_alpha/(n_alpha+n_beta)
      n'= (n_inf - n) / n_tau

      gca = pca* 2 * up * F * (CAi - CAo*exp(-1.0*up) ) / (1-exp(-1.0*up) )
      ica = gca * n**2
      up = 2 * V * F / (R*T)

      T = 300 K

      <=> INPUT V: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
   <=> OUTPUT ica:(mA/cm2) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }

      <=> PARAMETER pca
      <=> PARAMETER CAi:(mole /m3)
      <=> PARAMETER CAo
 }
```

Listing C.5 – An example of neurounit LEVEL-3, used to define a voltage-gated GHK-type calcium channel channel

```
eqnset syn_simple {
   g' = - g/g_tau
   i = gmax * (v-erev) * g

   gmax = 300pS
   erev = 0mV

   g_tau = 20ms
   <=> INPUT v: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
   <=> OUTPUT i:(mA) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }

   ==>> on_event() {
      g = g + 1.0
   }
}
```

Listing C.6 – An example of neurounit LEVEL-3, used to define a simple synapse model

D

MORPHFORGE IMPLEMENTATION

D.1 SOFTWARE ARCHITECTURE

D.1.1 *simulation layer*

Using factories (i) - simulator specific object construction

A simulation in morphforge is entirely encapsulated within a `Simulation` object, which is populated with *primitive objects* such as `Cells`, `Channels`, `Synapses` and then executed by calling `Simulation.run()` (e. g. Listing 3.5). In morphforge, in contrast to the PyNEURON interface for example, all interaction with the simulator-backend is deferred until the `run()` method is called. That is, all changes to the `Simulation` object before this are only modifying morphforge's internal object-model. This approach offers several advantages over interacting with the simulator-backends immediately; (i) a set of simulations can be built in a single thread and their executions can be easily distributed over multiple processors, for example using Python's multiprocessing module[1]; (ii) it is simple to cache the results of a simulation run[2]; (iii) this approach allows us to robustly perform multiple NEURON simulations in a single script (iv) it

1 Although interaction with the simulator directly does not preclude parallelisation, it means that aspects of the simulation will be more difficult to query in the original thread. To parallelise a simulation interacting with the backend directly, the code to build the simulation will likely need to be run in the separate process, and for example, if the number of synapses in the model are made randomly, it will require more effort to return connection information back to the parent process.

2 For example, provided all the information for the simulation is contained in the `Simuation` object, computing and comparing a hash-function of the serialised the `Simulation` objects can provide an efficient way to check whether two simulations are identical.

offers the possibility to optimise aspects of the simulation, for example the possibility of sharing of postsynaptic receptors (discussed in Section 3.6.6).

There are different approaches to creating such a `Simulation` object-model that would support different simulator backends, for example NEURON, GENESIS, MOOSE). One possibility would be to create a `Simulation` object and populate it with backend-independent primitives (e. g. `Cell` & `Synapse` objects). When `Simulation.run()` was called, these objects would then be traversed using a backend-specific mapping tool to convert them into appropriate code to run the on the specific backend.

The approach taken by morphforge is to specify the simulator-backend before constructing the `Simulation` object. Then, instead of populating the object-model with *backend-independent primitives* (e. g. `Cell` & `Synapse` objects), it is populated with *backend-specific primitives* (e. g. `NEURONCell` & `NEURONSynapse` objects). These backend-specific primitives are subclasses of the backend-independent primitives (for example, `NEURONSynapse` inherits from `Synapse`), which allows the simulator-backends to add their specialist methods and variables to the primitives, which greatly simplifies the internal code needed when `Simulation.run()` is called (Fig. D.1). Since we are using a dynamically typed language, it is not strictly necessary to use inheritance, but doing so ensures a consistent interface and separates conceptual and implementation details.

How is the object-model populated with these backend-specific primitives such that is easy to switch simulator-backends? In morphforge, all instantiation of primitives occurs through an `Environment` object. The `Environment` is a factory object which produces backend-specific objects, e. g., the `NEURONEnvironment` produces NEURON-specific primitives [Gamma et al., 1994]. For example, in Listing 3.5, all the objects have all been created, possibly indirectly[3], through calls to the `NEURONEnvironment`, which was defined on line 4, which means the resulting object-model is populated by objects such as: `NEURONSimulation`, `NEURONCell`, `NEURONStdChlLeak` & `NEURONCurrentClamp`.

3 In some cases, instantiation methods are provided via the `Simulation` class, for example, `sim.create_cell` on but internally these methods construct objects using the associated `Environment` object.

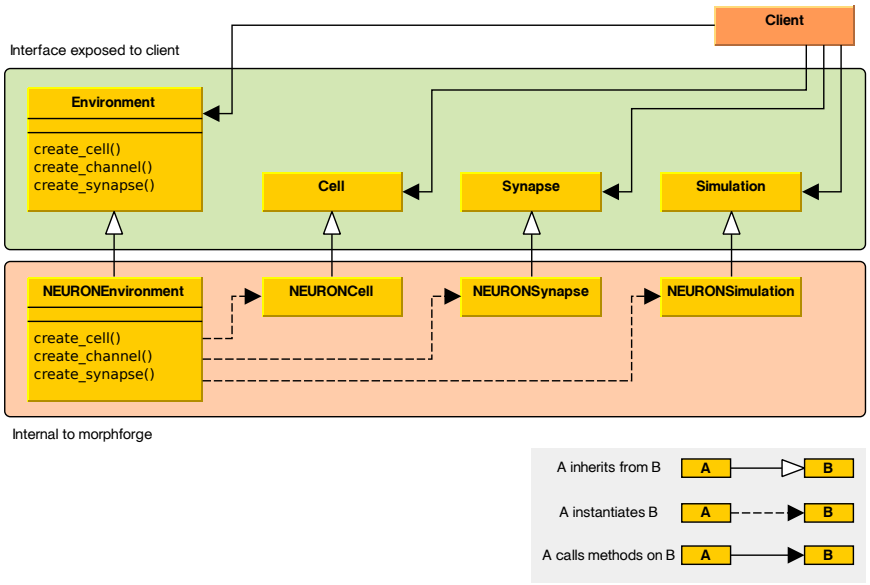


Figure D.1 – Using indirection to create simulation primitives using the `Environment` object. The client first instantiates a subclass of `Environment`, for example `NEURONEnvironment`. By constructing primitives through this object, an internal data structure of backend-specific objects (e. g. `NEURONCell`, `NEURONSynapse` and `NEURONSimulation`) can be created, but the interface only exposes *backend-independent* classes (e. g. `Cell`, `Synapse` and `Simulation`).

Using factories (ii) - channel creation

Several specialist file formats already exist for defining membrane channel and synaptic dynamics and parameters in simulators, for example MODL, NeuroML, NineML, or `neurounit`, and we would also like to be able to define channels and synapses in code directly. Rather than choose a single format for specifying primitive dynamics, morphforge uses Python's dynamic typing to support a flexible model for membrane channels and synapses. We begin by discussing channel models.

Morphforge is agnostic about the underlying format of channel models. We assume that an abstract channel can have a series of parameters, that can change in different areas of the membrane, and there is a set of default values for these parameters. To integrate with the morphforge framework, `Channel` objects are expected to provide a particular interface, some methods of which are general and some of which are simulator-backend specific. All `Channel` objects must implement the meth-

ods `get_variables()` & `get_defaults()` which return a list of parameter names for that `Channel`, (for example: `['g_bar','erev']`) and their default values respectively. These are used by the channel-distribution infrastructure in morphforge when calculating the parameter values which should be applied to each compartment of a `Cell` (this is described in Section D.1.1). Additionally, when the NEURON-backend is used, `Channel` objects must also implement the methods `create_modfile()` and `build_hoc_section()`, which build the MODL code and insert the relevant code into the HOC file.

As with the simulation primitives above, this is achieved by inheritance. For example, `NeuroMLChl` represents NeuroML channel loaded from an XML file, and is subclassed to produce the NEURON- specific `NEURONNeuroMLChl`. `NeuroMLChl`, implements the methods `get_variables()` and `get_defaults()`, which return the names of parameters that can be varied over the neuron, (for example `g_bar`) and their default values which are found by parsing the XML file. `NEURONNeuroMLChl` implements the methods `create_modfile()`, which returns a string of MODL code for the NeuroML file (for example generated using an XSLT) and `build_hoc_section()`, which returns the relevant HOC statements for inserting the `Channel` into a `Section` with a particular set of parameter values.

As with the simulation primitives, `Channel` objects are constructed using indirection through the `Environment` object, which allows morphforge to produce the appropriate backend-specific objects (Listing D.1). This architecture allows a modeller to use any supported formats for a particular simulator-backend. `Channel` types can be mixed and matched within a single `Simulation`, providing a path for incrementally translating their model into newer formats, for example NineML, NeuroML or `neurounit`, while still allowing the use of simulator-specific features.

```
env = NEURONEnvironment()
sim = env.Simulation()
# Create a series of channels of different types:
chl1 = env.Channel( NeuroMLChl, file='my_neuroml_channel.xml')
chl2 = env.Channel( NeuroUnitChl, file='my_neurounit_channel.eqn')
chl3 = env.Channel( NEURONMODChl, file='my_channel.mod')
chl4 = env.Channel( StdChlLeak, reversal_potential=qty('-50mV'), conductance=('0.03mS/cm2') )

# All channels are applied to the neuron in same way:
cell = sim.create_cell(...)
cell.apply_channel(chl1)
cell.apply_channel(chl2)
cell.apply_channel(chl3)
cell.apply_channel(chl4)
```

Listing D.1 – Constructing `Channel` objects via the `Environment`. By using the environment, NEURON-specific channel objects are created: `NEURONNeuroMLChl` (chl1); `NEURONNeuroUnitChl` (chl2); `NEURONMODChl` (chl3) & `NEURONStdChlLeak` (chl4).

Using flyweights to template postsynaptic receptors

In morphforge, as in NEURON [Carnevale and Hines, 2006], a `Synapse` is built from a pair of `Presynaptic` and `Postsynaptic` objects (Fig. D.2). `Presynaptic` objects generate *events*, for example from a list of times or in response to the voltage of a presynaptic neuron crossing a threshold (Fig. D.2). `PostSynaptic` objects represent synaptic receptors in the postsynaptic neuron, which open and close with particular dynamics and change the membrane conductance. `PostSynaptic` consume *events* to produce discrete changes in the receptor state.

Using this scheme allows the presynaptic and postsynaptic components of the synapse to be uncoupled; a particular postsynaptic receptor can be modelled and it is simple to change to the source of spike timing. Another advantage of this scheme is that if the model of the postsynaptic receptors are linearly superposable, then it is possible to replace multiple, identical postsynaptic receptors on a single neuron with a single instance which is driven by multiple event sources (Fig. D.3), which can dramatically reduce the number of equations that need to be solved by the simulator.

Different forms of equations and description languages have already been used to describe postsynaptic receptor dynamics. As with the definition of `Channel` objects (Section D.1.1), morphforge does not require a particular format, but expects that any `PostSynaptic` objects conform to a particular, backend-specific, interface. As

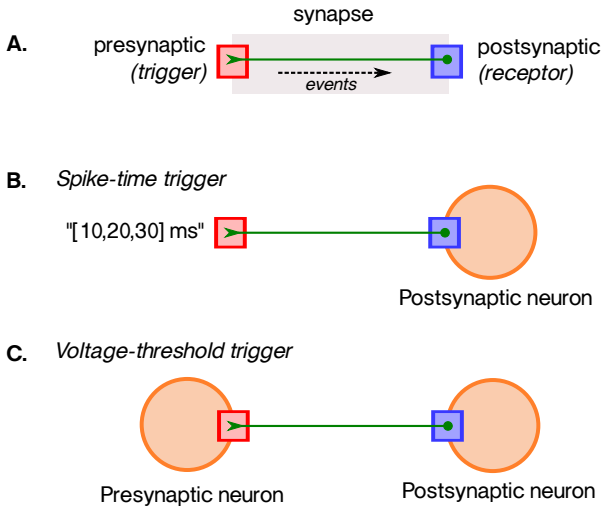


Figure D.2 – Synapses in morphforge. **A.** Synapses are composed of a presynaptic *trigger* and postsynaptic *receptor*. The trigger produces *events*, which cause discrete changes in the state of the receptor. **B.** The trigger can be either a set of event times or **C.** an object that monitors the voltage of a presynaptic neuron and produces an event each time a threshold is crossed.

with the `Channel` objects, `PostSynaptic` object are constructed indirectly, using the `Environment` factory object.

Although morphforge is not designed with efficiency as the first priority, efficiency must be considered with `Synapse` objects. Even in networks with a relatively small number of neurons, the number of synapses between them can quickly become very large. A recent model of the tadpole spinal cord has only 840 neurons, but approximately 180,000 synaptic connections [Borisyuk et al., 2008]. It is important to be able to specify large numbers of synapses, with variation in the parameters, which can be translated to efficient simulator-specific code.

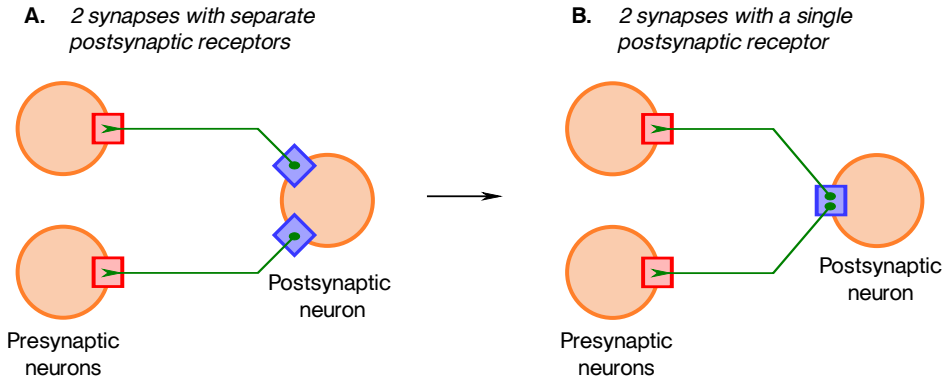


Figure D.3 – Merging postsynaptic receptors. **A.** A single postsynaptic neuron has synapses from two presynaptic neurons. The postsynaptic receptors (blue) are distinct objects. **B.** If the postsynaptic objects are linearly superposable, then we can replace the two postsynaptic objects with a single postsynaptic object, which is driven with events from the two presynaptic triggers.

```
synapse_def = """
eqnset syn_alpha {
   g' = -g/{2ms}
   g2' = -g2/{10ms}
   i = (g-g2) * (v-{0mV})
   gInc = 5pS
   <=> INPUT v: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
      <=> OUTPUT i:(mA) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }

      ==>> on_event(){
            g = g + gInc
            g2 = g2 + gInc
      }
}
"""
cell1 = sim.create_cell(...)
cell2 = sim.create_cell(...)
cell3 = sim.create_cell(...)

syn_tmpl = env.PostSynapticMechanismTemplate(NeuroUnitsTemplate, synapse_def)
sim.create_synapse(
            trigger = VoltageThresholdTrigger(cell_location = cell1.soma, v=0*mV),
            post_synaptic = env.PostSynaptic(NeuroUnitSynapse, synapse_def, ↙
                  ↳ cell_location=cell2.soma )
)
sim.create_synapse(
            trigger=VoltageThresholdTrigger(cell_location = cell1.soma, v=0*mV),
            post_synaptic = env.PostSynaptic(NeuroUnitSynapse, synapse_def, ↙
                  ↳ cell_location=cell3.soma )
)
```

Listing D.2 – Motivation for postsynaptic templates. In this example, we create three neurons, and two synapses (cell1 → cell2, cell1 → cell3). If we use the NEURON backend, it is difficult to infer that only a single MODL file needs to be built, which can be used for both of the synapses.

Morphforge takes an explicit approach to constructing multiple synapses of the same type by using the flyweight-pattern [Gamma et al., 1994]. This involves another factory class, `PostSynapticTemplate`, which has a method, `instantiate()`, which returns a new `PostSynapticObject`. This object delegates its backend-specific methods (e. g. `create_modfile()`) back to the parent `PostSynapticTemplate` object, to allow efficient code-generation. Since the `instantiate()` method can take parameters, it is possible to incorporate variation into the parameters of individual synapses. An example of using postsynaptic templates is given in Listing D.3.

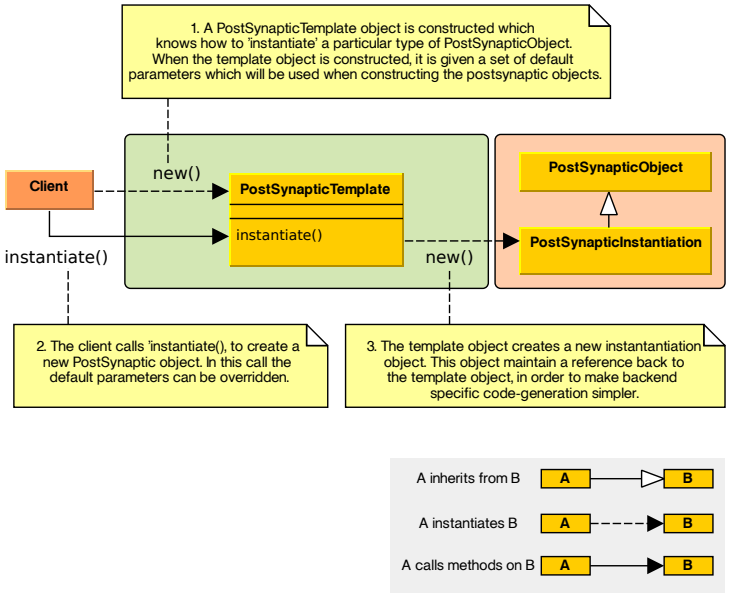


Figure D.4 – Overview of `PostSynapticTemplate` objects in morphforge.

Using strategy patterns to encapsulate concepts as objects

Abstractions are an effective tool for communicating ideas, allowing salient information to be conveyed whilst shielding the listener from less important details. In programming, suitable use of abstractions can make code much easier to understand and many techniques are used from naming constants to functions to make the conceptual aims of code more apparent [McConnell, 2004]. As our models grow in complexity,

```
# As before:
synapse_def = """
eqnset syn_alpha { ...
}
"""

syn_tmpl = env.PostSynapticMechanismTemplate(NeuroUnitsTemplate, synapse_def)
sim.create_synapse(
            trigger = [100*ms],
            post_synaptic = synaptic_tmpl.instantiate(cell_location = cell1.soma)
)
sim.create_synapse(
            trigger=VoltageThresholdTrigger(cell_location = cell1.soma, v=0*mV),
            post_synaptic=syn_tmpl.instantiate(
                              cell_location = cell2.soma,
                              parameter_multipliers={'g':2.0})
)
```

Listing D.3 – Example of defining synapses using a `PostSynapticTemplate` object.

we are likely to spend more time needing to reassimilate the intention of what we previously wrote, so it is important to be able to quickly understand the purpose of our own code. For example, in Listing D.4, although all three blocks of code achieve the same goal, calculating the diameter of sections as distance varies in order to produce a sphere, to the intention of the third version is more readily apparent than the first and second.

```
# Version 1
spine_sections = [
      (0., 0.00), (1., 3.00), (2., 4.00), (3., 4.58), (4., 4.90),
      (5., 5.00), (6., 4.90), (7., 4.58), (8., 4.00), (9., 3.00), (10., 0.00) ]

# Version 2
R = 5.
spine_sections = [(a, sqrt(R**2 - (x-R)**2)) for a in linspace(0.,10.,num=11)]

# Version 3
spine_sections = get_section_radii_for_spherical_spinehead(radius=5, nsections=11)
```

Listing D.4 – Making the intention of code explicit. The three examples given below may perform the same function, but it is much simpler to understand the authors intention from looking at Version 3 than Version 1.

There is a tradeoff in this indirection and defining interfaces which can encapsulate ideas yet still allow flexibility is difficult. In Listing D.4, if we wanted to change the shape of the sections to produce an oblate spheroid instead, then it is more immediately obvious what to change in Version 2 than Version 3. We must try to define modelling libraries that allow concepts to be defined at a suitably high level, so a simple statement like, 'remove all potassium channels' corresponds to a single line modification in the model code, but that these interfaces also offer sufficient flexibility and do not burden the modeller with learning a complex internal API.

One source of problems is that it has been difficult to separate orthogonal abstractions in code in mainstream languages. An example of an orthogonal abstraction is in sorting, where a general sorting algorithm such as quicksort is orthogonal to the comparison function between objects [Hoare, 1961; Press et al., 2007]. One long-standing difficulty in encapsulating data and algorithms - most languages allow data to be passed to functions, but often passing an algorithm to a function has involved either complex class hierarchies, difficult low-level syntax, or dangerous techniques. Modern languages such as Python makes it easy to build treat algorithms as objects, either by using callable objects or through duck typing. This technique is used extensively in morphforge to express concepts; two examples are given in the next sections: first, for defining the segmentation of `Cell`, and second for defining the distribution of `Channels` over a neuron.

(I) `cells` & SEGMENTATION A simple example of using a functor in morphforge is in the compartmentalisation of neurons. In morphforge, a `Cell` contains a `Morphology` object, which contains a tree of `Sections`, corresponding to cylinders. The `Morphology` and `Section` objects represent gross neuronal morphology. In electrical simulation these can be further subdivided, or *segmented*, into a number of smaller compartments for better spatial accuracy during simulation. Rather than specify the number of segments per `Section` directly, instead an object is passed as a parameter when a `Cell` object is constructed. This object implements the method `get_num_segments()`, which returns the number of segments for a given compartment. Listing D.5 shows 2 examples: the `CellSegmenter_MaxCompartmentLength` object on line 4 segments the morphology so that the longest compartment is 5 µm and

the `CellSegmenter_CompartmentLengthToDiameterRatio` object on line 6 makes compartments that are no longer than five times the length of the thinnest diameter of the `Section`. It is also simple to define custom classes, for example to implement the d-lambda rule [Carnevale and Hines, 2006] as illustrated in Listing D.6.

```
morphology = Morphology.load_from_swc('myfile.swc'),
cell = sim.create_cell(morphology = morphology,
      segmenter = CellSegmenter_MaxCompartmentLength(5)
      )

cell = sim.create_cell(morphology = morphology,
      segmenter = CellSegmenter_CompartmentLengthToDiameterRatio(5)
      )
```

Listing D.5 – Defining the segmentation of a morphology for simulation using objects defined in morphforge

```
class MyDLambdaSegmenter(AbstCellSegmenter):

   def __init__(self, d_lambda=0.1, frequency=100*units.Hz):
      self.d_lambda = d_lambda
      self.frequency = frequency

   def get_num_segments(self, section):
      # Query the model (not shown):
      Cm, Ra, = ...
      # Choose the thinnest radius from proximal and distal:
      d = min([section.p_r, section.d_r]) * 2.0 * units.um
      # Calculate d_lambda (see 'The NEURON book', pg. 122)
      lambda_100 = (1./2.) * math.sqrt( d / (pi*self.frequency*Ra*Cm))
      n_sections = int(ceil(sect.length / (lambda_100 * self.d_lambda)))
      # Ensure odd number of segments:
      if n_sections % 2 == 0:
         n_sections = n_sections + 1
      return n_sections

cell = sim.create_cell(morphology=morphology, segmenter=MyDLambdaSegmenter(d_lambda=0.1))
```

Listing D.6 – Defining the segmentation of a morphology for simulation using user-defined objects. Pseudo code for implementing the d-lambda rule is given

(II) CHANNELS & DISTRIBUTIONS In Section D.1.1, we discussed how a `Channel` object is created. We now discuss how it is applied to a neuron's membrane. In many neurons it is known that the distribution density of a particular channel type over

the membrane is not uniform. Often in models, we want to incorporate this, and specify that a `Channel` exists all over particular regions of the neuron, and use specific parameters in specific regions. For example, the conductance density of potassium channels might be $30\,\mathrm{mS/cm^2}$ all over on a model neuron's membrane, except in apical dendrites where it is $50\,\mathrm{mS/cm^2}$. Existing models have used even more complex channel distribution schemes, for example that the density of sodium channels on the initial segment of the axon should vary as the function of distance from the soma [Schmidt-Hieber et al., 2008].

Morphforge supports complex specifications of channel densities over neurons using a high-level notation. This is achieved by passing a triplet of objects to the `apply_channel` method of `Cell` objects: (`Channel`, `Applicator`,`Targeter`). The `Targeter` object defines which `Sections` in the `Cell` this triplet applies to (i. e. a predicate object). The `Applicator` object defines how the parameters of the `Channel` should vary over the specified `Sections`. Listing D.7 shows an example in which twice the density of potassium channels are applied in the "dendrites" as the rest of the `Cell`. In this example, we use two `Targeters`: `TargetEverywhere` and `TargetRegion`, and one `Applicator`: `ApplyUniform`. A `Channel` object has an associated set of default parameters (e. g. `gbar`, see Section D.1.1), which are used by default by `ApplyUniform` (e. g. Listing D.7 line 2), although they can be overridden or scaled (e. g. Listing D.7 line 3).

```
# Apply more potassium in the dendrites:
cell.apply_channel(k_chl, TargetEverywhere(), ApplyUniform() )
cell.apply_channel(k_chl, TargetRegion('dendrites'), ApplyUniform(multiply_parameters={'gbar':2.0}) )
```

Listing D.7 – Example of defining channel distributions in `morphforge`

The `apply_channel` method can be called many times for the same `Channel` on the same `Cell`, with different `Targeters` and `Applicators`. However, in the simulation, a particular `Channel` will only be applied once to any given `Section`. If multiple `Targeters` affect the same `Section`, a system is needed to resolve which parameter values to use. For example, in Listing D.7, which value of `gbar` should be applied to the dendrites — should it be the *default* (since the "dendrites" region will be targeted

by `TargetEverywhere`), or should twice the default (since the "dendrites" region will also be targeted by `TargetRegion`)?

To resolve these conflicts, each `Targeter` object has a *priority level* associated with it. For example `TargeterEverywhere` has a priority of 10, and `TargeterRegion` has a priority of 20. When `Simulation.run()` is called, for every `Channel` applied to every `Section`, morphforge finds the corresponding targeter with the highest priority. The algorithm described in Algorithm 1 in Appendix D.2.1. Therefore in Listing D.7, the dendrites will have twice the value of `g_bar` for `k_chl` in the dendrites than the rest of the neuron.

The mechanism also allows us to define non-uniform distributions of channels over a particular region. For example, we might want to distribute a type of sodium channel along the initial part of the axon, such that the channel density is specified as a function of distance from the soma. This is achieved by creating a new `Applicator` class, and implementing the method `get_variable_value_for_section()`. An example of this is shown in pseudocode in Listing D.8.

```
class CustomApplyDensityAsAFunctionOfDistance(ChannelApplicator):
   def get_variable_value_for_section(self, variable_name, section):
      if variable_name=='g_bar':
         # return a conductance density for 'section', that is some function of its
         # distance to the soma.
      else:
         # return a constant, default value for the other parameters (e.g. reversal potential)

# Apply more sodium non-uniformly in the axon:
cell.apply_channel(na_chl, TargetEverywhere(), ApplyUniform() )
cell.apply_channel(na_chl, TargetRegion('axon'), CustomApplyDensityAsAFunctionOfDistance() )
```

Listing D.8 – Example of defining a custom `ChannelApplicator` for defining a particular channel distribution on a neuron

(III) COMPLEXITY AND CLARITY In the previous sections, we have complicated morphforge by adding more indirection and classes into the framework. What are the benefits of this architecture? For one, once we understand the triplet (`Channel`, `Targeter`, `Applicator`) system, then the intention of Listings D.7 & D.8 becomes immediately clear. We do not need to decipher a complex set of `for`-loops and `if` statements, nor trust in comments in the code in order understand the intention, and we

have reduced the code needed to define a complex concept to a single line of code. In order to ask simple scientific questions, such as, what happens if the distribution of channels is simplified to be uniform, we are making changes at a single well-defined point in the code.

What are the advantages of this system, over simply defining allowing user-defined function that calculates the distribution of channels for each `Section` explicitly? Firstly, orthogonal issues, (such as iterating over the tree and calculating the conductances for each `Section`) are partitioned from each other. Since these have been decoupled, and there is no need to write any code for the iteration over `Section`s in Listing D.7, we reduce the likelihood of introducing bugs into our model. Secondly, we retain this abstraction within our object model. We may want to produce a summary output of the simulation as a pdf document for example. In this case, it will be much more useful to summarise the distribution of channels using the (`Channel`, `Targeter`, `Applicator`) notation, rather than as list of every parameter for every `Channel` for every compartment.

Using interfaces to keeping scientific questions centre-stage

As was touched upon in the previous section, we will need to read parts of our model many times, so it is important this is as easy as possible. Using interfaces allows us ignore implementation details, and focus on scientific questions. Simple and consistent interfaces are easier to understand and allow more mental resources to be dedicated to the conceptual problems at hand. Interfaces are widely used in morphforge where they are used to hide implementation details. I discuss how they are used to solve two particular problems, the first is in hiding the complexity of recording values from a simulation, and the second is a class to represent analogue signals with units.

(I) RECORDING VALUES DURING SIMULATIONS We need to record various values during a simulation, for example voltages and currents. The morphforge object-model agnostic to the underlying formats of particular synapse and channel formats and supports the recording of any values from any `Channel`, `Synapse` or other objects through a consistent interface using the method `Simulation.record()`, as shown in

Listing D.9. Internally, `record(obj,...)` forwards calls to `obj.get_recordable(...)`, which provides a flexible, yet consistent interface to recording what could be specific data from simulation objects (for example the voltage-dependence term of an NMDA synapse).

```
env = NEURONEnvironment()
sim = env.Simulation()

# Create a passive cell:
cell = CellLibrary.create_cell(sim, StandardModels.SingleCompartmentPassive, area=qty('1000:um2'), ↙
    ↳ input_resistance=qty('300:MOhm'))

# Add active channels:
chl = ChannelLibrary.get_channel( modelsrc = StandardModels.HH52, channeltype="Na", ↙
    ↳ env=sim.environment)
cell.apply_channel(chl)

cc = current_clamp = sim.create_currentclamp(amp=qty('100:pA'), dur=qty('100:ms'), ↙
    ↳ delay=qty('100:ms'), cell_location = cell.soma)
#syn = sim.create_synapse()

sim.record(cell, what='Voltage', name='V', cell_location=cell.soma)
sim.record(chl, what='StateVariable', state='m', cell_location=cell.soma)
sim.record(cc, what='Current')
#sim.record(syn, what='Current', user_tags=['mytag'])
```

Listing D.9 – Example of recording from different objects in a simulation

A set of *standard* strings, such as `Voltage`, `Current` and `StateVariable` are defined in morphforge, which are used in the method calls on lines [14-16] in order to specify what to record. The use of strings allows loose coupling across this interface and allows arbitrary variables to be recorded. `sim.record()` can take a variety of parameters, depending on the particular object being recorded from.

In order to provide a clean architecture behind the scenes, another object called `Recordable` is introduced (Fig. D.5). Following a call to `record(obj,...)`, the method `get_recordable()` of `obj` is called which returns a `Recordable` object. This contains the relevant machinery to record a particular value from `obj` using a particular simulator-backend. The `get_recordable()` method must internally determine which type of `Recordable` object to return, depending on the parameters passed. As a more concrete example, if the NEURON simulator backend is being used and if `get_recordable()` is called on a NeuroML channel to record the conductance density at a specific neuron location, then a `Recordable` object is returned that knows the name of the conductance variable in the generated MODL file (see Section D.1.1), and is able to insert

suitable statements into the HOC script to record and return this conductance after the simulation.

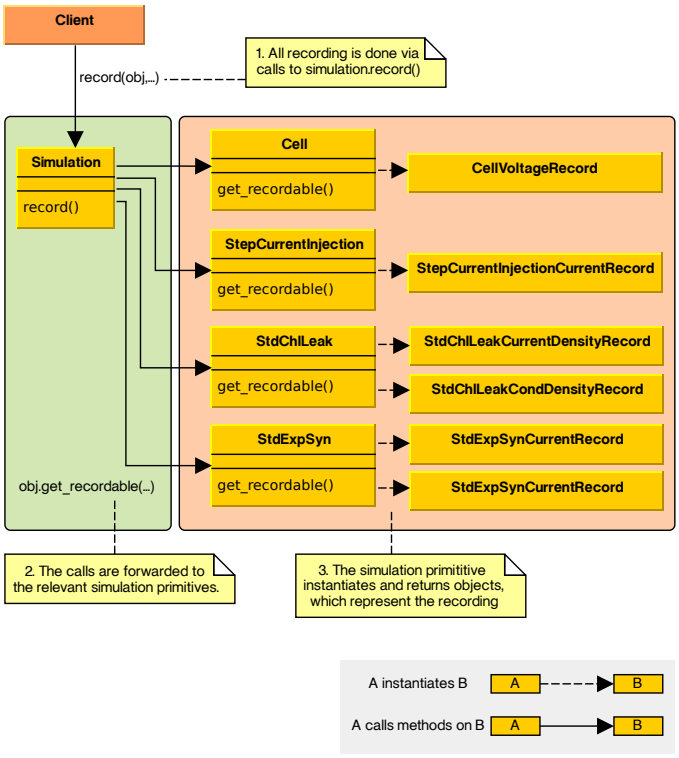


Figure D.5 – A simplified view of the architecture for recording in morphforge. All recording is done via calls to `Simulation.record()`, which forwards the calls onto the appropriate simulation primitives. The primitive objects return `Record` objects, which internally define what should be recorded, and are used by the simulator-backend when `Simulation.run()`is called. At the end of the simulation, the `SimulationResults` object is populated with `Trace` objects corresponding `Record` objects, (i. e. one for each call to `record()`).

A `Recordable` object can have a name (e. g. Listing D.9 line 14), which can be used to access the corresponding `Trace` object after the simulation has been run (see next section). A `Recordable` object can also take a set of additional user-defined tags, which will be attached to the `Trace` object (see Section D.1.2). The units of the recordings are automatically handled, for example, when the user requests the `Trace` object corresponding to the `record()` statement on line 14 of Listing D.9, it will automatically have the units of *millivolts*.

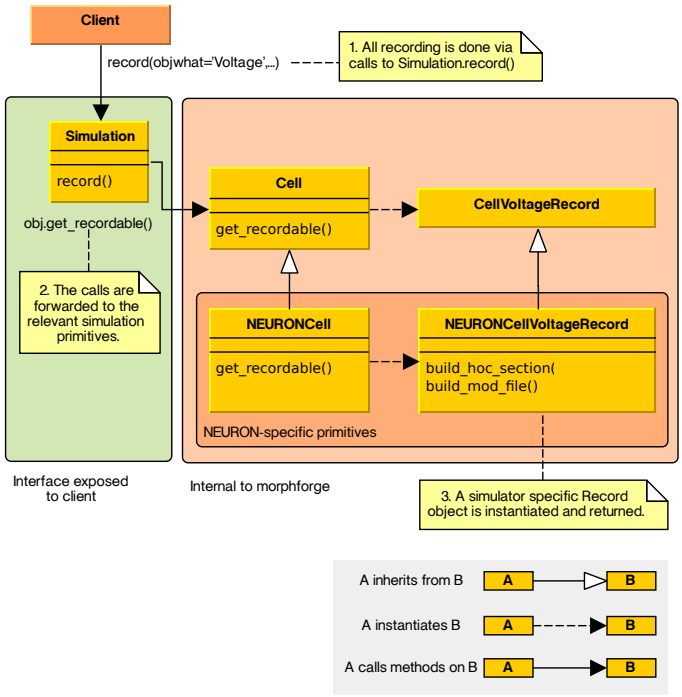


Figure D.6 – Construction of simulator dependant `Record` objects in morphforge. Simulation-backend specific `Record` objects (e. g. `NEURONCellVoltageRecord`) are created by calling `get_recordable()` on the simulation objects (e. g. `NEURONCell`). This allows a simple interface to be exposed to the client, through calls to `Simulation.record()`, but allows simulator-backend specific actions to be executed (e. g. `build_hoc_section()` and `build_mod_file()` methods for NEURON objects).

(II) `traces` After simulations have been run, we might want to perform a variety of analyses on the output. A common task is to analyse part of a recorded value, for example for plotting or finding spike times from a membrane voltage. The analysis might be more complex, for example calculating coupling coefficients between two electrically coupled neurons using their deviations from resting potential in a particular time window following a step current injection to one of the neurons . We might also want to compare a simulated membrane voltage trace against an experimentally recorded one for example to find the difference as was done in Chapter 2.

These analogue signals are often represented as an array of times and an array of values. In many cases, DEs can be solved efficiently using variable time-step integrat-

ors [Hindmarsh et al., 2005], which return values of the signal at irregularly spaced times. This means that even simple operations such as adding two signals and may require interpolation of values so that both signals use the same time values. Since the time and data points for a single recording are coupled, it is useful to encapsulate them together as a single object and define functions that operate on both together.

Morphforge builds a `Trace` class on top of the python-quantities and `neurounit` libraries and encapsulates a time and signal-value array. The design of the `Trace` class in morphforge supports both regular and irregularly spaced time-bases transparently. The class has basic methods such as `mean()` and `max()` for simple analysis, and additional methods can be added dynamically. Operators, such as `+-/*` are suitably overloaded and return new `Trace` objects. The `Trace` objects transparently handle units.

Listing D.10 illustrates how the input resistance of a neuron can be calculated. The simulation code is omitted, but we assume that the voltage, `V`, is recorded from a single neuron, which is given a step current injection of 30 pA from 100 ms to 200 ms. We assume that the resting potential is reached after 50 ms and the steady state after the current injection after 150 ms. This code will work regardless of whether variable time steps are used in the simulation, and the resulting object will automatically contain the correct units.

```
# <code to create, and run a simulation omitted>
v = res.get_trace('v')
result=(v.window(150, 200)*ms - v.window(50, 90)*ms).mean() / (-30*pA)
```

Listing D.10 – Calcuating input resistance using `Trace` objects

D.1.2 *`simulationanalysis` layer*

After a simulation has been run, a `SimulationResult` object is returned which contains a set of `Trace` objects (see Section D.1.1). Most of the tools in the `simulationanalysis` layer operate on these `SimulationResult` objects. I next discuss three features of the `simulationanalysis` layer: (i) how a system of *tags* can be used to quickly select

simulation records, (ii) how a high-level plotting library makes it simple to investigate simulation traces (iii) how a summary can be automatically generated from an object-model of a `Simulation` as a pdf-document.

Facilitating loose coupling by using a simple system of tags

In a small network simulation, we may want to visualise the internal states of many neurons and synapses — how do we effectively choose which values to plot or use in other forms of analysis? Imagine we have two populations of neurons, `P1` & `P2`, synapses form stochastically between each pair of neurons in `P1` and `P2` with a probability of 0.3, and we want to plot the conductances of the synapses.

One option would be to store a *handle* from each call to `record()` for the conductance of each synapse. After simulation had `run()`, these handles could be used to look up the corresponding `Trace` object in the `SimulationResult` object. Alternatively, each call to `record()` could be passed an explicit name, as is done on line 29 in Listing 3.5, and later use this string to retrieve the relevant results. However both mechanisms require adding complex code to the simulation script: if handles are used, it is necessary to track which handle refers to which synapse recording and if explicit string names are used, a suitable naming system will be required that can cope with the stochasticity in the number of synapses. The situation quickly becomes more complex, for example, to plot the conductance traces of all synapses which have postsynaptic receptors on a particular neuron from a particular source population.

To solve this problem, morphforge introduces a system of *tags* in order to quickly find `Trace` objects recorded in a simulation. Each `Trace` object contains a set of strings called *tags*, which are used to attach contextual information about the `Trace`. The tags can be specified by the user explicitly during the call to `Simulation.record()`, (for example Listing D.9) and morphforge will also add certain tags automatically. For example, `'Voltage'` or `'CurrentDensity'` will be automatically added if the `Trace` object represents a voltage or current density recording and when recording from a `Synapse` object, morphforge will automatically add the tags `'PRE:cell1'` and `'POST:cell2'` where `cell1` and `cell2` are the names of the presynaptic and postsynaptic neurons respectively (more examples are given in the documentation).

A simple string-based language has been designed, for selecting specific sets of `Trace` objects after a simulation has run. The language uses the keywords: `ALL`, `ANY`, `AND`, `OR` and `NOT`. The terms `ALL{A,B,..,C}` and `ANY{X,Y,...,Z}` are matching predicates which take comma separated arguments. `ALL{A,B,..,C}` returns whether a particular `Trace` contains *all* the tags specified (i. e. A, B, C) and `ANY{X,Y,...,Z}` returns whether a `Trace` contains any of the tags specified (i. e. X, Y, Z). These match predicates can be joined with the `AND`, `OR` and `NOT` operators as well as brackets to allow more complex queries. For example, `ALL{Voltage}` will return all the voltages recorded in the simulation and `ALL{CONDUCTANCE,SYNAPTIC,PRE:cell1,POST:cell2} AND ANY{NMDA,AMPA}` could be used to retrieve all `Trace` objects representing conductances in AMPAR and NMDAR synapses from *cell1* to *cell2*.

This system of tagging, and the use of conventions such as *voltage traces always have a 'Voltage' tag* allows looser coupling between different parts of the code and allows more scripts to be more succinct.

Using conventions to simplify plotting

The `Trace` object contains methods, `time_pts_in()` & `data_pts_in()` that return `numpy` arrays of their time and data converted to specific units. These arrays can then be used for analysis with other scientific Python libraries. Listing D.11 shows how a `Trace` object can be plotted using the library matplotlib. Although this is one way of plotting results, morphforge provides another class `TagViewer` which makes it much less verbose to plot a selection of `Trace` objects from a simulation.

```
my_voltage_trace = result.get_trace(...)
pylab.plot(my_voltage_trace.time_pts_in('ms'), my_voltage_trace.data_pts_in('mV') )
```

Listing D.11 – Plotting a `Trace` object with matplotlib

The output of the `TagViewer` is a single figure, containing a series of axes with the same time base. The details of each axis, such as the y-label, the appropriate display range and unit are specified by `PlotSpec` objects. The `PlotSpec` object also takes a tag-selection string, to define which `Traces` should be plot on that axis, and rather than needing to explicitly specify which traces should be plot, the `TagViewer` object

directly queries the `SimulationResults` object. An example is given in Listing D.12, in which two axes will be displayed, one in which contains all `Traces` containing the tag 'Voltage', and another which contains `Traces` with both *Voltage* and *cell47* as tags. `TagViewer` objects have a set of `PlotSpecs` that are used by default and will automatically plot `'Voltage'`, `'Current'`, `'Conductance'` and other *standard-tags*. More examples of the figures generated for different simulations by `TagViewer` are given in Appendix E.

```
results = simulation.run()
TagViewer(results,
      specs = [
            PlotSpec('Voltage',....)
            PlotSpec('ALL{Voltage,cell47}', ....)
])
```

Listing D.12 – Plotting the results of a simulation with `TagViewer`

Producing self-documenting models

Models in computational neuroscience involve complex equations, many units and have large numbers of parameters; a typical HH-type sodium channel has 7 equations involving 12 parameters. Often modelling involves adjusting parameters. How do we keep track of which parameters produce which results? Experimentalists use lab notebooks as a way to record protocol setups but manually noting all the details of a complex simulation is unfeasible. One approach is to use version control, for example Sumatra [Davison, 2012]. An alternative approach is to generate summaries of a simulation from the internal object-model to produce a human readable output directly. The need for standard presentation formats for models has been recognised, even if exact formats have not yet been defined (e. g. [Nordlie et al., 2009; Nordlie and Plesser, 2010; Crook et al., 2012]).

Morphforge supports the production of html and pdf-document summaries from `Simulation` objects directly using `mredoc`, (Modular Reduced Documentation) library. This library is a high-level interface for producing documents containing images, tables, code-snippets and equations for documenting mathematical models. After the `Simulation` object has been populated, it can be summarised as shown in Listing D.13

```
sim = env.Simulation()

# Populate the simulation ...
sim.create_cell(...)
sim.create_synapse(...)

# Summarise the object
SummaryManager.summarise(simulation).to_pdf('~/mysimulation.pdf')
```

Listing D.13 – Building summaries of simulations in morphforge

Simulations in morphforge can be populated with `Synapse` and `Channel` objects of different types, for example NineML, NeuroML & `neurounit`. The summary architecture allows these objects to create summaries of themselves. An example of summarising a simulation and the resulting pdf document are given in Appendix E.

D.2 IMPLEMENTATION DETAILS

D.2.1 *Defining channel distributions*

The distribution of channels on a neuron is specified by calls to `Cell.apply_channel( chl, applicator, targeter)`, abbreviated `C,A,T`, described in Section D.1.1. Because multiple calls to `apply_channel()` are allowed, in the case that for a single `Channel` object, different parameters are specified to be specified to the same `Section`, a system of priorities is used to determine which parameters should be used. The targeter object has a method `get_priority_level()` which returns an integer. For each `Section`, for each `Channel`, the relevant triplet,`(c,a,t)` in which the targeter object has the highest priority is chosen. The pseudocode for this process is given in Algorithm 1.

Algorithm 1: The algorithm used by morphforge to resolve which parameters are used by a channels in different `Sections` of a neuron

```
Data: all_sections, cat_list
for s in all_sections do
    chls ⟵ ∅ ;
    cat_target ⟵ ∅ ;
    for (c, a, t) in cat_list do
        if t.does_target(s) then
            if c not in chls then
                chls.add(c) ;
            cat_target.add((c, a, t)) ;
    for chl in chls do
        (hp, hpa) ⟵ (0, None) ;
        hp_ambiguous ⟵ False ;
        for (c, a, t) in cat_targets do
            if t.priority = hp then
                hp_ambiguous ⟵ True ;
            if t.priority > hp then
                hp_ambiguous ⟵ False ;
                hp, hpa ⟵ (t.priority, a) ;
        if hp_ambiguous == False then
            Apply chl to s using hpa ;
        else
            Raise an exception. ;
```

D.2.2 *Tag selection grammar*

`Trace` objects in morphforge can be selected using *tag-selection* strings, which specify which tags an object must have to be selected (see Section D.1.2). Examples are given in Listing D.14. A simple BNF grammar to parse these strings in Grammar D.1. The starting symbol for the grammar is `expr`. A `<TAG>` token must start with letter, followed by any number of alphanumeric characters, underscore, dash, period and colon. (defined by the Python regular expression `r"'[a-zA-Z_][a-zA-Z0-9_.:-]*"'`). The precedence rules for `and`, `or` and `not` operators are the same as those in C [Kernighan and Ritchie, 1988].

```
TagSelect('ALL{Voltage}')
TagSelect('ALL{Voltage} AND ANY{cell1,cell2,cell3}')
TagSelect('{ALL{Current,cell1} OR ( ALL{Synaptic,Current,POST:cell1} AND ANY {NMDA,AMPA} ) }')
```

Listing D.14 – Example tag-selection strings, which could be used to select particular `Trace` objects after a simulation

⟨*expr*⟩ ::= ⟨*tag_term_factor*⟩
| ⟨*tag_line_simple*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*expr*⟩ 'AND' ⟨*expr*⟩
| ⟨*expr*⟩ 'OR' ⟨*expr*⟩
| 'NOT' ⟨*expr*⟩

⟨*tag_item_simple*⟩ ::= ⟨*TAG*⟩

⟨*tag_line_simple*⟩ ::= ⟨*TAG*⟩
| ⟨*tag_line_simple*⟩ ',' ⟨*TAG*⟩

⟨*tag_term_factor*⟩ ::= ⟨*tag_item_simple*⟩
| ⟨*tag_group_factor_all*⟩
| ⟨*tag_group_factor_any*⟩

⟨*tag_group_factor_all*⟩ ::= 'ALL' ⟨*tag_group_bracketed*⟩

⟨*tag_group_factor_any*⟩ ::= 'ANY' ⟨*tag_group_bracketed*⟩

⟨*tag_group_bracketed*⟩ ::= '' ⟨*tag_group*⟩ ''

⟨*tag_group*⟩ ::= ⟨*empty*⟩
| ⟨*TAG*⟩
| ⟨*tag_group*⟩ ',' ⟨*TAG*⟩

Grammar D.1 – The BNF grammar used to define a syntax for selecting objects based on a system of tags

E

MORPHFORGE EXAMPLE SIMULATIONS

E.1 CURRENT INJECTION INTO A SINGLE COMPARTMENT NEURON WITH HH-TYPE CHANNELS

In this example, a single-compartment neuron containing HH-type leak, sodium and potassium channels is stimulated with a step current injection (Listing E.1). This produces the graphs shown in Figure E.1.

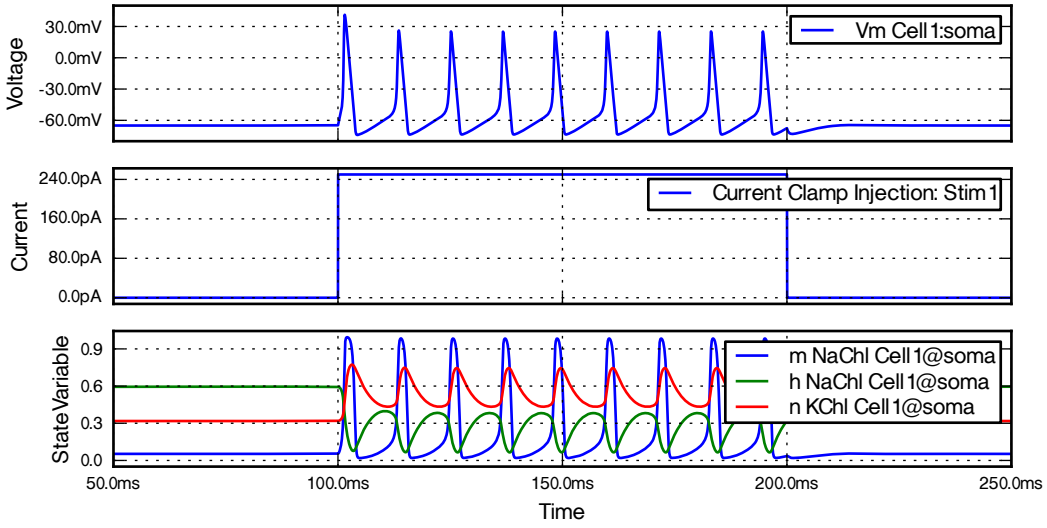


Figure E.1 – The output figure produced from running Listing E.1. The graphs show the membrane voltage, the injected current, and the state variables of the channels during simulation.

```
from morphforge.stdimports import *
from morphforgecontrib.stdimports import *

#from morphforgecontrib.simulation.channels.hh_style.core.mmleak import StdChlLeak
#from morphforgecontrib.simulation.channels.hh_style.core.mmalphabeta import StdChlAlphaBeta

# Create the environment & simulation:
env = NEURONEnvironment()
sim = env.Simulation()

# Create a cell:
cell = sim.create_cell(name="Cell1", morphology=MorphologyBuilder.get_single_section_soma(rad=10))

lk_chl = env.Channel(StdChlLeak, name="LkChl",
                conductance=qty("0.3:mS/cm2"), reversalpotential=qty("-54.3:mV"))

na_state_vars = { "m": {
                  "alpha":[-4.00,-0.10,-1.00,40.00,-10.00],
                  "beta": [4.00, 0.00, 0.00,65.00, 18.00]},
                 "h": {
                    "alpha":[0.07,0.00,0.00,65.00,20.00] ,
                    "beta": [1.00,0.00,1.00,35.00,-10.00]}}

na_chl = env.Channel(StdChlAlphaBeta,
                name="NaChl", equation="m*m*m*h",
                conductance=qty("120:mS/cm2"), reversalpotential=qty("50:mV"),
                statevars=na_state_vars)

k_state_vars = { "n": {
                  "alpha":[-0.55,-0.01,-1.0,55.0,-10.0],
                  "beta": [0.125,0,0,65,80]}}

k_chl = env.Channel(StdChlAlphaBeta,
                name="KChl", equation="n*n*n*n",
                conductance=qty("36:mS/cm2"), reversalpotential=qty("-77:mV"),
                statevars=k_state_vars)

# Apply the channels uniformly over the cell, and set capacitance
cell.apply_channel(lk_chl)
cell.apply_channel(na_chl)
cell.apply_channel(k_chl)
cell.set_passive( PassiveProperty.SpecificCapacitance, qty('1.0:uF/cm2'))

# Create the stimulus and record the injected current:
cc = sim.create_currentclamp(
         name="Stim1", cell_location=cell.soma,
         amp=qty("250:pA"), dur=qty("100:ms"), delay=qty("100:ms"))

# Define what to record:
sim.record(cc, what=StandardTags.Current)
sim.record(cell, what=StandardTags.Voltage, name="SomaVoltage", cell_location = cell.soma)
sim.record(na_chl, what=StandardTags.StateVariable, state='m', cell_location = cell.soma)
sim.record(na_chl, what=StandardTags.StateVariable, state='h', cell_location = cell.soma)
sim.record(k_chl, what=StandardTags.StateVariable, state='n', cell_location = cell.soma)

# Run the simulation & display the results:
results = sim.run()
TagViewer(results, timerange=(50, 250)*units.ms, show=True)
```

Listing E.1 – An example simulation using morphforge, in which an HH-type neuron is stimulated with a step current injection.

E.2 THE EFFECT OF AXONAL RADIUS ON THE SPEED OF ACTION POTENTIAL PROPAGATION.

In this example, a neuron consisting of a soma with a long axon, which containing HH-type leak, sodium and potassium channels is stimulated in the soma with a short step current injection to initiate an action potential (Listing E.2). The voltage is recorded at points along the axon to show the action potential propagation. Three simulations are run, in which the radius of the axon is 0.2, 0.4 and 0.6 µm. This produces the graphs shown in Figure E.2.

```
from morphforge.stdimports import *
from morphforgecontrib.data_library.stdmodels import StandardModels


def sim(axon_radius, tag):
    # Create the environment:
    env = NEURONEnvironment()

    # Create the simulation:
    sim = env.Simulation()

    # Create a cell:
    morph = MorphologyBuilder.get_soma_axon_morph(axon_length=3000.0, axon_radius=axon_radius, ↙
        ↳ soma_radius=9.0, axon_sections=20)
    cell = sim.create_cell(name="Cell1", morphology=morph)
    lk_chl = ChannelLibrary.get_channel(modelsrc=StandardModels.HH52, channeltype="Lk", env=env)
    na_chl = ChannelLibrary.get_channel(modelsrc=StandardModels.HH52, channeltype="Na", env=env)
    k_chl = ChannelLibrary.get_channel(modelsrc=StandardModels.HH52, channeltype="K", env=env)

    # Apply the channels uniformly over the cell
    cell.apply_channel(lk_chl)
    cell.apply_channel(na_chl)
    cell.apply_channel(k_chl)

    # Over-ride the parameters in the axon:
    cell.apply_channel(channel=lk_chl, where="axon", parameter_multipliers={'gScale':1.2})
    cell.apply_channel(channel=na_chl, where="axon", parameter_multipliers={'gScale':1.2})

    # Record at 100um intervals along the axon
    for cell_location in CellLocator.get_locations_at_distances_away_from_dummy(cell=cell, ↙
        ↳ distances=range(9, 3000, 100)):
        sim.record(cell, what=StandardTags.Voltage, cell_location=cell_location, user_tags=[tag])
```

```
    # Create the stimulus and record the injected current:
    cc = sim.create_currentclamp(name="Stim1", amp=qty("250:pA"), dur=qty("5:ms"), ↙
        ↳ delay=qty("100:ms"), cell_location=cell.soma)
    sim.record(cc, what=StandardTags.Current,user_tags=[tag])

    # run the simulation
    return sim.run()



results = [
    sim(axon_radius=0.2, tag="SIM1"),
    sim(axon_radius=0.4, tag="SIM2"),
    sim(axon_radius=0.6, tag="SIM3"),
]

kw = { 'yrange':(-80, 50)*units.mV, 'legend_labeller':None, 'yticks':3}
TagViewer(results, timerange=(97.5, 120)*units.ms, show=False,
    plots = [
        TagPlot("ALL{Current,SIM1}", ylabel='IInj', legend_labeller=None),
        TagPlot("ALL{Voltage,SIM1}", ylabel='R:0.2um\nVoltage', **kw),
        TagPlot("ALL{Voltage,SIM2}", ylabel='R:0.4um\nVoltage', **kw),
        TagPlot("ALL{Voltage,SIM3}", ylabel='R:0.6um\nVoltage', **kw),
    ])
```

Listing E.2 – An example of running three simulations in a single script using morphforge. An HH-type multicompartmental neuron with an axon is stimulated with a step current injection. The radius of the axon is set to 0.2, 0.4 and 0.6 µm in the three simulations. The current injected into the first stimulation is shown on the top graph, and the voltages recorded along the neurons' axons in the three experiments are shown in the graphs below.

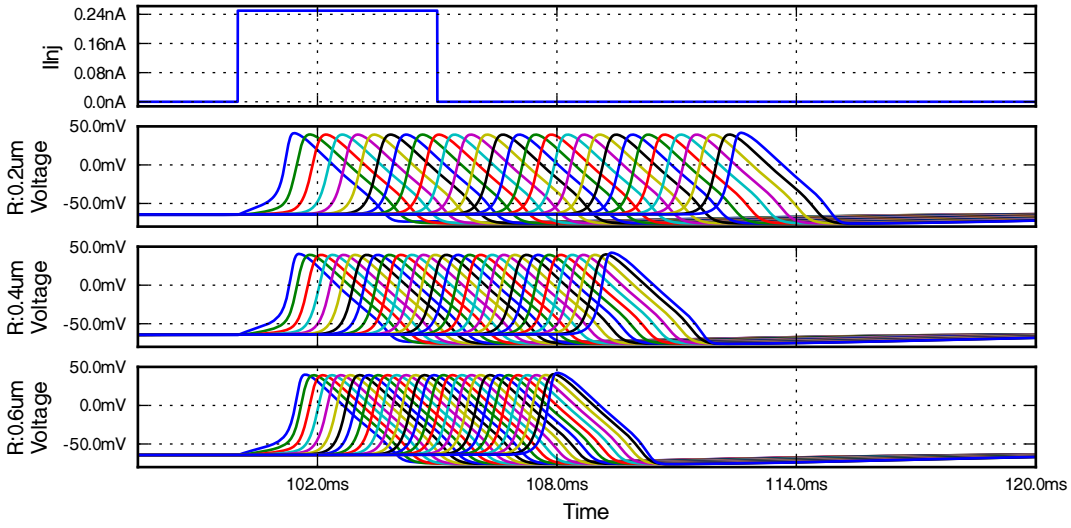


Figure E.2 – The output figure produce from running Listing E.2. The graphs show the effect of axon diameter on action potential propagation velocity.

E.3 SIMULATION OF THREE NEURONS USING NEUROUNIT TO DEFINE SYNAPSES

In this example, three neurons are created (cell1, cell2 & cell3), each of which have a soma and a long axon. A PostSynaptic template is created from a neurounit string (Listing E.1). Two synapses are instantiated using this template, from cell1 → cell2 and cell1 → cell3, in which the presynaptic trigger is at different points on the axon of cell1. The conductance of the second synapse is twice that of the first. This produces the graphs shown in Figure E.3. This simulation also produces a summary pdf document, shown in Figure E.4.

```
from morphforge.stdimports import *
from morphforgecontrib.stdimports import *

# Create a cell:
def build_cell(name, sim):
    morph = MorphologyBuilder.get_soma_axon_morph(axon_length=1500., axon_radius=0.3, ↙
        ↳ soma_radius=10.0)
    cell = sim.create_cell(name=name, morphology=morph)
    model = StandardModels.HH52
    na_chls = ChannelLibrary.get_channel(modelsrc=model, channeltype="Na", env=sim.environment)
    k_chls = ChannelLibrary.get_channel(modelsrc=model, channeltype="K", env=sim.environment)
    lk_chls = ChannelLibrary.get_channel(modelsrc=model, channeltype="Lk", env=sim.environment)
```

```
    cell.apply_channel(lk_chls)
    cell.apply_channel(k_chls)
    cell.apply_channel(na_chls)
    cell.apply_channel(na_chls, where="axon", parameter_multipliers={'gScale':1.2})
    return cell


# Create a simulation:
env = NEURONEnvironment()
sim = env.Simulation(name='Example3_Test_Neurounit_Synapse')

# Create three cells
cell1 = build_cell(name="cell1", sim=sim)
cell2 = build_cell(name="cell2", sim=sim)
cell3 = build_cell(name="cell3", sim=sim)

# Define a synapse in NeuroUnit format, and use it to build a template:
simple_ampa_syn = """
eqnset syn_simple {
    g' = - g/g_tau
    i = gmax * (v-erev) * g

    gmax = 300pS * scale
    erev = 0mV
    g_tau = 10ms

    ==>> on_event() {
        g = g + 1.0
    }

    <=> INPUT v: mV METADATA {"mf":{"role":"MEMBRANEVOLTAGE"} }
    <=> OUTPUT i:(mA) METADATA {"mf":{"role":"TRANSMEMBRANECURRENT"} }
    <=> PARAMETER scale:()
}"""

post_syn_tmpl = env.PostSynapticMechTemplate(NeuroUnitEqnsetPostSynaptic,
      name = 'MyTemplate', eqnset = simple_ampa_syn, default_parameters = { 'scale':1.0} )

# Create two instances of the synaptic template: cell1->cell2 and cell1-> cell3
syn1 = sim.create_synapse(
      name='Syn1',
      trigger = env.SynapticTrigger(
          SynapticTriggerByVoltageThreshold,
          cell_location = CellLocator.get_location_at_distance_away_from_dummy(cell1, 300),
          voltage_threshold = qty("0:mV"), delay=qty("0:ms") ),
      postsynaptic_mech = post_syn_tmpl.instantiate(
```

```
        cell_location = cell2.soma)
    )

syn2 = sim.create_synapse(
    name='Syn2',
    trigger = env.SynapticTrigger(
        SynapticTriggerByVoltageThreshold,
        cell_location = CellLocator.get_location_at_distance_away_from_dummy(cell1, 700),
        voltage_threshold = qty("0:mV"), delay = qty("0:ms") ),
    postsynaptic_mech = post_syn_tmpl.instantiate(
        cell_location = cell3.soma,
        parameter_overrides={'scale':2.0})
    )

# Record Voltages from axons:
record_locs = CellLocator.get_locations_at_distances_away_from_dummy(cell1, range(0, 1000, 50))
for loc in record_locs:
   sim.record( what=StandardTags.Voltage, cell_location = loc, user_tags=['cell1'])
sim.record(what=StandardTags.Voltage, cell_location=cell2.soma, user_tags=['cell2'])
sim.record(what=StandardTags.Voltage, cell_location=cell3.soma, user_tags=['cell3'])

# Create the stimulus and record the injected current:
cc = sim.create_currentclamp(name="CC1", amp=qty("200:pA"), dur=qty("1:ms"), delay=qty("100:ms"), ↙
    ↳ cell_location=cell1.soma)
sim.record(cc, what=StandardTags.Current)

results = sim.run()
mV = units.mV
t = TagViewer(results,
   timerange=(98, 120)*units.ms,
   plots = [
      TagPlot('Current', yunit=units.picoamp),
      TagPlot('Voltage,cell1', ylabel='Presynaptic\nVoltages', yrange=(-80, 50)*mV, yunit=mV, ↙
          ↳ legend_labeller=False),
      TagPlot('Voltage AND ANY{cell2,cell3}', ylabel='Postsynaptic\nVoltages',yrange=(-70, -55)*mV, ↙
          ↳ yunit=units.mV),
        ],
      )
# Build the summary:
options = SummariserOptions()
options.include_details_indvidual_neuron_morphology_mpl = False
SimulationMRedoc.build(sim, options=options).to_pdf('app_ex3_summary.pdf')
```

Listing E.3 – An example simulation containing three neurons and two synapses. The neurons are all multicompartmental and have a soma and a long axon. The neurons are connected via two synapses, (cell1 → cell2 and (cell1 → cell3) and the kinetics of the synapses are defined using neurounit.

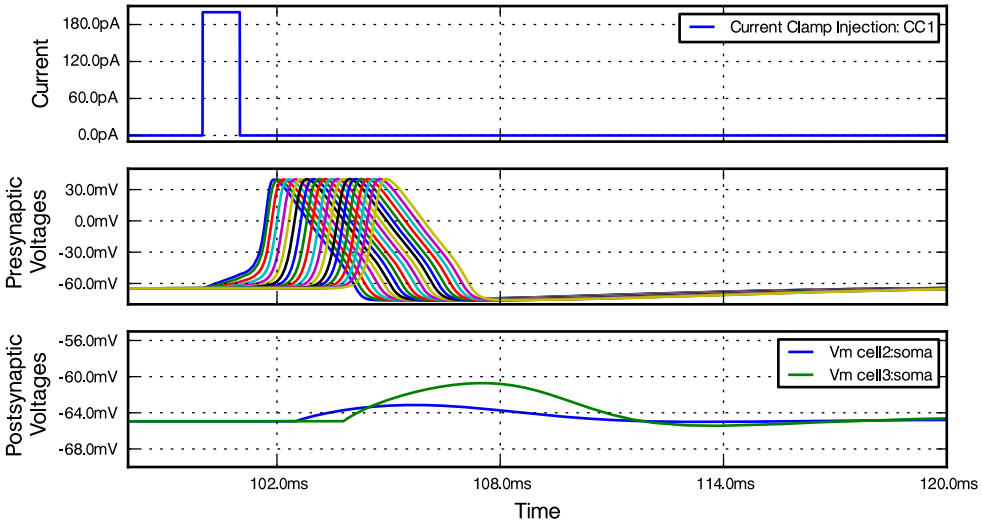
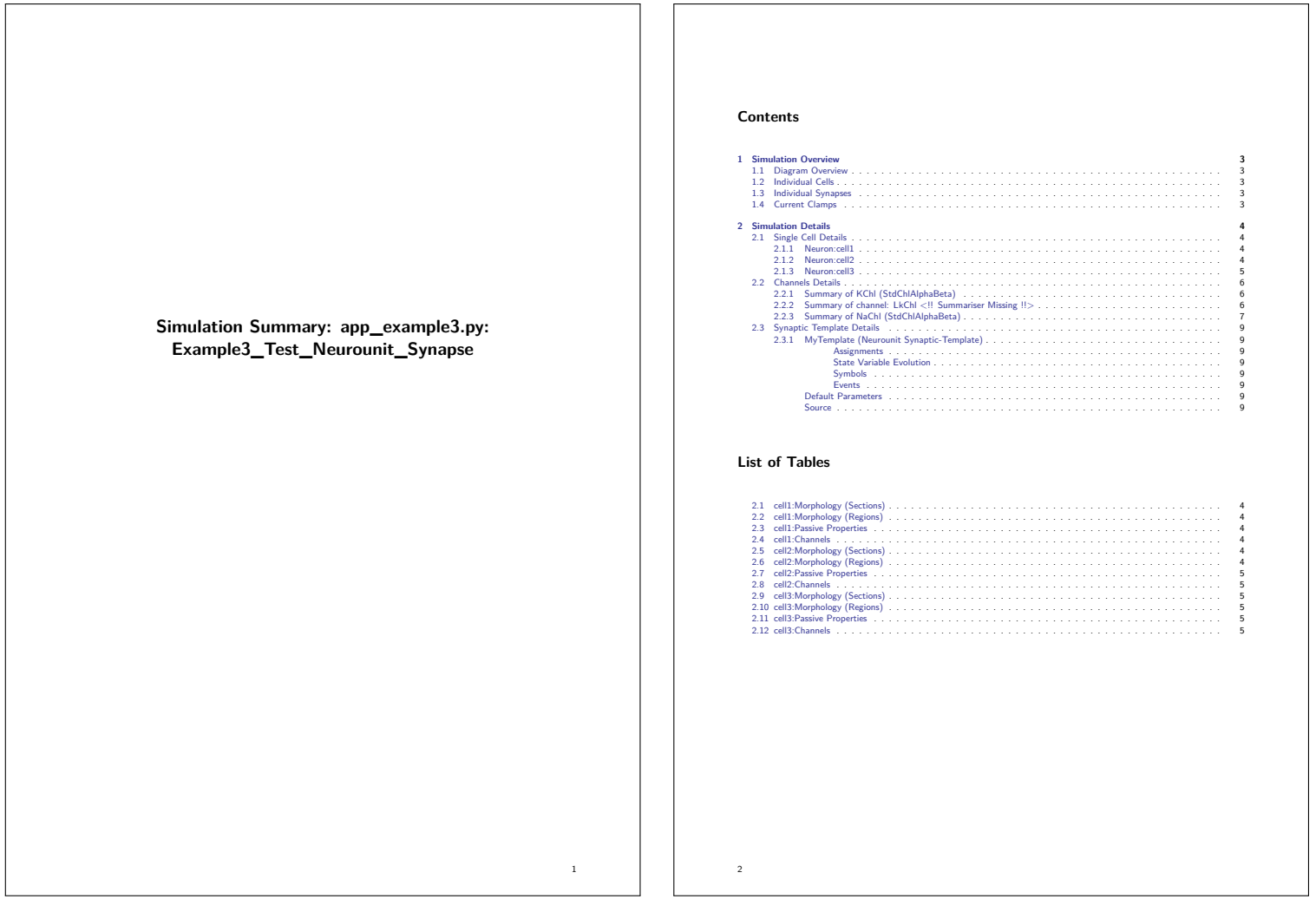


Figure E.3 – The output figure produce from running Listing E.3



Simulation Summary: app_example3.py: Example3_Test_Neurounit_Synapse

1

Contents

1 Simulation Overview 3
1.1 Diagram Overview 3
1.2 Individual Cells 3
1.3 Individual Synapses 3
1.4 Current Clamps 3

2 Simulation Details 4
2.1 Single Cell Details 4
2.1.1 Neuron:cell1 4
2.1.2 Neuron:cell2 4
2.1.3 Neuron:cell3 5
2.2 Channels Details 6
2.2.1 Summary of KChl (StdChlAlphaBeta) 6
2.2.2 Summary of channel: LkChl <!! Summariser Missing !!> 6
2.2.3 Summary of NaChl (StdChlAlphaBeta) 7
2.3 Synaptic Template Details 9
2.3.1 MyTemplate (Neurounit Synaptic-Template) 9
Assignments 9
State Variable Evolution 9
Symbols 9
Events 9
Default Parameters 9
Source 9

List of Tables

2.1 cell1:Morphology (Sections) 4
2.2 cell1:Morphology (Regions) 4
2.3 cell1:Passive Properties 4
2.4 cell1:Channels 4
2.5 cell2:Morphology (Sections) 4
2.6 cell2:Morphology (Regions) 4
2.7 cell2:Passive Properties 5
2.8 cell2:Channels 5
2.9 cell3:Morphology (Sections) 5
2.10 cell3:Morphology (Regions) 5
2.11 cell3:Passive Properties 5
2.12 cell3:Channels 5

2

Figure E.4 – The summary pdf document produced by Listing E.3

1 Simulation Overview

1.1 Diagram Overview

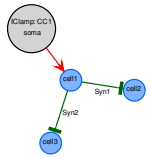


Figure 1.1

1.2 Individual Cells

Name	Type	SA(um2)	#sections/segments	Regions(SA(um2):nseg)	#Pre/Post-Synapse	#GapJunctions	Chls
cell1	<?>	8382	10:284	axon(7125:279) soma(1256:5)	2 0	0	LkChl NaChl KChl
cell2	<?>	8382	10:284	axon(7125:279) soma(1256:5)	0 1	0	LkChl NaChl KChl
cell3	<?>	8382	10:284	axon(7125:279) soma(1256:5)	0 1	0	LkChl NaChl KChl

1.3 Individual Synapses

Name	Type	Trigger	PostSynaptic Cell	Receptor
Syn1	<?>	cell1@axon_2: [threshold: 0.0 mV]	cell2@soma	<Defined through NeuroUnit: 'MyTemplate' >
Syn2	<?>	cell1@axon_5: [threshold: 0.0 mV]	cell3@soma	<Defined through NeuroUnit: 'MyTemplate' Overrides: 'scale': 2.0>

1.4 Current Clamps

Name	Location	Description
CC1	cell1 (0um from soma)	Step-Change: amp=2e-10 A dur=0.001 s delay=0.1 s

2 Simulation Details

2.1 Single Cell Details

2.1.1 Neuron:cell1

ID	Tags	Lateral Surface Area (um2)	Region	nseg	L	diam (prox/dist)
0	soma	1257	soma	5	20.0	20.0/20.0
1	axon_1	4864	axon	31	150.0	20.0/0.6
2	axon_2	283	axon	31	150.0	0.6/0.6
3	axon_3	283	axon	31	150.0	0.6/0.6
4	axon_4	283	axon	31	150.0	0.6/0.6
5	axon_5	283	axon	31	150.0	0.6/0.6
6	axon_6	283	axon	31	150.0	0.6/0.6
7	axon_7	283	axon	31	150.0	0.6/0.6
8	axon_8	283	axon	31	150.0	0.6/0.6
9	axon_9	283	axon	31	150.0	0.6/0.6

Table 2.1 – cell1:Morphology (Sections)

Region	Surface Area	#Sections
axon	7125.84544684	9
soma	1256.63706144	1

Table 2.2 – cell1:Morphology (Regions)

PassiveProp	Priority	Targetter	Value
AxialResistance	0	Default	80.0 ohmcm
SpecificCapacitance	0	Default	1.0 uF/cm2

Table 2.3 – cell1:Passive Properties

Mechanism	Priority	Targetter	Applicator
LkChl	10	Everywhere	Uniform Applicator:
KChl	10	Everywhere	Uniform Applicator:
NaChl	10	Everywhere	Uniform Applicator:
NaChl	20	Region: axon	Uniform Applicator: Multipliers:gScale=1.2

Table 2.4 – cell1:Channels

2.1.2 Neuron:cell2

ID	Tags	Lateral Surface Area (um2)	Region	nseg	L	diam (prox/dist)
0	soma	1257	soma	5	20.0	20.0/20.0
1	axon_1	4864	axon	31	150.0	20.0/0.6
2	axon_2	283	axon	31	150.0	0.6/0.6
3	axon_3	283	axon	31	150.0	0.6/0.6
4	axon_4	283	axon	31	150.0	0.6/0.6
5	axon_5	283	axon	31	150.0	0.6/0.6
6	axon_6	283	axon	31	150.0	0.6/0.6
7	axon_7	283	axon	31	150.0	0.6/0.6
8	axon_8	283	axon	31	150.0	0.6/0.6
9	axon_9	283	axon	31	150.0	0.6/0.6

Table 2.5 – cell2:Morphology (Sections)

Region	Surface Area	#Sections
axon	7125.84544684	9
soma	1256.63706144	1

Table 2.6 – cell2:Morphology (Regions)

PassiveProp	Priority	Targetter	Value
AxialResistance	0	Default	80.0 ohmcm
SpecificCapacitance	0	Default	1.0 uF/cm2

Table 2.7 – cell2:Passive Properties

Mechanism	Priority	Targetter	Applicator
LkChl	10	Everywhere	Uniform Applicator:
KChl	10	Everywhere	Uniform Applicator:
NaChl	10	Everywhere	Uniform Applicator:
NaChl	20	Region: axon	Uniform Applicator: Multipliers:gScale=1.2

Table 2.8 – cell2:Channels

2.1.3 Neuron:cell3

ID	Tags	Lateral Surface Area (um2)	Region	nseg	L	diam (prox/dist)
0	soma	1257	soma	5	20.0	20.0/20.0
1	axon_1	4864	axon	31	150.0	20.0/0.6
2	axon_2	283	axon	31	150.0	0.6/0.6
3	axon_3	283	axon	31	150.0	0.6/0.6
4	axon_4	283	axon	31	150.0	0.6/0.6
5	axon_5	283	axon	31	150.0	0.6/0.6
6	axon_6	283	axon	31	150.0	0.6/0.6
7	axon_7	283	axon	31	150.0	0.6/0.6
8	axon_8	283	axon	31	150.0	0.6/0.6
9	axon_9	283	axon	31	150.0	0.6/0.6

Table 2.9 – cell3:Morphology (Sections)

Region	Surface Area	#Sections
axon	7125.84544684	9
soma	1256.63706144	1

Table 2.10 – cell3:Morphology (Regions)

PassiveProp	Priority	Targetter	Value
AxialResistance	0	Default	80.0 ohmcm
SpecificCapacitance	0	Default	1.0 uF/cm2

Table 2.11 – cell3:Passive Properties

Mechanism	Priority	Targetter	Applicator
LkChl	10	Everywhere	Uniform Applicator:
KChl	10	Everywhere	Uniform Applicator:
NaChl	10	Everywhere	Uniform Applicator:
NaChl	20	Region: axon	Uniform Applicator: Multipliers:gScale=1.2

Table 2.12 – cell3:Channels

2.2 Channels Details

2.2.1 Summary of KChl (StdChlAlphaBeta)

$$g = gmax * n * n * n * n$$
$$i = g * (erev - V)$$

$$\frac{d}{dt} n = \frac{n_\infty (V) - n}{\tau_n (V)}$$
$$n_\infty (V) = \frac{\alpha_n (V)}{\alpha_n (V) + \beta_n (V)}$$
$$\tau_n (V) = \frac{1.0}{\alpha_n (V) + \beta_n (V)}$$
$$\alpha_n (V), \beta_n (V) = \frac{A + BV}{C + exp\left(\frac{D+V}{E}\right)}$$

Parameter	Value
Conductance (gmax)	36.0 mS/cm2
Reversal Potential	-77.0 mV

n	A	B	C	D	E
Alpha	-0.55	-0.01	-1.0	55.0	-10.0
Beta	0.125	0	0	65	80

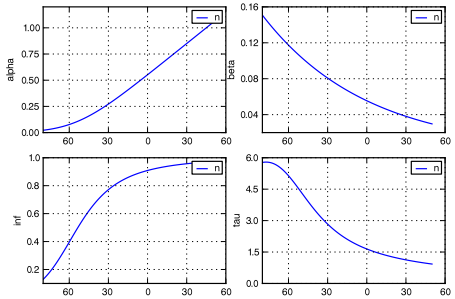


Figure 2.1 – The rate constants and resulting steady-state activation and time constants for n

2.2.2 Summary of channel: LkChl <!! Summariser Missing !!>

<Summariser Missing for type: <class 'morphforgecontrib.simulation.channels.hh_style.neuron.mm_neuron_leak.NEURONChl_Leak'»

2.2.3 Summary of NaChl (StdChlAlphaBeta)

$$g = gmax * m * m * m * h$$
$$i = g * (erev - V)$$

$$\frac{d}{dt}h = \frac{h_\infty(V) - h}{\tau_h(V)}$$
$$h_\infty(V) = \frac{\alpha_h(V)}{\alpha_h(V) + \beta_h(V)}$$
$$\tau_h(V) = \frac{1.0}{\alpha_h(V) + \beta_h(V)}$$
$$\alpha_h(V), \beta_h(V) = \frac{A + BV}{C + exp\left(\frac{D+V}{E}\right)}$$
$$\frac{d}{dt}m = \frac{m_\infty(V) - m}{\tau_m(V)}$$
$$m_\infty(V) = \frac{\alpha_m(V)}{\alpha_m(V) + \beta_m(V)}$$
$$\tau_m(V) = \frac{1.0}{\alpha_m(V) + \beta_m(V)}$$
$$\alpha_m(V), \beta_m(V) = \frac{A + BV}{C + exp\left(\frac{D+V}{E}\right)}$$

Parameter	Value
Conductance (gmax)	120.0 mS/cm2
Reversal Potential	50.0 mV

h	A	B	C	D	E
Alpha	0.07	0.0	0.0	65.0	20.0
Beta	1.0	0.0	1.0	35.0	-10.0

m	A	B	C	D	E
Alpha	-4.0	-0.1	-1.0	40.0	-10.0
Beta	4.0	0.0	0.0	65.0	18.0

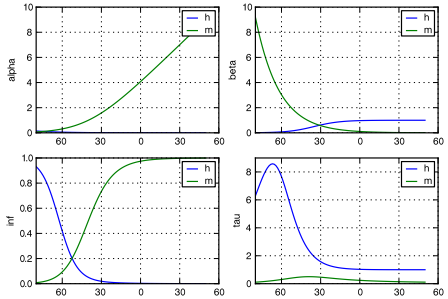


Figure 2.2 – The rate constants and resulting steady-state activation and time constants for h,m

2.3 Synaptic Template Details

2.3.1 MyTemplate (Neurounit Synaptic-Template)

Summary of 'syn_simple'

Assignments

$$\boldsymbol{gmax} = \langle 300.0 \left(10^{-12}\right) \mathrm{m}^{-2} \cdot \mathrm{kg}^{-1} \cdot \mathrm{s}^{3} \cdot \mathrm{A}^{2} \rangle \cdot \boldsymbol{scale}$$
$$\boldsymbol{i} = \boldsymbol{gmax} \cdot (\boldsymbol{v} - \boldsymbol{erev}) \cdot \boldsymbol{g}$$

State Variable Evolution

$$\frac{d}{dt}\boldsymbol{g} = -1.0 \cdot \frac{\boldsymbol{g}}{\boldsymbol{g_\tau}}$$

Symbol	Type	Value	Dimensions	Dependancies	Metadata
$\boldsymbol{scale}$	Param	-	-	-	-
$\boldsymbol{v}$	Supp	-	V	-	u'mf': u'role': u'MEMBRANEVOLTAGE'
$\boldsymbol{erev}$	Const	0.0 (10e-3) m 2 kg 1 s -3 A -1	V	-	-
$\boldsymbol{g_\tau}$	Const	10.0 (10e-3) s 1	s^1	-	-
$\boldsymbol{gmax}$	Assd	-	S	$\{\boldsymbol{scale}\}$	-
$\boldsymbol{i}$	Assd	-	A	$\{\boldsymbol{g}, \boldsymbol{scale}, \boldsymbol{v}\}$	u'mf': u'role': u'TRANSMEMBRANECURRENT'
$\boldsymbol{g}$	State	-	-	$\{\}$	-

Symbols

Events

$$on\_event\ () \rightarrow \left\{\boldsymbol{g} = (\boldsymbol{g} + 1.0)\right.$$

Default Parameters

Parameter	Default Value
scale	1.0

Source

Listing 2.1 – Source code for template: MyTemplate

```
eqnset syn_simple{
g'=-g/g_tau;
i=gmax*(v-erev)*g;
gmax=300pS*scale;
erev=0mV;
g_tau=10ms;
==>>on_event(){
g=g+1.0;
};
<=>INPUT v:mV METADATA{"mf":{"role":"MEMBRANEVOLTAGE"}};
<=>OUTPUT i:(mA)METADATA{"mf":{"role":"TRANSMEMBRANECURRENT"}};
<=>PARAMETER scale:();
};
```

F

SIMULATOR TEST DATA REPOSITORY

Morphforge has been built iteratively according to the needs of the modelling in this thesis. In order to ensure that bugs are not accidentally introduced into the morphforge library, a testsuite was written to allow morphforge to be tested (Section 3.6.5). The Simulator-TestData repository defines a set of *scenarios* in a consistent, human and machine-readable text file.

Each scenario file describes the setup for a simulation. A scenario designed to test the implementation of an alpha-type synapse is shown in Listing F.1. The description is given as a text string [lines: 4-13]. The specification allows parameters to be used in the description (e. g. `<GLEAK>`, `<GSYN>`, `<ESYN>` & `<TCLOSE>`), and defines what should be recorded (e. g. the voltage of *cell1* as `V` and the conductance and currents of the synapse as `SYN_G` & `SYN_I`). The file defines which units to use for all the parameters and recorded values [lines: 20-28], and defines the values that should be used for each parameter in a parameter sweep [lines: 30-34]. The format also describes the columns that should be used for output results in CSV format [line: 37] and the output filename which is defined by the parameters used for that particular simulation [lines: 38]. The repository allows validation of simulation results in two ways. Firstly, results can be compared against a set of hand-calculated results [lines: 43-77], and secondly, results from different simulators can be compared against each other. The repository supports the parametrisation of simulations. To define hand-calculated responses for particular parameters in the text file concisely, *ASCII-table* syntax is used. These tables have two sets of columns: 1 set specifies the particular parameters used [line: 47 `<GLEAK>`, `<GSYN>`, `<ESYN>` & `<TCLOSE>`] another set defines the expected output [line 47: `SYN_G[100.5:290].max`, `SYN_I[100.5:290].min` & `V[100:290].max`]. Values of output

at particular point in time [line 72: `SYN_G[50]`], can be used for comparison, or simple numerical methods such as `mean`, `min` and `max` [line 47] can be used with a *slice-like* notation for specifying periods of time, for example, `SYN_G[100.5:290].max` gives the maximum value of `SYN_G` between 100 to 349 ms. It is possible to set tolerances per column [line: 47] , per table, or per scenario [line: 41]. The repository contains 10 scenario files to test the objects used in this thesis: neurons, channels, synapses & gap junctions. Implementations of the scenarios have been written in NEURON for comparison with morphforge.

```
scenario_short= scenario020
title = Response of a passive cell to an alpha-synapse driven with events

description = """
In a simulation lasting 350ms
Create a single compartment neuron 'cell1' with area 10000um2 and initialvoltage -50mV and capacitance 1e-3 uF/cm2
Add Leak channels to cell1 with conductance <GLEAK> and reversalpotential -50mV
Create a SingleExponential synapse 'SYN' onto cell1 with conductance <GSYN> and reversalpotential <ESYN> and closing-time <TCLOSE>
    driven with spike-times at [100, 300, 300]ms
Record cell1.V as $V
Record SYN.Conductance as $SYN_G
Record SYN.Current as $SYN_I
Run the simulation
"""

[Sampling]
    stop = 350
    dt = 0.1

[Units]
    t = ms
    GLEAK=mS/cm2
    TCLOSE=ms
    ESYN=mV
    GSYN=pS
    V=mV
    SYN_G=pS
    SYN_I=pA

[Parameter Values]
    GSYN=  500, 1000
    ESYN=  0, -20
    GLEAK= 0.03333, 0.014286
    TCLOSE= 5, 20, 100000

[Output Format]
    columns = t, V, SYN_G, SYN_I
    base_filename = scenario020_ESYN<ESYN>_GSYN<GSYN>_GLEAK<GLEAK>_TCLOSE<TCLOSE>_result_

[Check Values]
    eps = 0.05

expectations_i = """
# 1. Use a long decaying syanpse to check that the steady state voltages seem right
# for different values of input resistance, synaptic conductance and reveral potential
| GLEAK    | GSYN    | ESYN | TCLOSE  | SYN_G[100.5:290].max (eps=0.5) | SYN_I[100.5:290].min | V[100:290].max |
|----------|---------|------|---------|--------------------------------|----------------------|----------------|
| 0.03333  | 1000.00 | 0    | 100000  | 1000                           | -38.46               | -38.4615       |
| 0.014286 | 1000.00 | 0    | 100000  | 1000                           | ?                    | -29.4118       |
| 0.03333  |  500.00 | 0    | 100000  | 500                            | ?                    | -43.4769       |
| 0.014286 |  500.00 | 0    | 100000  | 500                            | ?                    | -37.0370       |
| 0.03333  | 1000.00 | -20  | 100000  | 1000                           | ?                    | -43.0769       |
| 0.014286 | 1000.00 | -20  | 100000  | 1000                           | ?                    | -37.6471       |
| 0.03333  |  500.00 | -20  | 100000  | 500                            | ?                    | -46.0869       |
| 0.014286 |  500.00 | -20  | 100000  | 500                            | ?                    | -42.2222       |
"""
expectations_ii = """
# 2. Check that the synapse decays as expected over time:
| GLEAK       | GSYN     | ESYN   | TCLOSE    | SYN_G[102](eps=0.005) |
|-------------|----------|--------|-----------|-----------------------|
| 0.03333     | 1000.00  | 0      | 5         | 670.32                |
| 0.03333     |  500.00  | 0      | 5         | 335.16                |
| 0.03333     | 1000.00  | 0      | 20        | 904.84                |
| 0.03333     |  500.00  | 0      | 20        | 452.42                |
"""

expectations_iii = """
# 3. Check that multiple events work as expected:
| GLEAK       | GSYN     | ESYN   | TCLOSE    | SYN_G[302] (eps=0.5) |
|-------------|----------|--------|-----------|----------------------|
| 0.03333     | 1000.00  | 0      | 5         | 1340.64              |
| 0.03333     |  500.00  | 0      | 5         |  670.32              |
| 0.03333     | 1000.00  | 0      | 20        | 1809.68              |
| 0.03333     |  500.00  | 0      | 20        |  904.84              |
"""
```

Listing F.1 – An example scenario file from the Simulator-TestData repository. The scenario is designed to test an alpha-type synapse.

G

SIMULATION PLATFORM

All simulations were run on a quad-core Intel i5 desktop processor, running Ubuntu Linux, 12.10, kernel `3.2.0-38-generic-pae`. The versions of software used for the simulations are given in Table G.1. Several additional packages were developed through the course of this thesis, details are given in Table G.2.

SOFTWARE	VERSION	URL
Python	2.7.3	http://www.python.org/
numpy	1.6.1	http://www.numpy.org/
scipy	0.9.0	http://www.scipy.org/
matplotlib	1.1.1rc	http://www.matplotlib.org/
quantities	0.10.1	http://pythonhosted.org/quantities/
NEURON	7.1 (359:7f113b76a94b)	http://www.neuron.yale.edu/neuron/

Table G.1 – The versions of software used in for simulations

SOFTWARE	URLS
morphforge	DOC: https://morphforge.rtfd.org SRC: https://github.com/mikehulluk/morphforge
neurounit	DOC: https://neurounit.rtfd.org SRC: https://github.com/mikehulluk/neurounit
mreorg	DOC: https://mredoc.rtfd.org SRC: https://github.com/mikehulluk/mreorg
mredoc	DOC: https://mredoc.rtfd.org SRC: https://github.com/mikehulluk/mredoc
TestData Repository	SRC: https://github.com/mikehulluk/simulator-test-data

Table G.2 – Packages written during this thesis.

BIBLIOGRAPHY

Aiken, S. P., Kuenzi, F. M., and Dale, N. (2003). *Xenopus* embryonic spinal neurons recorded *in situ* with patch-clamp electrodes - conditional oscillators after all? *Eur J Neurosci*, 18(2):333–343. (Cited on pages 129, 145, and 186.)

Alexandrescu, A., Meyers, S., and Vlissides, J. (2001). *Modern C++ Design: Applied Generic and Design Patterns (C++ in Depth)*. Addison Wesley. (Cited on page 77.)

Angstadt, J. D., Grassmann, J. L., Theriault, K. M., and Levasseur, S. M. (2005). Mechanisms of postinhibitory rebound and its modulation by serotonin in excitatory swim motor neurons of the medicinal leech. *J Comp Physiol*, 191(8):715–32. (Cited on pages 9 and 128.)

Arbas, E. and Calabrese, R. L. (1987). Slow oscillations of membrane potential in interneurons that control heartbeat in the medicinal leech. *J Neurosci*, 12(7):3953–60. (Cited on page 128.)

Arshavsky, Y. I. (2003). Cellular and network properties in the functioning of the nervous system: from central pattern generators to cognition. *Brain Res Rev*, 41(2-3):229–267. (Cited on page 128.)

Ayers, J., Carpenter, G., Currie, S., and Kinch, J. (1983). Which behavior does the lamprey central motor program mediate? *SCIENCE*, 221:1313–4. (Cited on page 4.)

Barry, M. J. and O'Donovan, M. J. (1987). The effects of excitatory amino acids and their antagonists on the generation of motor activity in the isolated chick spinal cord. *Brain Res*, 433(2):271–6. (Cited on page 10.)

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional. (Cited on pages 62 and 65.)

Belousov, A. B. (2011). The regulation and role of neuronal gap junctions during development. *Communicative & Integrative Biology*, 4(5):579–81. (Cited on page 97.)

Bennett, M. V., Pappas, G. D., Giménez, M., and Nakajima, Y. (1967). Physiology and ultrastructure of electrotonic junctions. IV. Medullary electromotor nuclei in gymnotid fish. *J Physiol*, 30(2):236–300. (Cited on page 96.)

Bennett, M. V. L. (1966). Physiology of electrotonic junctions. *Ann NY Acad Sci*, 137(2):509–539. (Cited on pages 30 and 58.)

Bennett, M. V. L. and Zukin, R. S. (2004). Neuronal synchronization in the mammalian brain. *Neuron*, 41(4):495–511. (Cited on pages 19, 96, and 118.)

Berkowitz, A., Roberts, A., and Soffe, S. R. (2010). Roles for multifunctional and specialized spinal interneurons during motor pattern generation in tadpoles, zebrafish larvae, and turtles. *Front Behav Neurosci*, 4(36):36. (Cited on page 4.)

Berry, M. and Pentreath, V. (1976). Criteria for distinguishing between monosynaptic and polysynaptic transmission. *Brain Research*, 105(1):1 – 20. (Cited on pages 158 and 186.)

Bertrand, S. (1998). Postinhibitory rebound during locomotor-like activity in neonatal rat motoneurons *in vitro*. *J Neurophysiol*, pages 342–351. (Cited on pages 10 and 128.)

Boiko, T., Van Wart, A., Caldwell, J. H., Levinson, S. R., Trimmer, J. S., and Matthews, G. (2003). Functional specialization of the axon initial segment by isoform-specific sodium channel targeting. *J Neurosci*, 23(6):2306–13. (Cited on pages 119 and 120.)

Boothby, K. M. and Roberts, A. (1995). Effects of site of tactile stimulation on the escape swimming responses of hatchling *Xenopus laevis* embryos. *J Zool*, 235(1):113–125. (Cited on pages 152 and 174.)

Borisyuk, R., Al Azad, A. K., Conte, D., Roberts, A., and Soffe, S. R. (2011). Modeling the connectome of a simple spinal cord. *Front Neuroinform*, 5(September):20. (Cited on page 21.)

Borisyuk, R., Cooke, T., and Roberts, A. (2008). Stochasticity and functionality of neural systems: mathematical modelling of axon growth in the spinal cord of tadpole. *Bio Systems*, 93(1-2):101–14. (Cited on pages 21 and 220.)

Bostock, H., Sherratt, R., and Sears, T. (1978). Overcoming conduction failure in demyelinated nerve fibres by prolonging action potentials. *Nature*, 274(July):385–387. (Cited on page 120.)

Bower, J. M. and Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Springer. (Cited on page 67.)

Boyan, G. S., Ashman, S., and Ball, E. E. (1986). Initiation and modulation of flight by a single giant interneuron in the cercal system of the locust. *Naturwissenschaften*, 73(5):272–274. (Cited on page 5.)

Boyle, J. H. and Cohen, N. (2008). *Caenorhabditis elegans* body wall muscles are simple actuators. *Bio Systems*, 94(1-2):170–81. (Cited on page 11.)

Boyle, M. B., Cohen, L. B., Macagno, E. R., and Orbach, H. (1983). The number and size of neurons in the CNS of gastropod molluscs and their suitability for optical recording of activity. *Brain Res*, 266(2):305–317. (Cited on page 8.)

Bresadola, M. (1998). Medicine and science in the life of Luigi Galvani (1737-1798). *Brain Res Bull*, 46(5):367–80. (Cited on page 29.)

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., El Boustani, S., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J Comp Neuro*, 23(3):349–98. (Cited on page 67.)

Brooks, F. P. (1995). *The Mythical Man Month and Other Essays on Software Engineering*. Addison Wesley. (Cited on page 63.)

Brown, T. (1914). On the nature of the fundamental activity of the nervous centres; together with an analysis of the conditioning of rhythmic activity in progression, and a theory of the evolution of function in the nervous system. *J Physiol*, 48(1):18. (Cited on pages 4, 6, 9, and 126.)

Brown, T. G. (1911). The intrinsic factors in the act of progression in the mammal. *P Roy Soc B-Biol Sci*, 84(572):308–319. (Cited on pages 4, 6, and 9.)

Bruederle, D., Davison, A., Kremkow, J., Mueller, E., Muller, E., Nawrot, M., Pereira, M., Perrinet, L., Schmuker, M., and Yger, P. (2010). NeuroTools: Analysis, visualization and management of real and simulated neuroscience data. (Cited on page 66.)

Bryden, J. and Cohen, N. (2004). A simulation model of the locomotion controllers for the nematode *Caenorhabditis elegans*. In Schaal, S., J, I. A., S, B. A. V., J, H., and Meyer, J.-A., editors, *From Animals to Animats 8: Proceedings of the Eighth International Conference on the Simulation of Adaptive Behavior*, number July, pages 183–192. MIT Press. (Cited on pages 9 and 11.)

Buchanan, J. T. (2001). Contributions of identifiable neurons and neuron classes to lamprey vertebrate neurobiology. *Prog Neurobiol*, 63(4):441–66. (Cited on page 7.)

Buhl, E., Roberts, A., and Soffe, S. R. (2012). The role of a trigeminal sensory nucleus in the initiation of locomotion. *J Physiol*, 590(Pt 10):2453–2469. (Cited on pages 16, 17, 22, 25, 53, 56, 128, 132, 134, 152, 153, 156, 157, 158, 174, and 185.)

Calabrese, R. and Feldman, J. (1997). Intrinsic membrane properties and synaptic mechanisms in motor rhythm generators. In Stein, P. S. G., Grillner, S., Selverston, A. I., and Stuart, D. G., editors, *Neurons, Networks and Motor Behaviour*, pages 119–130. A Bradford Book. (Cited on page 128.)

Calabrese, R. L. (1998). Cellular, synaptic, network, and modulatory mechanisms involved in rhythm generation. *Curr Opin Neurobiol*, 8(6):710–7. (Cited on page 10.)

Calì, C., Berger, T., Pignatelli, M., Carleton, A., Markram, H., and Giugliano, M. (2008). Inferring connection proximity in networks of electrically coupled cells by subthreshold frequency response analysis. *J Comp Neuro*, 24:330–345. (Cited on page 97.)

Calin-Jageman, R. J., Tunstall, M. J., Mensh, B. D., Katz, P. S., and Frost, W. N. (2007). Parameter space analysis suggests multi-site plasticity contributes to motor pattern initiation in *Tritonia*. *J Neurophysiol*, 98(4):2382–98. (Cited on pages 11 and 126.)

Cannon, R., Gewaltig, M., and Gleeson, P. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics*, 5(2):127–138. (Cited on page 68.)

Carnevale, N. T. and Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press, Cambridge. (Cited on pages 35, 67, 68, 84, 195, 219, and 225.)

Catterall, W. (1981). Localization of sodium channels in cultured neural cells. *J Neurosci*, 1(7):777–783. (Cited on pages 119 and 120.)

Chalfie, M., Sulston, J., White, J., Southgate, E., Thomson, J., and Brenner, S. (1985). The neural circuit for touch sensitivity in *Caenorhabditis elegans*. *J Neurosci*, 5(4):956–964. (Cited on page 8.)

Clarac, F. and Pearlstein, E. (2007). Invertebrate preparations and their contribution to neurobiology in the second half of the 20th century. *Brain Res Rev*, 54(1):113–161. (Cited on page 4.)

Clarke, J., Hayes, B., Hunt, S., and Roberts, A. (1984). Sensory physiology, anatomy and immunohistochemistry of Rohon-Beard neurones in embryos of *Xenopus laevis*. *J Physiol.*, 384(March):511–525. (Cited on pages 16 and 109.)

Coghill, G. E. (1929). *Anatomy and the problem of behavior*. The University Press. (Cited on page 12.)

Connor, J. and Stevens, C. (1971). Prediction of repetitive firing behaviour from voltage clamp data on an isolated neurone soma. *J Physiol*, 213:31–53. (Cited on pages 34, 43, 46, 144, and 188.)

Connors, B. W. and Long, M. A. (2004). Electrical synapses in the mammalian brain. *Ann Rev Neuro*, 27:393–418. (Cited on pages 19 and 97.)

Cornelis, H., Rodriguez, A. L., Coop, A. D., and Bower, J. M. (2012). Python as a federation tool for GENESIS 3.0. *PLoS One*, 7(1):e29018. (Cited on page 68.)

Crawley, M. J. (2007). *The R Book*. Wiley-Blackwell. (Cited on page 159.)

Crook, S., Gleeson, P., Howell, F., Svitak, J., and Silver, R. A. (2007). MorphML: level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics*, 5(2):96–104. (Cited on pages 71 and 84.)

Crook, S. M., Bednar, J. A., Berger, S., Cannon, R., Davison, A. P., Djurfeldt, M., Eppler, J., Kriener, B., Furber, S., Graham, B., Plesser, H. E., Schwabe, L., Smith, L., Steuber, V., and van Albada, S. (2012). Creating, documenting and sharing network models. *Network*, 23(4):1–19. (Cited on pages 189 and 235.)

Currie, S. N. (1999). Fictive hindlimb motor patterns evoked by AMPA and NMDA in turtle spinal cord-hindlimb nerve preparations. *J Physiology-Paris*, 93(3):199–211. (Cited on page 10.)

Dale, D. (2011). Python quantities. http://pythonhosted.org/quantities/. Accessed: 2013-06-27. (Cited on page 77.)

Dale, N. (1985). Reciprocal inhibitory interneurones in the *Xenopus* embryo spinal cord. *J Physiol*, 363:61–70. (Cited on page 53.)

Dale, N. (1993). A large, sustained Na (+)-and voltage-dependent K+ current in spinal neurons of the frog embryo. *J Physiol*, 462(1):349. (Cited on page 36.)

Dale, N. (1995a). Experimentally derived model for the locomotor pattern generator in the *Xenopus* embryo. *J Physiol*, 489(1995):489–510. (Cited on pages 12, 19, 23, 37, 47, 49, 50, 122, 128, 129, 142, 143, and 196.)

Dale, N. (1995b). Kinetic characterization of the voltage-gated currents possessed by *Xenopus* embryo spinal neurons. *J Physiol*, 489(2):473–88. (Cited on pages 23, 36, 37, 47, 49, 50, 51, 59, 107, and 121.)

Dale, N. (2003). Coordinated motor activity in simulated spinal networks emerges from simple biologically plausible rules of connectivity. *J Comp Neuro*, 14(1):55–70. (Cited on pages 122 and 128.)

Dale, N., Ottersen, O., and Roberts, A. (1986). Inhibitory neurones of a motor pattern generator in *Xenopus* revealed by antibodies to glycine. *Nature*. (Cited on page 53.)

Dale, N. and Roberts, A. (1985). Dual-component amino-acid-mediated synaptic potentials: excitatory drive for swimming in *Xenopus* embryos. *J Physiol*, 363(1):35. (Cited on pages 10, 18, 53, and 133.)

Davison, A. (2012). Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56. (Cited on pages 65, 72, 73, and 235.)

Davison, A. P., Hines, M. L., and Muller, E. (2009). Trends in programming languages for neuroscience simulations. *Front Neurosci*, 3(3):374–80. (Cited on page 68.)

de Rijk, M. C., Tzourio, C., Breteler, M. M., Dartigues, J. F., Amaducci, L., Lopez-Pousa, S., Manubens-Bertran, J. M., Alpérovitch, A., and Rocca, W. A. (1997). Prevalence of parkinsonism and Parkinson's disease in Europe: the EUROPARKINSON collaborative study. *J Neurol Neurosur Psychiatry*, 62(1):10–5. (Cited on page 150.)

Delcomyn, F. (1980). Neural basis of rhythmic behavior in animals. *Science*, 210(4469):492. (Cited on page 4.)

Delcomyn, F. (1998). *Foundations of Neurobiology*. W. H. Freeman. (Cited on pages 30 and 53.)

des Poids et Mesures, B. I. (2006). *International System of Units (SI)*. Bureau International des Poids et Mesures. (Cited on page 208.)

Djurfeldt, M. (2012). The connection-set algebra–a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics*, 10(3):287–304. (Cited on page 70.)

Domenici, P., Blagburn, J. M., and Bacon, J. P. (2011). Animal escapology I: theoretical issues and emerging trends in escape trajectories. *J Exp Biol*, 214(Pt 15):2463–73. (Cited on page 152.)

Draguhn, A., Traub, R. D., Schmitz, D., and Jefferys, J. G. (1998). Electrical coupling underlies high-frequency oscillations in the hippocampus *in vitro*. *Nature*, 394(6689):189–92. (Cited on page 97.)

Dubuc, R., Brocard, F., Antri, M., Fénelon, K., Gariépy, J.-F., Smetana, R., Ménard, A., Le Ray, D., Viana Di Prisco, G., Pearlstein, E., Sirota, M. G., Derjean, D., St-Pierre, M., Zielinski, B., Auclair, F., and Veilleux, D. (2008). Initiation of locomotion in lampreys. *Brain Res Rev*, 57(1):172–82. (Cited on page 150.)

Edwards, D., Yeh, S.-R., and Krasne, F. B. (1998). Neuronal coincidence detection by voltage-sensitive electrical synapses. *Proc Natl Acad Sci*, 95(June):7145–7150. (Cited on page 96.)

Edwards, D. H., Heitler, W. J., and Krasne, F. B. (1999). Fifty years of a command neuron: the neurobiology of escape behavior in the crayfish. *Trends Neurosci*, 22(4):153–61. (Cited on page 154.)

Eisen, J. and Marder, E. (1982). Mechanisms underlying pattern generation in lobster stomatogastric ganglion as determined by selective inactivation of identified neurons. III. Synaptic connections of electrically coupled pyloric neurons. *J Neurophysiol*, 48:1392–1415. (Cited on page 96.)

Ekeberg, O. (1994). An integrated neuronal and mechanical model of fish swimming. In Eeckman, F. H., editor, *Computation in Neurons and Neural Systems*, pages 217–222. Springer, Washington, DC. (Cited on page 11.)

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: A convenient interface to the NEST simulator. *Front Neuroinform*, 2(January):12. (Cited on page 68.)

Faber, J. and Nieuwkoop, P. D. (1956). *Normal table of Xenopus laevis (Daudin)*. Amsterdam, North-Holland. (Cited on page 13.)

Faisal, A. A. and Laughlin, S. B. (2007). Stochastic simulations on the reliability of action potential propagation in thin axons. *PLoS Comput Biol*, 3(5):783–795. (Cited on pages 105 and 119.)

Faisal, A. A., White, J. A., and Laughlin, S. B. (2005). Ion-channel noise places limits on the miniaturization of the brain's wiring. *Curr Biol*, 15(12):1143–9. (Cited on page 119.)

Feldman, J. L., Mitchell, G. S., and Nattie, E. E. (2003). Breathing: rhythmicity, plasticity, chemosensitivity. *Ann Rev Neuro*, 26:239–66. (Cited on page 126.)

Fetcho, J. R., Higashijima, S.-i., and McLean, D. L. (2008). Zebrafish and motor control over the last decade. *Brain Res Rev*, 57(1):86–93. (Cited on page 7.)

Fetcho, J. R. and Liu, K. S. (1998). Zebrafish as a model system for studying neuronal circuits and behavior. *Ann NY Acad Sci*, 860(1):333–345. (Cited on page 7.)

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code (Object Technology Series)*. Addison Wesley. (Cited on page 65.)

Friesen, W. and Kristan, W. (2007). Leech locomotion: swimming, crawling, and decisions. *Curr Opin Neurobiol*, 17(6):704–711. (Cited on page 8.)

Frost, W., Hoppe, T., Wang, J., and Tian, L. (2001). Swim initiation neurons in *Tritonia diomedea*. *Am Zool*, 41(4):952. (Cited on page 8.)

Fukuda, T. and Kosaka, T. (2000). Gap junctions linking the dendritic network of GABAergic interneurons in the hippocampus. *J Neurosci*, 20(4):1519–28. (Cited on page 97.)

Furshpan, E. J. and Potter, D. D. (1959). Transmission at the giant motor synapses of the crayfish. *J Physiol.*, 145:289–325. (Cited on pages 19, 96, and 117.)

Gabriel, J. P., Ausborn, J., Ampatzis, K., Mahmood, R., Eklöf-Ljunggren, E., and El Manira, A. (2010). Principles governing recruitment of motoneurons during swimming in zebrafish. *Nat Neuro*, 14(1):93–99. (Cited on page 10.)

Galarreta, M. and Hestrin, S. (2002). Electrical and chemical synapses among parvalbumin fast-spiking GABAergic interneurons in adult mouse neocortex. *P Natl Acad Sci USA*, 99(19):12438. (Cited on page 97.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns : elements of reusable object-oriented software*. Addison Wesley. (Cited on pages 70, 216, and 222.)

Getting, P. a. (1983). Mechanisms of pattern generation underlying swimming in *Tritonia*. II. Network reconstruction. *J Neurophysiol*, 49(4):1017–35. (Cited on pages 11 and 126.)

Gewaltig, M.-O. and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia*, 2(4):1430. (Cited on page 67.)

Gibson, J. R., Beierlein, M., and Connors, B. W. (1999). Two networks of electrically coupled inhibitory neurons in neocortex. *Nature*, 402(6757):75–9. (Cited on page 97.)

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., and Myers, J. (2007). Examining the challenges of scientific workflows. *Computer*, 40(12):24–32. (Cited on page 64.)

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., Morse, T. M., Davison, A. P., Ray, S., Bhalla, U. S., Barnes, S. R., Dimitrova, Y. D., and Silver, R. A. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PloS Comput Biol*, 6(6):e1000815. (Cited on page 69.)

Goddard, N. H., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philos T R Soc B*, 356(1412):1209–28. (Cited on page 69.)

Gold, C., Henze, D. a., Koch, C., and Buzsáki, G. (2006). On the origin of the extracellular action potential waveform: A modeling study. *J Neurophysiol*, 95(5):3113–28. (Cited on page 190.)

Goodman, D. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front Neuroinform*, 2(November):5. (Cited on pages 67 and 77.)

Goulding, M. (2009). Circuits controlling vertebrate locomotion: moving in a new direction. *Nat Rev Neuro*, 10(7):507–518. (Cited on page 126.)

Grillner, S. (1975). Locomotion in vertebrates: central mechanisms and reflex interaction. *Physiol Rev*, 55(2):247–304. (Cited on page 4.)

Grillner, S. (1999). Bridging the gap–from ion channels to networks and behaviour. *Curr Opin Neurobiol*, pages 663–669. (Cited on page 128.)

Grillner, S., McClellan, A., Sigvardt, K., Wallén, P., and Wilén, M. (1981). Activation of NMDA-receptors elicits "fictive locomotion" in lamprey spinal cord *in vitro*. *Acta Physiol Scand*, 113(4):549–51. (Cited on page 10.)

Grillner, S. and Wallen, P. (1985). Central pattern generators for locomotion, with special reference to vertebrates. *Ann Rev Neuro*, 8(1):233–261. (Cited on page 4.)

Gronenschild, E. H. B. M., Habets, P., Jacobs, H. I. L., Mengelers, R., Rozendaal, N., van Os, J., and Marcelis, M. (2012). The effects of FreeSurfer version, workstation type, and Macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS One*, 7(6):e38234. (Cited on page 63.)

Groth, P. T. and Gil, Y. (2008). A scientific workflow construction command line. In *Proceedings of the 13th International Conference on Intelligent User Interfaces - IUI '09*, page 445, New York, New York, USA. ACM Press. (Cited on page 65.)

Grubb, M. S. and Burrone, J. (2010). Building and maintaining the axon initial segment. *Curr Opin Neurobiol*, 20(4):481–488. (Cited on page 108.)

Grundfest, H. (1968). Invertebrate nervous systems: their significance for mammalian neurophysiology. *Arch Neurol-Chicago*, 18(2):222–222. (Cited on page 9.)

Grune, D., Bal, H., Jacobs, C., and Langendoen, K. (2000). *Modern Compiler Design*. Wiley. (Cited on page 200.)

Halliday, D., Resnick, R., and Walker, J. (2004). *Fundamentals of Physics*. John Wiley & Sons. (Cited on page 33.)

Hille, B. (2001). *Ion channels of excitable membranes*, volume 3rd. Sinauer Associates. (Cited on pages 9, 23, 29, 32, 34, 35, 43, 46, 50, 107, 109, 144, 186, and 188.)

Hillis, W. D. (1993). Why physicists like models and why biologists should. *Curr Biol*, 3(2):79–81. (Cited on pages 181 and 182.)

Hindmarsh, A., Brown, P., and Grant, K. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM T Math Software*, 31(3):363–396. (Cited on pages 67 and 232.)

Hines, M. (1984). Efficient computation of branched nerve equations. *Int J Biomed Comput*, 15(1):69–76. (Cited on page 67.)

Hines, M. L. and Carnevale, N. T. (2000). Expanding NEURON's Repertoire of Mechanisms with NMODL. *Neural Comput*, 12(5):995–1007. (Cited on page 68.)

Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front Neuroinform*, 3:1. (Cited on page 68.)

Hnasko, T. S. and Edwards, R. H. (2012). Neurotransmitter corelease: mechanism and physiological role. *Annu Rev Physiol*, 74:225–43. (Cited on page 53.)

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun ACM*, 4(7):321. (Cited on page 224.)

Hodgkin, A. and Huxley, A. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117:500–544. (Cited on pages 29, 30, 32, 59, and 139.)

Hoffman, N. and Parker, D. (2010). Lesioning alters functional properties in isolated spinal cord hemisegmental networks. *Neuroscience*, 168(3):732–43. (Cited on pages 10, 145, and 187.)

Hoge, G. J., Davidson, K. G. V., Yasumura, T., Castillo, P. E., Rash, J. E., and Pereda, A. E. (2011). The extent and strength of electrical coupling between inferior olivary neurons is heterogeneous. *J Neurophysiol*, 105(3):1089–101. (Cited on page 97.)

Hormuzdi, S. G., Filippov, M. a., Mitropoulou, G., Monyer, H., and Bruzzone, R. (2004). Electrical synapses: a dynamic signaling system that shapes the activity of neuronal networks. *Biochim Biophys Acta*, 1662(1-2):113–37. (Cited on page 97.)

Hu, H., Ma, Y., and Agmon, A. (2011). Precise Synchrony by Mutual Inhibition: Experiments Vindicate Theory. *Molecular and Cellular Pharmacology*, 3(2):59–65. (Cited on page 176.)

Hudson, R., Norris, J., and Reid, L. (2011). Data-intensive management and analysis for scientific simulations. In *Ninth Australasian Symposium on Parallel and Distributed Computing*, volume 118. (Cited on page 64.)

Humphries, D. A. and Driver, P. M. (1970). Protean defence by prey animals. *Oecologia*, 5(4):285–302. (Cited on page 152.)

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer*. Addison Wesley. (Cited on page 65.)

Ijspeert, A. (2001). A connectionist central pattern generator for the aquatic and terrestrial gaits of a simulated salamander. *Biol Cybern*, 84(5):331–48. (Cited on page 126.)

Izhikevich, E. (2007). *Dynamical systems in neuroscience: the geometry of excitability and bursting*. The MIT press. (Cited on page 35.)

Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE T Neural Networ*, 14(6):1569–72. (Cited on page 35.)

Jamieson, D. and Roberts, a. (2000). Responses of young *Xenopus laevis* tadpoles to light dimming: possible roles for the pineal eye. *J Exp Biol*, 203(Pt 12):1857–67. (Cited on page 13.)

Jordan, L. M. (1998). Initiation of locomotion in mammals. *Ann NY Acad Sci*, 860:83–93. (Cited on pages 6 and 150.)

Jordan, L. M., Liu, J., Hedlund, P. B., Akay, T., and Pearson, K. G. (2008). Descending command systems for the initiation of locomotion in mammals. *Brain Res Rev*, 57(1):183–91. (Cited on page 150.)

Juszczak, G. R. and Swiergiel, A. H. (2009). Properties of gap junction blockers and their behavioural, cognitive and electrophysiological effects: Animal and human studies. *Progress in Neuro-Psychopharmacology and Biological Psychiatry*, 33(2):181 – 198. (Cited on page 119.)

Kahn, J. and Roberts, A. (1982a). The central nervous origin of the swimming motor pattern in embryos of *Xenopus laevis*. *J. Exp. Biol.*, 99(1):185–196. (Cited on page 176.)

Kahn, J. and Roberts, A. (1982b). The neuromuscular basis of rhythmic struggling movements in embryos of *Xenopus laevis*. *J Exp Biol*, 99(1):197. (Cited on page 13.)

Kahn, J., Roberts, A., and Kashin, S. (1982). The neuromuscular basis of swimming movements in embryos of the amphibian *Xenopus laevis*. *J Exp Biol*, 99(1):175. (Cited on pages 13, 14, 15, and 127.)

Karlsson, B. (2005). *Beyond the C++ Standard Library: An Introduction to Boost*. Addison Wesley. (Cited on page 77.)

Kernighan, B. W. and Ritchie, D. (1988). *The C Programming Language (2nd Edition)*. Prentice Hall. (Cited on pages 65, 70, 208, and 237.)

Kiehn, O. and Tresch, M. C. (2002). Gap junctions and motor behavior. *Trends Neurosci*, 25(2):108–15. (Cited on pages 117 and 154.)

Knudsen, D., Arsenault, J., Hill, S., Omalley, D., and Jose, J. (2006). Locomotor network modeling based on identified zebrafish neurons. *Neurocomputing*, 69(10-12):1169–1174. (Cited on page 12.)

Knuth, D. E. (1992). *Literate Programming (Center for the Study of Language and Information Publication Lecture Notes)*. Cambridge University Press. (Cited on page 66.)

Koch, C. (1999). *Biophysics of Computation: Information Processing in Single Neurons*. Oxford University Press, New York. (Cited on pages 11, 32, 35, 50, 56, 59, 100, 136, 195, 197, and 209.)

Kole, M. H. P., Ilschner, S. U., Kampa, B. M., Williams, S. R., Ruben, P. C., and Stuart, G. J. (2008). Action potential generation requires a high sodium channel density in the axon initial segment. *Nat Neurosci*, 11(2):178–86. (Cited on page 108.)

Korn, H. and Faber, D. S. (2005). The Mauthner cell half a century later: a neurobiological model for decision-making? *Neuron*, 47(1):13–28. (Cited on page 5.)

Krasne, F. B. (1969). Excitation and habituation of the crayfish escape reflex: the depolarizing response in lateral giant fibres of the isolated abdomen. *J Exp Biol*, 50(1):29–46. (Cited on page 5.)

Kress, G. J., Dowling, M. J., Meeks, J. P., and Mennerick, S. (2008). High threshold, proximal initiation, and slow conduction velocity of action potentials in dentate granule neuron mossy fibers. *J Neurophysiol*, 100(1):281–91. (Cited on pages 108, 119, and 120.)

Kristan, W. B., Calabrese, R. L., and Friesen, W. O. (2005). Neuronal control of leech behavior. *Prog Neurobiol*, 76(5):279–327. (Cited on pages 8 and 126.)

Krug, E. and Kresnow, M. (1999). Retraction: Suicide after natural disasters. *New Engl J Med*, 340(2):148–149. (Cited on page 63.)

Kyriakatos, A., Mahmood, R., Ausborn, J., Porres, C. P., Büschges, A., and El Manira, A. (2011). Initiation of locomotion in adult zebrafish. *J Neurosci*, 31(23):8422–31. (Cited on page 7.)

Lamb, D. G. and Calabrese, R. L. (2011). Neural circuits controlling behavior and autonomic functions in medicinal leeches. *Neural Systems & Circuits*, 1(1):13. (Cited on page 8.)

Lambert, T. D. (2004). Mechanisms and significance of reduced activity and responsiveness in resting frog tadpoles. *J Exp Biol*, 207(7):1113–1125. (Cited on pages 13, 21, and 171.)

Langer, J., Stephan, J., Theis, M., and Rose, C. (2012). Gap junctions mediate intercellular spread of sodium between hippocampal astrocytes *in situ*. *Glia*, 252(March 2011):239–252. (Cited on page 117.)

Leifer, A. M., Fang-Yen, C., Gershow, M., Alkema, M. J., and Samuel, A. D. T. (2011). Optogenetic manipulation of neural activity in freely moving *Caenorhabditis elegans*. *Nat Methods*, 8(2):147–52. (Cited on page 9.)

Leisch, F. (2002). Sweave: dynamic generation of statistical reports using literate data analysis. In *Compstat 2002 Proceedings in Computational Statistics*, volume CompStat 2, pages 575–580. Physica Verlag, Heidelberg. (Cited on page 66.)

Li, W. and Moult, P. (2012). The control of locomotor frequency by excitation and inhibition. *J Neurosci*. (Cited on page 188.)

Li, W.-C., Cooke, T., Sautois, B., Soffe, S., Borisyuk, R., and Roberts, A. (2007a). Axon and dendrite geography predict the specificity of synaptic connections in a functioning spinal cord network. *Neural Dev*, 2:17. (Cited on pages 20, 21, and 163.)

Li, W.-C., Perrins, R., Soffe, S., Yoshida, M., Walford, A., and Roberts, A. (2001). Defining classes of spinal interneuron and their axonal projections in hatchling *Xenopus laevis* tadpoles. *J Comp Neurol*, 441:248–265. (Cited on pages 100 and 154.)

Li, W.-C., Roberts, A., and Soffe, S. R. (2009). Locomotor rhythm maintenance: electrical coupling among premotor excitatory interneurons in the brainstem and

spinal cord of young *Xenopus* tadpoles. *J Physiol*, 587(Pt 8):1677–93. (Cited on pages 19, 23, 37, 57, 97, 98, 100, 101, 118, 119, 154, 172, 177, 185, 187, and 188.)

Li, W.-C., Roberts, A., and Soffe, S. R. (2010). Specific brainstem neurons switch each other into pacemaker mode to drive movement by activating NMDA receptors. *J Neurosci*, 30(49):16609–20. (Cited on pages 18, 19, 98, 107, 121, 122, 128, 130, 132, 133, 134, 139, 142, 144, 145, 147, 151, 172, 173, 186, and 188.)

Li, W.-C., Sautois, B., Roberts, A., and Soffe, S. R. (2007b). Reconfiguration of a vertebrate motor network: specific neuron recruitment and context-dependent synaptic plasticity. *J Neurosci*, 27(45):12267–76. (Cited on pages 16, 19, 20, 53, and 145.)

Li, W.-C., Soffe, S. R., and Roberts, A. (2003). The spinal interneurons and properties of glutamatergic synapses in a primitive vertebrate cutaneous flexion reflex. *J Neurosci*, 23(27):9068–77. (Cited on pages 16, 121, and 156.)

Li, W.-C., Soffe, S. R., and Roberts, A. (2004a). A direct comparison of whole cell patch and sharp electrodes by simultaneous recording from single spinal neurons in frog tadpoles. *Journal of Neurophysiology*, 92(1):380–386. (Cited on pages 122 and 154.)

Li, W.-C., Soffe, S. R., and Roberts, A. (2004b). Dorsal spinal interneurons forming a primitive, cutaneous sensory pathway. *J Neurophysiol*, 92(2):895–904. (Cited on pages 15, 16, 53, 146, and 176.)

Li, W.-C., Soffe, S. R., Wolf, E., and Roberts, A. (2006). Persistent responses to brief stimuli: feedback excitation among brainstem neurons. *J Neurosci*, 26(15):4026–35. (Cited on pages 17, 18, 98, 100, 107, 109, 120, 121, 127, 128, 129, 132, 133, 142, 146, 152, 153, and 186.)

Lin, J. W. and Faber, D. S. (1988). Synaptic transmission mediated by single club endings on the goldfish Mauthner cell. I. Characteristics of electrotonic and chemical postsynaptic potentials. *J Neurosci*, 8(4):1302–12. (Cited on pages 19 and 96.)

Liu, M.-G., Chen, X.-F., He, T., Li, Z., and Chen, J. (2012). Use of multi-electrode array recordings in studies of network synaptic plasticity in both time and space. *Neuroscience bulletin*, 28(4):409–22. (Cited on page 190.)

Llinas, R., Baker, R., and Sotelo, C. (1974). Electrotonic coupling between neurons in cat inferior olive. *J Neurophysiol*. (Cited on page 97.)

Lockery, S., Goodman, M., and Faumont, S. (2009). First report of action potentials in a C. elegans neuron is premature. *Nat Neuro*, 12(4):365–366. (Cited on page 9.)

López-Muñoz, F. and Alamo, C. (2009). Historical evolution of the neurotransmission concept. *J Neural Transm (Vienna, Austria : 1996)*, 116(5):515–33. (Cited on pages 29 and 30.)

Luthi, A. (1998). H-current: properties of a neuronal and network pacemaker. *Neuron*, 21:9–12. (Cited on page 34.)

Maex, R. and De Schutter, E. (2007). Mechanism of spontaneous and self-sustained oscillations in networks connected through axo-axonal gap junctions. *Eur J Neurosci*, 25(11):3347–58. (Cited on page 97.)

Mann-Metzer, P. and Yarom, Y. (1999). Electrotonic coupling interacts with intrinsic properties to generate synchronized activity in cerebellar networks of inhibitory interneurons. *J Neurosci*, 19(9):3298–306. (Cited on page 97.)

Marder, E. (1998). Electrical synapses: beyond speed and synchrony to computation. *Curr Biol*, 8(22):R795–7. (Cited on pages 19 and 117.)

Marder, E. (2000). Motor pattern generation. *Curr Opin Neurobiol*, 10(6):691–8. (Cited on page 4.)

Marder, E. and Bucher, D. (2007). Understanding circuit dynamics using the stomatogastric nervous system of lobsters and crabs. *Annu. Rev. Physiol.*, 69:291–316. (Cited on pages 18, 96, and 126.)

Marder, E. and Calabrese, R. L. (1996). Principles of rhythmic motor pattern generation. *Physiol Rev*, 76(3):687–717. (Cited on page 128.)

Marder, E., Tobin, A.-E., and Grashow, R. (2007). How tightly tuned are network parameters? Insight from computational and experimental studies in small rhythmic motor networks. *Prog Brain Res*, 165:193–200. (Cited on pages 12 and 182.)

Matsushima, T., Tegnér, J., Hill, R. H., and Grillner, S. (1993). GABA-B receptor activation causes a depression of low- and high-voltage activated Ca2+ currents, postinhibitory rebound, and postspike afterhyperpolarization in lamprey neurons. *J Neurophysiol*, 70(6):2606–19. (Cited on page 10.)

McCabe, T. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320. (Cited on page 63.)

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press. (Cited on pages 63, 90, 92, and 222.)

McLean, D. L., Fan, J., Higashijima, S.-i., Hale, M. E., and Fetcho, J. R. (2007). A topographic map of recruitment in spinal cord. *Nature*, 446(7131):71–5. (Cited on page 7.)

Mecklenburg, R. (2004). *Managing Projects with GNU Make (Nutshell Handbooks)*. O'Reilly Media. (Cited on page 65.)

Merali, Z. (2010). Why scientific programming does not compute. *Nature*, pages 6–8. (Cited on page 61.)

Mercer, A., Bannister, A. P., and Thomson, A. M. (2006). Electrical coupling between pyramidal cells in adult cortical regions. *Brain Cell Biol*, 35(1):13–27. (Cited on page 97.)

Miller, G. (2006). Scientific publishing. A scientist's nightmare: software problem leads to five retractions. *Science*, 314(5807):1856–7. (Cited on page 63.)

Mills, S. and Massey, S. (1998). The kinetics of tracer movement through homologous gap junctions in the rabbit retina. *Vis Neurosci*, 15(4):765–777. (Cited on page 97.)

Molkov, Y. I., Loskutov, E. M., Mukhin, D. N., and Feigin, A. M. (2012). Random dynamical models from time series. *Phys Rev E*, 85(3):036216. (Cited on page 190.)

Morris, C. and Lecar, H. (1981). Voltage oscillations in the barnacle giant muscle fiber. *Biophys J*, 35(1):193–213. (Cited on pages 21 and 35.)

Moult, P. R., Cottrell, G. A., and Li, W.-C. (2013). Fast silencing reveals a lost role for reciprocal inhibition in locomotion. *Neuron*, 77(1):129–40. (Cited on pages 145, 187, and 188.)

Mullins, O., Hackett, J., and Buchanan, J. (2010). Neuronal control of swimming behavior: comparison of vertebrate and invertebrate model systems. *Prog Neurobiol*, 93(2):244–269. (Cited on page 7.)

Nakamura, Y. and Nobuo, K. (1995). Generation of masticatory rhythm in the brainstem. *Neurosci Res*, 23:1–19. (Cited on page 126.)

Noble, D. (1962). A modification of the Hodgkin-Huxley equations applicable to Purkinje fibre action and pacemaker potentials. *J Physiol*, pages 317–352. (Cited on page 9.)

Noble, D. (2004). Modeling the heart. *Physiology*, 19:191–7. (Cited on page 34.)

Nordlie, E., Gewaltig, M.-O., and Plesser, H. E. (2009). Towards reproducible descriptions of neuronal network models. *PloS Comput Biol*, 5(8). (Cited on pages 66 and 235.)

Nordlie, E. and Plesser, H. E. (2010). Visualizing neuronal network connectivity with connectivity pattern tables. *Front Neuroinform*, 3(January):39. (Cited on pages 66 and 235.)

Nowak, L., Bregestovski, P., Ascher, P., Herbet, A., and Prochiantz, A. (1984). Magnesium gates glutamate-activated channels in mouse central neurones. *Nature*, 307(5950):462–465. (Cited on pages 55 and 130.)

Olague, H. and Etzkorn, L. (2008). An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of Software*, 20(February):171–197. (Cited on page 64.)

Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20. (Cited on page 68.)

Orlovsky, G., Deliagina, T. G., and Grillner, S. (1999). *Neuronal Control of Locomotion: From Mollusc to Man*. Oxford University Press, Oxford. (Cited on pages 6, 10, and 150.)

Pan, F., Mills, S. L., and Massey, S. C. (2007). Screening of gap junction antagonists on dye coupling in the rabbit retina. *Vis Neurosci*, 24(4):609–18. (Cited on page 97.)

Pangratz-Fuehrer, S. and Hestrin, S. (2011). Synaptogenesis of electrical and GABAergic synapses of fast-spiking inhibitory neurons in the neocortex. *J Neurosci*, 31(30):10767–75. (Cited on page 97.)

Pappas, G. D. G. and Bennett, M. M. V. L. (1966). Specialized junctions involved in electrical transmission between neurons. *Ann NY Acad Sci*, 137(2):495–508. (Cited on page 57.)

Parker, D. (2006). Complexities and uncertainties of neuronal network function. *Philos T R Soc B*, 361(1465):81–99. (Cited on pages 181 and 182.)

Parker, D. (2009). Exciting times in the tadpole spinal cord. *J Physiol*, 587(Pt 8):1635. (Cited on page 21.)

Parker, D. (2010). Neuronal network analyses: premises, promises and uncertainties. *Philos T R Soc B*, 365(1551):2315–28. (Cited on pages 158, 182, and 186.)

Parnin, C. and Rugaber, S. (2012). Programmer information needs after memory failure. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 123–132. (Cited on page 63.)

Perge, J. A., Niven, J. E., Mugnaini, E., Balasubramanian, V., and Sterling, P. (2012). Why do axons differ in caliber? *J Neurosci*, 32(2):626–38. (Cited on page 119.)

Perkel, D. and Mulloney, B. (1974). Motor pattern production in reciprocally inhibitory neurons exhibiting postinhibitory rebound. *Science*, 185(4146):181. (Cited on pages 11 and 128.)

Perrins, R. and Roberts, A. (1995). Cholinergic and electrical synapses between synergistic spinal motoneurones in the *Xenopus laevis* embryo. *J Physiol*, 485 (Pt 1:135–44. (Cited on pages 19 and 53.)

Perrins, R., Walford, A., and Roberts, A. (2002). Sensory activation and role of inhibitory reticulospinal neurons that stop swimming in hatchling frog tadpoles. *J Neurosci*, 22(10):4229–40. (Cited on pages 53, 56, 128, 136, 137, and 138.)

Peterson, B. and Healy, M. (1996). ModelDB: an environment for running and storing computational models and their results applied to neuroscience. *J Am Med Inform Assn*, 3(6):389–98. (Cited on page 68.)

Petty, G. W. (2001). Automated computation and consistency checking of physical dimensions and units in scientific programs. *Software: Practice and Experience*, 31(11):1067–1076. (Cited on page 77.)

Pirtle, T. J. and Satterlie, R. a. (2007). The role of postinhibitory rebound in the locomotor central-pattern generator of *Clione limacina*. *Integr Comp Biol*, 47(4):451–6. (Cited on pages 9 and 10.)

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*. Foundation Books, 3rd editio edition. (Cited on pages 62 and 224.)

Prinz, A. a. (2010). Computational approaches to neuronal network analysis. *Philos T R Soc B*, 365(1551):2397–405. (Cited on page 12.)

Qu, Y. and Dahl, G. (2002). Function of the voltage gate of gap junction channels: selective exclusion of molecules. *P Natl Acad Sci USA*, 99(2):697–702. (Cited on page 56.)

Ramón y Cajal, S. (1909). *Histologie du système nerveux de l'homme & des vertébrés.* Maloine, Paris, ed. França edition. (Cited on page 29.)

Ray, S. and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front Neuroinform*, 2(December):6. (Cited on page 68.)

Ritchie, J. M. and Rogart, R. B. (1977). Density of sodium channels in mammalian myelinated nerve fibers and nature of the axonal membrane under the myelin sheath. *Proc Natl Acad Sci*, 74(1):211–5. (Cited on page 120.)

Roberts, A. (1980). The function and role of two types of mechanoreceptive 'free' nerve endings in the head skin of amphibian embryos. *J Comp Physiol*, 135(4):341–348. (Cited on pages 14 and 16.)

Roberts, A. (2000). Early functional organization of spinal neurons in developing lower vertebrates. *Brain Res Bull*, 53(5):585–93. (Cited on page 15.)

Roberts, A. and Alford, S. T. (1986). Descending projections and excitation during fictive swimming in *Xenopus* embryos: neuroanatomy and lesion experiments. *J Comp Neurol*, 250(2):253–61. (Cited on page 18.)

Roberts, A., Dale, N., Ottersen, O., and Storm-Mathisen, J. (1988). Development and characterization of commissural interneurones in the spinal cord of *Xenopus laevis* embryos revealed by antibodies to glycine. *Development*, 103(3):447–461. (Cited on pages 18 and 53.)

Roberts, A., Dale, N., Ottersen, O. P., and Storm-Mathisen, J. (1987). The early development of neurons with GABA immunoreactivity in the CNS of *Xenopus laevis* embryos. *J Comp Neurol*, 261(3):435–49. (Cited on page 53.)

Roberts, A., Li, W.-C., and Soffe, S. (2010). How neurons generate behavior in a hatchling amphibian tadpole: an outline. *Front Behav Neurosci*, 4(June):Article 16 (1–11). (Cited on pages 4, 7, 10, 13, 15, 16, 18, 98, 107, 128, 132, 145, 147, 151, 152, and 163.)

Roberts, A., Li, W.-C., Soffe, S. R., and Wolf, E. (2008). Origin of excitatory drive to a spinal locomotor network. *Brain Res Rev*, 57(1):22–8. (Cited on pages 10, 18, 53, 128, and 145.)

Roberts, A. and Tunstall, M. J. (1990). Mutual re-excitation with post-inhibitory rebound: a simulation study on the mechanisms for locomotor rhythm generation in the spinal cord of *Xenopus* embryos. *Eur J Neurosci*, 2(1):11–23. (Cited on pages 19, 20, 122, 128, and 147.)

Roberts, A., Walford, A., Soffe, S. R., and Yoshida, M. (1999). Motoneurons of the axial swimming muscles in hatchling *Xenopus* tadpoles: features, distribution, and central synapses. *J Comp Neurol*, 411(3):472–86. (Cited on page 14.)

Saffrey, P. and Orton, R. (2009). Version control of pathway models using XML patches. *BMC Syst Biol*, 3:34. (Cited on page 69.)

Safronov, B. V., Wolff, M., and Vogel, W. (1997). Functional distribution of three types of Na+ channel on soma and processes of dorsal horn neurones of rat spinal cord. *J Physiol*, 503 (Pt 2(1997):371–85. (Cited on pages 119 and 120.)

Sakmann, B. and Neher, E. (2010). *Single-Channel Recording*. Springer. (Cited on page 29.)

Sanes, D. H., Reh, T. A., and Harris, W. A. (2006). *Development of the nervous system*. Neuroscience Textbook Set. Elsevier Academic Press. (Cited on pages 12 and 20.)

Sattelle, D. B. and Buckingham, S. D. (2006). Invertebrate studies and their ongoing contributions to neuroscience. *Invert Neurosci*, 6(1):1–3. (Cited on page 7.)

Satterlie, R. (1985). Reciprocal inhibition and postinhibitory rebound produce reverberation in a locomotor pattern generator. *Science*, 229(4711):402–404. (Cited on pages 9 and 128.)

Sautois, B., Soffe, S. R., and Roberts, A. (2007). Role of type-specific neuron properties in a spinal cord motor network. *J Comp Neuro*, pages 59–77. (Cited on pages 12, 19, 20, 23, 53, 55, 56, 59, 98, 109, 112, 120, 121, 122, 126, 128, 130, 133, 144, 146, 147, 154, 173, and 197.)

Schmidt-Hieber, C. and Bischofberger, J. (2010). Fast sodium channel gating supports localized and efficient axonal action potential initiation. *J Neurosci*, 30(30):10233–42. (Cited on page 34.)

Schmidt-Hieber, C., Jonas, P., and Bischofberger, J. (2008). Action potential initiation and propagation in hippocampal mossy fibre axons. *J Physiol*, 586(7):1849–57. (Cited on pages 71, 119, and 226.)

Selverston, A. I. (1980). Are central pattern generators understandable? *Behav Brain Sci*, 3(04):535. (Cited on pages 181 and 182.)

Selverston, A. I. (2010). Invertebrate central pattern generator circuits. *Philos T R Soc B*, 365(1551):2329–45. (Cited on pages 4 and 126.)

Shik, M. L., Severin, F. V., and Orlovskii, G. N. (1966). Control of walking and running by means of electric stimulation of the midbrain. *Biofizika+*, 11(4):659–66. (Cited on page 150.)

Simon, A. (1999). Gap junctions: more roles and new structural data. *Trends Cell Biol*, 9(5):169. (Cited on page 19.)

Simon, A. M. and Goodenough, D. A. (1998). Diverse functions of vertebrate gap junctions. *Cell*, 8924(98):477–483. (Cited on pages 19, 56, and 97.)

Soffe, S., Roberts, A., and Li, W.-C. (2009). Defining the excitatory neurons that drive the locomotor rhythm in a simple vertebrate: insights into the origin of reticulospinal control. *J Physiol*, 587(Pt 20):4829–44. (Cited on pages 18, 53, 98, 107, 109, 127, 142, and 188.)

Soffe, S. R. (1989). Roles of glycinergic inhibition and N-Methyl-D-Aspartate receptor mediated excitation in the locomotor rhythmicity of one half of the *Xenopus* embryo central nervous system. *Eur J Neurosci*, 1(6):561–571. (Cited on pages 144, 145, 187, and 188.)

Soffe, S. R., Zhao, F. Y., and Roberts, a. (2001). Functional projection distances of spinal interneurons mediating reciprocal inhibition during swimming in *Xenopus* tadpoles. *Eur J Neurosci*, 13(3):617–27. (Cited on page 107.)

Sommerville, I. (2004). *Software Engineering, 7th Edition*. Addison Wesley. (Cited on pages 63 and 65.)

Squire, L., Berg, D., Bloom, F. E., du Lac, S., Ghosh, A., and Spitzer, N. C. (2012). *Fundamental Neuroscience*. Academic Press. (Cited on pages 30, 52, and 53.)

Srinivas, M., Rozental, R., Kojima, T., Dermietzel, R., Mehler, M., Condorelli, D. F., Kessler, J. a., and Spray, D. C. (1999). Functional properties of channels formed by the neuronal gap junction protein connexin36. *J Neurosci*, 19(22):9848–55. (Cited on page 119.)

Stein, P. S. G., Grillner, S., Selverston, A. I., and Stuart, D. G. (1997). *Neurons, Networks, and Motor Behavior*. A Bradford Book. (Cited on pages 6 and 7.)

Stephenson, A. G., LaPiana, L. S., Mulville, D. R., Rutledge, P. J., Bauer, F. H., Folta, D., Dukeman, G. A., and Sackheim, R. (1999). Mars Climate Orbiter Mishap Investigation Board Phase I Report. Technical report, NASA. (Cited on page 76.)

Stroustrup, B. (1997). *The C++ Programming Language: Third Edition*. Addison Wesley. (Cited on page 70.)

Sulston, J. E. and White, J. G. (1980). Regulation and cell autonomy during postembryonic development of *Caenorhabditis elegans*. *Dev Biol*, 78(2):577–597. (Cited on page 8.)

Szabo, T. M. and Zoran, M. J. (2007). Transient electrical coupling regulates formation of neuronal networks. *Brain Res*, 1129(1):63–71. (Cited on page 19.)

Tabak, J. and Moore, L. E. (1998). Simulation and parameter estimation study of a simple neuronal model of rhythm generation: role of NMDA and non-NMDA receptors. *J Comp Neuro*, 5(2):209–35. (Cited on page 143.)

Taylor, A., Cottrell, G., and Kristan, W. B. (2000). A model of the leech segmental swim central pattern generator. *Neurocomputing*, 33:573–584. (Cited on page 126.)

Taylor, A. L., Goaillard, J.-M., and Marder, E. (2009). How multiple conductances determine electrophysiological properties in a multicompartment model. *J Neurosci*, 29(17):5573–86. (Cited on page 12.)

Terman, D., Kopell, N., and Bose, A. (1998). Dynamics of two mutually coupled slow inhibitory neurons. *Physica D*. (Cited on page 176.)

Traub, R. D. and Bibbig, A. (2000). A model of high-frequency ripples in the hippocampus based on synaptic coupling plus axon-axon gap junctions between pyramidal neurons. *J Neurosci*, 20(6):2086–93. (Cited on page 97.)

Traub, R. D., Middleton, S. J., Knöpfel, T., and Whittington, M. a. (2008). Model of very fast (> 75 Hz) network oscillations generated by electrical coupling between the proximal axons of cerebellar Purkinje cells. *Eur J Neurosci*, 28(8):1603–16. (Cited on page 97.)

Tunstall, M. J., Roberts, A., and Soffe, S. R. (2002). Modelling inter-segmental coordination of neuronal oscillators: synaptic mechanisms for uni-directional coupling during swimming in *Xenopus* tadpoles. *J Comp Neuro*, 13(2):143–58. (Cited on pages 122, 128, and 147.)

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy array : a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):1–8. (Cited on page 68.)

van Heesch, D. (1997). Doxygen. http://www.stack.nl/ dimitri/doxygen/index.html. Accessed: 2013-06-27. (Cited on page 66.)

Várkonyi, P. L., Kiemel, T., Hoffman, K., Cohen, A. H., and Holmes, P. (2008). On the derivation and tuning of phase oscillator models for lamprey central pattern generators. *J Comp Neuro*, 25(2):245–61. (Cited on pages 11 and 12.)

Vivar, C., Traub, R. D., and Gutiérrez, R. (2012). Mixed electrical-chemical transmission between hippocampal mossy fibers and pyramidal cells. *Eur J Neurosci*, 35(1):76–82. (Cited on page 97.)

Volman, V., Perc, M., and Bazhenov, M. (2011). Gap junctions and epileptic seizures–two sides of the same coin? *PloS One*, 6(5):e20572. (Cited on page 97.)

Weeks, J. and Kristan Jr, W. (1978). Initiation, maintenance and modulation of swimming in the medicinal leech by the activity of a single neurone. *J Exp Biol*, 77(1):71. (Cited on page 8.)

West, J., Hoesen, G., and Kosel, K. (1982). A demonstration of hippocampal mossy fiber axon morphology using the anterograde transport of horseradish peroxidase. *Exp Brain Res*, 48:209–216. (Cited on page 119.)

Wheatley, M. and Stein, R. B. (1992). An *in vitro* preparation of the mudpuppy for simultaneous intracellular and electromyographic recording during locomotion. *J Neurosci Meth*, 42(1-2):129–37. (Cited on page 10.)

Whelan, P. (1996). Control of locomotion in the decerebrate cat. *Prog Neurobiol*, 49(5):481–515. (Cited on page 150.)

White, J. G., Southgate, E., Thomson, J. N., and Brenner, S. (1986). The structure of the nervous system of the nematode C. elegans. *Philos T R Soc B*, 314:1–340. (Cited on pages 8 and 13.)

Willows, A. O. D. (1967). Behavioral acts elicited by stimulation of single, identifiable brain cells. *Science*, 157(3788):570–574. (Cited on page 7.)

Wilson, D. and Waldron, I. (1968). Models for the generation of the motor output pattern in flying locusts. *P IEEE*, 169:1058–1064. (Cited on page 126.)

Wilson, D. M. (1961). The central nervous control of flight in a locust. *J Exp Biol*, 38(2):471–490. (Cited on page 4.)

Wilson, G. (2006). Where's the real bottleneck in scientific computing? *American Scientist*. (Cited on pages 61 and 189.)

Winlove, C. I. P. and Roberts, A. (2011). Pharmacology of currents underlying the different firing patterns of spinal sensory neurons and interneurons identified *in vivo* using multivariate analysis. *J Neurophysiol*, 105(5):2487–500. (Cited on pages 36, 37, 107, 142, 143, and 145.)

Winlove, C. I. P. and Roberts, A. (2012). The firing patterns of spinal neurons: *in situ* patch-clamp recordings reveal a key role for potassium currents. *Eur J Neurosci*, 36(7):2926–40. (Cited on pages 23, 37, 38, 39, 43, and 59.)

Wolf, E., Zhao, F. Y., and Roberts, A. (1998). Non-linear summation of excitatory synaptic inputs to small neurones: a case study in spinal motoneurones of the young *Xenopus* tadpole. *J Physiol*, 511 (Pt 3:871–86. (Cited on pages 36 and 100.)

Wollner, D. A. and Catterall, W. A. (1986). Localization of sodium channels in axon hillocks and initial segments of retinal ganglion cells. *P Natl Acad Sci USA*, 83(21):8424–8. (Cited on page 108.)

Woodfield, S., Dunsmore, H., and Shen, V. (1981). The effect of modularization and comments on program comprehension. In *5th international conference on Software engineering*, pages 215–223. (Cited on page 63.)

Zaytsev, Y. V. and Morrison, A. (2012). Increasing quality and managing complexity in neuroinformatics software development with continuous integration. *Front Neuroinform*, 6:31. (Cited on page 65.)

Zeng, S. and Tang, Y. (2009). Effect of clustered ion channels along an unmyelinated axon. *Phys Rev E*, 80(2):1–9. (Cited on page 120.)

Zheng, M., Iwasaki, T., and Friesen, W. O. (2004). Systems approach to modeling the neuronal CPG for leech swimming. In *Eng Med Biol Soc Ann*, volume 1, pages 703–6. (Cited on page 12.)