Federated Scripting in the GENESIS 3.0 Neural Simulation Platform

Cornelis H., Rodriguez A. L., Coop A. D. and Bower J. M.
University of Texas Health Science Center at San Antonio.

September 7, 2010

Abstract

The GENESIS (GEneral NEural SImulation System, http://genesis-sim.org/) simulation platform was one of the first broad-scale modeling systems in computational biology to encourage modelers to develop and share model features and components. Supported by a large developer community the GENESIS simulator participated in innovative simulator technologies such as benchmarking [Bhalla et al., 1992], parallelization [Goddard and Hood, 1998] and declarative model specification [Goddard et al., 2001] and it was the first neural simulator to define binding for the Python scripting language [Vanier, 1997].

An important feature of the latest version of GENESIS, GENESIS 3.0 (G-3), is that it decomposes into self-contained software components, that conform to the CBI simulator architecture [Cornelis et al., 2008]. This federated architecture allows separate scripting bindings to be defined for the mathematical solvers and the GUI, as well as for other necessary simulator components.

Python and Perl are scripting languages that provide rich sets of freely available open source libraries [Langtangen, 2004, Valiente, 2009]. With a clean dynamic object-oriented design producing highly readable code, Python and Perl are widely employed in specialized areas of software component integration [Thiruvathukal et al., 2001, Lee and Ware, 2007 SWIG (Simplified Wrapper and Interface Generator [08:, 2008]) examines an application programming interface (API) and makes it available to a scripting language of choice. This way the software components of the G-3 simulator can be glued together, instantiated and connected to external libraries and applications from user-defined scripts in either Python or Perl.

We illustrate this approach with examples using Python scripting. The first example uses a mathematical solver as a stand-alone software component driven from a Python script that generates and runs a simple single compartment model neuron. This script is then contrasted with C code and GENESIS 2 (G-2) implementations that connect to the same mathematical solver. The second example interfaces the mathematical solver to a modeling environment for the exploration of a neuron morphology from an interactive command-line and a graphical shell. The third example applies scripting bindings to connect the G-3 simulator to external graphical libraries and an open source 3D content creation suite. This allows us to visualize 3D models based on electron microscopy and convert them to computational models [Cornelis et al., 2007].

Employed this way the stand-alone software components of the G-3 simulator provide a framework for progressive federated development in the computational neurosciences.

1 Introduction

GENESIS is a general purpose simulation platform that was originally developed to support the simulation of neural systems ranging from subcellular components and biochemical reactions to complex models of single neurons, simulations of large networks, and systems-level models. The software development of the GENESIS sim-

ulator was initiated during the 1980's through research projects that addressed specific scientific questions in computational neuroscience and was then logically continued with a life cycle of research project extensions. For example the libraries for kinetic pathway modeling were added for projects investigating how signalling networks store learned behaviour [Bhalla and Iyengar, 1999] and how light regulates release from intracellular calcium stores for photoreception [Blackwell, 2000]. The fast implicit solver was developed with the specific focus of complex Purkinje cell modeling [De Schutter and Bower, 1994a, De Schutter and Bower, 1994b] and more recently synaptic learning rules have been implemented [Günay et al., 2008]. In principle such linear or single-threaded development processes can continue forever. However, repetitive extension of the GENESIS simulator with source code of diverse functions and origin ultimately made the code structure so complicated that it became increasingly difficult, if not impossible, to extend. Because of the density of the GENESIS 2 (G-2) source code the application became 'monolithic' while user contributions to simulation functionality were marginalized. Ultimately, releases and updates became less frequent and the software life cycle moved from extension to maintenance.

GENESIS 3 (G-3) is a major revision and update of the GENESIS simulation system. The core simulator functionality has been restructured, with a more modern modular design (the CBI federated software architecture, described below). This not only results in improved simulator performance and portability, but also allows the use of new script parsers and user interfaces, as well as the ability to communicate with other modeling programs and environments. The CBI federated software architecture is specifically designed to support the integration of stand-alone software components and applications by using common integration technologies such as modern scripting languages.

2 Methods & Software

Starting from the existing source code base, and taking lessons from the past, G-3 is a modularization of the core functions of the G-2 simulator. The guiding principle for the definition of the core functions of the G-3 simulator is what is referred to as the CBI federated software architecture, a modular abstracted architecture that layers the data in a simulator and separates the data representations from the algorithms to process them. This is described in more detail in the next sections.

2.1 GENESIS 2

GENESIS is a general purpose simulation platform that was developed to support the simulation of neural systems ranging from subcellular components and biochemical reactions to complex models of single neurons, simulations of large networks, and systems-level models. It was the first broad scale modeling system in computational biology to encourage modelers to develop and share model features and components. For these people, it was the object-oriented approach taken by the simulator along with its high-

level simulation language that allowed the exchange, modification, and reuse of models or model components.

GENESIS simulations are constructed from model components that receive inputs, perform calculations on them, and then generate outputs. Model neurons are constructed from basic parts, such as compartments, and variable conductance ion channels. Channels are linked to their compartments which are then linked together to form multi-compartmental neurons of any desired level of complexity. Neurons may be connected together to form neural circuits. It is the paradigm used by the GENESIS 2 script language interpreter (SLI), the commands which it recognizes, and the main GENESIS 'objects' available for constructing simulations that have most powerfully assisted in the sharing of model features amongst the broader modeling community.

A high-level simulation language, the GENESIS SLI [1], provided a framework within which a modeler could easily extend the capabilities of the simulator and manipulate models or model components by exchange, modification, and reuse. The SLI interprets statements in the GENESIS simulation language, and constitutes the operating system 'shell'. User-defined SLI scripts were used to glue the pieces of a simulation together. The graphical objects used to define the front end of a simulation and GENESIS data handlers were all controlled from SLI scripts.

Developed by Michael Vanier in the late 1990's, PyGENESIS was a version of GENESIS that replaced the standard GENESIS SLI with a Python interface [Vanier, 1997]. Leveraging the power and clear syntax of the Python scripting language PyGENESIS in principle also could easily be bound to external Python libraries and applications. It was nevertheless never publicly released due to the then immaturity of Python as a scripting language. However with the current sophistication of the Python platform and development of G-3 as a federated software architecture, Python has become a powerful integration tool for GENESIS as described below.

2.2 Scripting Languages

Historically, there have been fundamental differences between the Unix shells and system programming languages such as C or C++ and scripting languages such as Perl [Wall, 1999], Python [Martelli, 2006], Rexx [O'Hara and Gomberg, 1988], Tcl [Ousterhout, 1994], and Visual Basic. System programming languages start from the most primitive computer elements, usually the 'words' of memory. They are designed to manage the complexity of building data structures and algorithms from scratch and usually require pre-declared data types. Alternatively, scripting languages as a replacement for shell scripts and shell communication pipes are designed for 'gluing': they assume the existence of a set of powerful components and are intended primarily for connecting components together. In this way, scripting languages operate at a higher level than system programming languages in the sense that on average a single statement does more work. For example, a typical statement in a system programming language executes about five machine instructions,

[1] Note: The GENESIS SLI interface is the standard scripting language of GENESIS 2. It is also supported by G-3 with the backward compatibility component **NS-SLI**.

whereas in a scripting language hundreds or thousands of machine instructions may be executed [Ousterhout, 1998].

The strongly typed nature of system programming languages discourages reuse. Scripting languages, on the other hand, have actually stimulated significant software reuse. They use a model where interesting components are built in a system programming language and then glued together into applications using a scripting language. This division of labor provides a natural framework for reusability. Components are designed to be reusable, and there are well-defined interfaces between components and scripts that make them easy to use. In this sense scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power: system programming languages are used to create functional components which are then assembled using scripting languages.

In summary, system programming languages are well suited to building components where the complexity is in the data structures and algorithms, while scripting languages are well suited for integrating applications where the complexity is in the connections. With an increasing requirement for software integration, scripting is providing an important programming paradigm.

2.3 The CBI Federated Software Architecture

The CBI (Computational Biology Initiative) federated software architecture provides a modular paradigm that places stand-alone software components into logical relationships. Each software module is an independent and standalone component such that development and maintenance can be implemented concurrently.

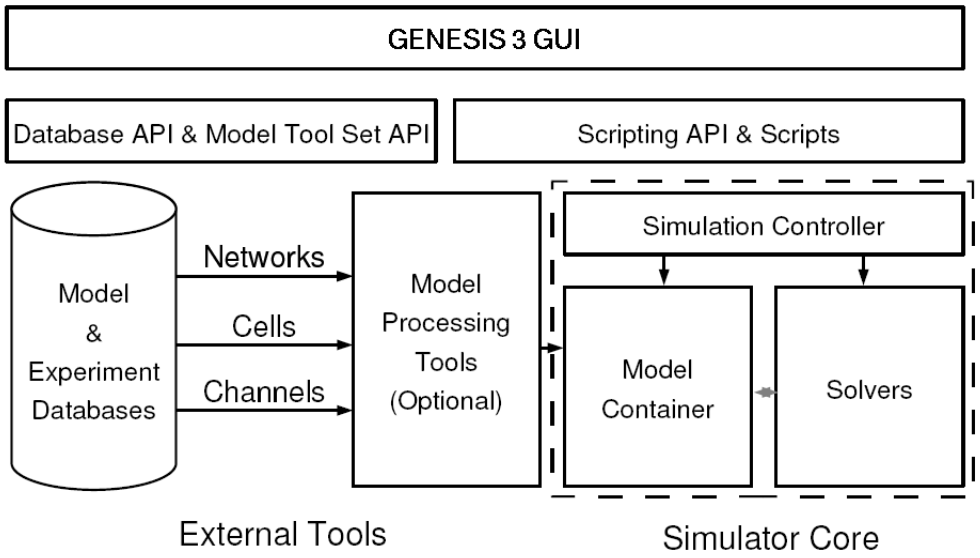

Figure 1: Relation of components in the CBI architecture.

The core components of the architecture are shown in Figure 1. On the bottom left

are databases of neuronal models or experimental data that can be accessed by the simulator. Optional model processors (e.g. the Reconstruct interface, http://synapses.clm.utexas.edu/to load a model into the **Model Container**. The **Model Container** stores a model in memory and makes it available to other software components in different formats. One function of the **Model Container** is to translate biological concepts and properties into mathematical concepts that can be understood by the mathematical solvers. Thus, importantly and unlike other existing neural simulators, the mathematical solvers are independent of the biological representation of a model. A simulation controller orchestrates and synchronizes the actions taken by the **Model Container** (e.g. when to load a model, the definition of the stimulus, and when to export a model) and mathematical solvers (when to fetch the model from the **Model Container**, when to start the calculations, and what the output variables are).

A scripting layer allows the simulation system to be driven from multiple scripting languages. Python and Perl are currently supported, as is (for backward compatibility) the GENESIS SLI. The G-3 Graphic User Interface or GUI (**G-Tube**), shown at the top of Figure 1, is entirely being developed in Python. It allows models to be imported from databases or constructed from scratch, the exploration of model structure and parameters, and the visualization of variables and model behavior.

Within the CBI paradigm each software component is self contained and can be run independently. This facilitates the interoperability of software obtained from different sources and has several important advantages for software development, including: (1) Reduced complexity of software components compared to a unitary system, (2) simplified documentation of components in terms of inputs and outputs, (3) as a consequence simplified development and testing of components as stand alone components, (4) clear delineation of scope for the development of new components, and (5) individual components can be independently updated, enhanced, or replaced when needed, making the life cycle of a modular architecture smoother than that of a non-scalable application.

The CBI federated software architecture provides a framework for the integration of independent software components into a functioning simulator using a scripting language of choice. Here we specifically illustrate the use of Perl and Python for this purpose.

2.4 G-3 as a CBI Compliant Simulator

Much existing software such as GUI libraries and plotting libraries, are application neutral. Other software packages are tailored to the needs of computational neuroscience. The Neurospaces project (http://www.neurospaces.org/) provides core software components of the G-3 simulator [Cornelis and De Schutter, 2003]. These include, (1) the **Model Container**: Stores two representations of a model, the first is conceptual and can be regarded as an enumeration of biological concepts and their relationships, the second is an expanded mathematical representation that, if complete, can be simulated, (2) **Heccer**: A fast compartmental solver based on the GENESIS *hsolve* object that can be instantiated from C, Perl, Python or other scripting languages, (3) **SSP** (Simple Scheduler in Perl): Binds **Heccer** and the **Model Container**, and activates

them correctly, such that they work together on a single simulation, (4) **Studio** and **G-Tube**: Contain graphical tools for model construction, exploration and simulation, (5) **G-Shell** (G-3 Interactive Shell): Dynamically loads other software components in an interactive environment, and the (6) **Project Browser**: For inspection of projects and simulation results. For completeness we also mention (7) **NS-SLI**: The G-3 component that provides backward compatibility for the GENESIS 2 SLI. All software can be downloaded from the GENESIS web site (http://genesis-sim.org/download/) and extensive installation instructions with examples are available from the GENESIS documentation website (http://www.genesis-sim.org/userdocs/genesis-installation/genesis-installation.html). Simulator correctness can be established by running automated regression and integration tests.

2.5 Perl

Perl was one of the first open source scripting languages. First released in 1987 [http://groups.google
it is unique in that it is very much informed by linguistic principles. Originally developed as a scripting language for UNIX, it aimed to blend the ease of use of the UNIX shell with the power and flexibility of a system programming language like C. With over 20 years of development and nearly half a million lines of code, Perl now runs on over 100 different platforms [ref: http://www.perl.org/about.html]. Currently, there are over 18,000 open source modules available from the Comprehensive Perl Archive Network (CPAN), assisting in system integration, scientific application, and user interface development. Via the CPAN Inline module, Perl integrates seemlessly with both system programming languages such as C and C++, and scripting languages including Python. Perl supports object-oriented programming, functional programming, and procedural programming paradigms. Perl source code has been certified to contain 0.03 defects per 1000 lines of code [http://scan.coverity.com/rung2.html]. In March 2010, 3.8 % of all lines of programming code were written in Perl to make it the 8th most popular programming language [Tiobe Software, 2010].

2.6 Python

Python is a powerful dynamic programming language comparable to Perl, Ruby, or Scheme. In February 2010 more than 4.2% of all code written was developed in Python to make it the 7th most popular programming language [Tiobe Software, 2010]. It combines considerable power with very clear syntax and has modules, classes, exceptions, and high level data types, in combination with a dynamic and loose typing. It runs on many hardware architectures, integrates with scientific and user interface libraries, and new modules are easily written in C or C++ (or other languages, depending on the chosen implementation). It is also usable as an extension language for applications written in other languages that need easy-to-use scripting or automation interfaces.

2.7 Meta-Programming in Perl and Python

Meta-programming is a programming technique where a program generates a new program and then executes it. Application of this technique for the G-3 Perl and Python bindings allows for the generation of an additional layer of script code that provides increased flexibility for the definition of models and simulations. A predefined Perl or Python data structure defines high-level interfaces and is translated into strings containing Perl or Python code such as class and method definitions. These, in turn, are then bound to the run-time environment using the Perl or Python *eval* functions during program initialization.

2.8 SWIG for Federated Software Integration

SWIG was chosen to facilitate the use of Perl and Python bindings in G-3. It is a software development tool that connects programs written in C and C++ with high-level scripting languages. For the CBI federated software architecture, it provides control over most aspects of wrapper generation and automates the generation of the required Perl and Python interfaces. SWIG uses a layered approach to build extension modules where different parts are defined in either C or the chosen scripting language. The C layer contains low-level wrappers whereas the script code is used to define high-level features. Considerably more flexibility is obtained by generating code in both languages as an extension module can be enhanced with support code in either language. Table 1 gives an overview of the resulting code. As expected, low-level software components emphasize low-level languages and have more lines of code (e.g. C), whereas, high-level software components emphasize high-level languages and have fewer lines of code (e.g. Python, Perl).

Language:	**C (H)**	**C (G)**	**Perl (H)**	**Perl (G)**	**Python (H)**	**Python (G**
Model Container	1,832,580	4,416,163	30,406	207,638	14,568	250,178
Heccer	1,163,991	1,575,615	57,565	107,261	1,586	171,219
NS-SLI	1,448,636	483,641	4,603	2,802	—	—
SSP	829	2,323	55,063	—	—	—
Studio	—	—	174,923	—	—	—
G-Shell	—	—	28,142	—	623	836

Table 1: **Languages Used:** Comparison of hand-written (H) and generated (G) code character counts.

3 Results

3.1 A Python Enabled Neural Simulator

Both Python and Perl use modules to group related functions together. The G-3 scripting bindings use modules to separate interfaces for simple models with many default settings (e.g. to start a new research project) from more complicated interfaces that expose the full functionality of the simulator.

As an example the Python *Neurospaces.SingleCellContainer* module contains functions to simplify the storage of single neuron models in computer memory. This module is a simplified front-end to the more complicated Neurospaces module. Neurospaces interfaces with the **Model Container** which is coded in an efficient system programming language. Likewise, *Heccer.SimpleHeccer* is a wrapper module around the **Heccer** component which in turn is an interface to the low-level single neuron solver. Other components are under construction to facilitate network modeling.

Here we show a simple high-level Python script [2] that runs a simulation of a single cylindrical segment defined by standard values for the parameters of membrane and axial resistance and membrane capacitance (`RM`, `RA`, and `CM`, respectively). These parameters are given by their specific values as commonly reported in the literature, instead of their actual values scaled to the compartment surface area as used by a mathematical solver [Cornelis and De Schutter, 2004]. The following script defines a Python function *run_simulation* and runs it when invoked from a shell command line. The script can also be imported as a Python module, thus allowing access to the function. We call this Python module *example*.

```
1  #!/usr/bin/python
2  # load the SingleCellContainer library
3  import sys
4  sys.path.append('/usr/local/glue/swig/python')
5  import Neurospaces.SingleCellContainer
6
7  # A function to run a simulation of a single cylindrical segment.
8
9  def run_simulation(simulationtime):
10
11     # create a cell for simulation
12     c = Neurospaces.SingleCellContainer.Cell("/cell");
13
14     # create a cylindrical segment inside the cell, and set its properties
15     s = Neurospaces.SingleCellContainer.Segment("/cell/soma");
16
17     s.parameter("Vm_init", -0.0680)
18     s.parameter("RM", 1.000)
19     s.parameter("RA", 2.50)
20     s.parameter("CM", 0.0164)
```

[2] The given code is written for clarity of the paper rather than for compactness or efficiency with relation to the scripting language used.

```
    s.parameter("ELEAK", -0.0800)

    s.parameter("DIA", 2e-05)
    s.parameter("LENGTH", 4.47e-05)

    # first example: apply current injection to the soma
    s.parameter("INJECT", 1e-9)

    # second example: use a wildcard to activate endogenous synapses
    Neurospaces.SingleCellContainer.query("setparameter spine::/Purk_spine/head/par 25")
    Neurospaces.SingleCellContainer.query("setparameter thickd::gaba::/Purk_GABA 1")

    # redirect output to the given file
    Neurospaces.SingleCellContainer.set_output_filename("/tmp/output")

    # compile the model
    Neurospaces.SingleCellContainer.compile("/cell")

    # define the output variables
    Neurospaces.SingleCellContainer.output("/cell/soma", "Vm")

    # run the simulation
    Neurospaces.SingleCellContainer.run(simulationtime)

# The main program executes a simulation of 0.5 seconds.
# The if statement allows this file to used as an executable script and as a library.

if __name__ == '__main__':
    run_sumulation(0.5)
```

Due to the CBI federated software architecture, the G-3 platform provides many user interfaces. As an example, the compartmental solver **Heccer** can be driven standalone from C code, from Python, or from Perl to run the simplest models, or it can be integrated with the **Model Container** for running more realistic multicompartment models based on morphological data. To illustrate this flexibility we now compare the above Python script with alternative implementations in C and the G-2 SLI.

In the C code there is an abundance of low level detail that interfaces directly to the solver. For example compartments are identified by their position in an array, and parameters such as `RM` and `CM` must be provided as an ordered sequence of their actual values (scaled to the compartment surface area).

The complexity of the G-2 SLI interface falls between that of the Python and Perl interfaces, and the C code interface. While compartments and parameters have names, numerical values are given in a format used by solvers.

C Code Implementation	**GENESIS 2 SLI Implementation**
#include "heccer/compartment.h"	
struct Compartment compSoma =	
{	
// type of structure	create neutral /cell
{ MATH_TYPE_Compartment, },	create compartment /cell/soma
	setfield /cell/soma dia 2e-05
-1, // no parent compartment	setfield /cell/soma len 4.47e-05
4.57537e-11, // Cm	setfield /cell/soma Cm 4.60608e-11
-0.08, // Em	setfield /cell/soma Em -0.0800
-0.068, // InitVm	setfield /cell/soma Vm_init -0.068
1e-9, // Inject 360502, // Ra	setfield /cell/soma Ra 355711
3.58441e+08, // Rm	setfield /cell/soma Rm 3.56051e+08
};	
	setfield /cell/soma inject 1e-9
// compartment and channel mapping	
int piC2m[] = 0, -1, ;	
// model definition	
struct Intermediary inte r =	
{ 1, &compSoma, NULL, piC2m, };	
// main simulation script	
#include "main.c"	reset
	step 0.5 -time

While Python and Perl bindings are suitable for construction of toy models from scratch, it is better to use a domain specific language to construct the various parts of a model. For example, the **Model Container** is installed with a library of domain specific model components where the standard Hodgkin-Huxley channels are provided in the file *channels/hodgkin-huxley.ndf*. These channels can be included in the *example* given above by adding the Python statements:

```
s.import_child("channels/hodgkin-huxley.ndf::/k")
s.import_child("channels/hodgkin-huxley.ndf::/na")
```

The **Model Container** can export models constructed in Perl, Python or other scripting languages as a library for incorporation into new models or for use with other tools such as the **Project Browser**. These new models can then be imported by a call to the `Neurospaces` *read* method. For example, importing a Purkinje cell model with over 4000 compartments may be done with the following statement:

```
Neurospaces.SingleCellContainer.read("cells/purkinje/edsjb1994.ndf")
```

After importation the **Model Container** provides a set of functions to analyze the structure of the model morphology. For example, the names of the most distal segment of each dendrite can be obtained with:

```
Neurospaces.SingleCellContainer.query("segmentertips /Purkinje")
```

3.2 Interactive Query and Simulation

The **G-Shell** is a G-3 software component that integrates other software components and makes their functions available through an interactive environment. Coded in Perl, the **G-Shell** is a communication abstraction layer for other software components such as the **Model Container**, **Heccer**, **SSP** and the **Studio**. After the **G-Shell** has been started from a system shell with

```
genesis-g3
```

the list of loaded software components is printed to the screen after issuing the command:

```
list components
```

Each loaded software component will be shown with associated status information helping in the diagnosis of possible problems. For example after correct initialization of the **Model Container** its status information should appear as:

```
model-container:
  description: internal storage for neuronal models
  integrator: Neurospaces::Integrators::Commands
  module: Neurospaces
  status: loaded
  type:
    description: intermediary
    layer: 2
```

Integration of the **G-Shell** with the **Model Container** allows for real-time analysis of the quantitative and structural aspects of a neuronal morphology. The library of model components that is installed with the **Model Container** provides a definition of a model Purkinje cell in the file *cells/purkinje/edsjb1994.ndf*. The command:

```
ndf_load cells/purkinje/edsjb1994.ndf
```

will make the model Purkinje cell available for interactive analysis. Alternatively, if the model is encoded in a GENESIS 2 SLI script with name *PurkM9_model/CURRENT9.g* the command *ndf_load* can be replaced with *sli_load*:

```
sli_load PurkM9_model/CURRENT9.g
```

This command imports the model that is specified in the SLI script without running the simulation. A similar command (*pynn_load*) is in development to interface with the PyNN network modeling environment [Davison et al., 2008].

Given the name of one of its dendritic segments, the number of branch points between that segment and the soma can be determined. After indicating which paths of the dendritic tree must be examined, the parameter `SOMATOPETAL_BRANCHPOINTS` contains the result, which can be obtained with:

```
morphology_summarize /Purkinje
show_parameter /Purkinje/segments/b1s06[182] SOMATOPETAL_BRANCHPOINTS
```

After finding a suitable dendritic segment, its synaptic channel can be stimulated with a precomputed spike train that is stored in a file with, for example, the filename *event_data/events.yml*:

```
set_runtime_parameter /Purkinje/segments/b1s06[182]/Purkinje_spine_0/head/par/synapse
   EVENT_FILENAME ‘‘event_data/events.yml’’
```

Finally, following the addition of an output comprising the somatic membrane potential, a simulation can conveniently be started using:

```
add_output /Purkinje/segments/soma Vm
run /Purkinje 0.1
```

This outputs the somatic response to the stimulus in a file named by default as */tmp/output*.

To query the parameters of the stimulated compartment the model can then be analyzed using the graphical front-end of the **Studio** with the command:

```
explore
```

Figure 2 shows sample output of running this command. Other capabilities of the **Studio** include rendering morphologies in three dimensions and generating overviews of network models (not shown). In the next section we explore more graphical capabilities of G-3.

3.3 Gluing Pre-existing Applications & Libraries

In the past, the graphical interface to G-2 was provided by the X-Window System Output and Display Utility for Simulations (XODUS). The XODUS interface made graphical objects available that could be connected to model components from within the SLI. Rather than providing a full GUI instance, the flexibility of XODUS came from its infrastructure which allowed modelers to easily develop new GUIs dedicated to their research and teaching projects[3]. However, the XODUS paradigm inevitably allowed modelers to contaminate their model script with GUI related statements.

[3]The official G-2 software distribution contains both simple and sophisticated example GUIs.

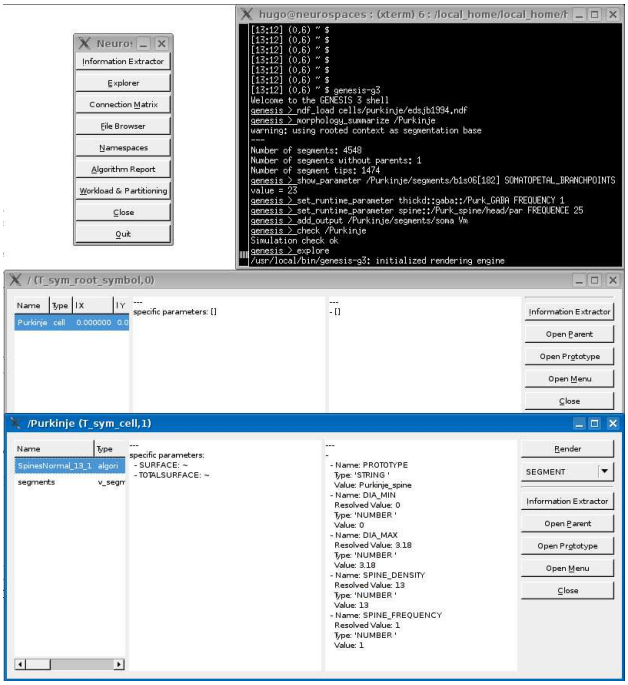

Figure 2: Using the **Studio** to query a model and its parameters.

As mentioned above, one advantage of the CBI federated software architecture is that it defines how to interface simulator components with external applications. An obvious example is the use of existing 3D graphics software to examine and edit the spatial properties of a model neuron morphology. Others include, integration with external graphing and windowing software to plot the values of solved variables against simulation time, or to allow the construction of button-rich tutorial applications.

GUI libraries typically communicate with other software components using an event based system. The functional core of this system is an event dispatching loop, usually called the *main loop*. The binding between button click event and the *main loop*, and the visual layout of most contemporary GUI applications is conveniently constructed using one of a number of available user interface builders. *wxFormBuilder*[4] is such an interface builder which allows a user to construct a GUI with visual elements such as menus and buttons, and write a description of the elements and their bindings to a file known as an XML resource (XRC) file. The GUI definitions in this file can then be rendered with the freely available *wxWidgets* library and its Python front end *wxPython*. Further integration with additional G-3 specific data bindings ensures that, for example, the data produced by a mathematical solver flows to a widget that plots the value of a variable against time. This functionality replaces the G-2 XODUS paradigm, that required SLI scripting to connect GUI components to model components and simulation actions, with a more contemporary paradigm that separates simulator and model scripts from GUI

[4]A user interface designer for the *wxPython* toolkit and the Linux desktop environment GNOME, available from http://wxformbuilder.org/.

related statements.

In the following example we create a *wxPython* application class called *G3App*. This demonstrates the Python scripting required to connect the software components that create a small GUI for G-3. We specifically show how to initialize the application (implementation of method *OnInit*), how to run a simple simulation based on the previous *example* (method *OnRun*), and how to plot output (method *Plot*). For this, it is assumed that a XRC file with the name *G3.xrc* can be found that describes a GUI with one frame (here, *mainFrame*) which allows the simulation duration to be set via a text control and contains a button to start the simulation.

The first lines of code in the script load the necessary Python modules which then load low-level libraries coded in a system programming language. The importation of *example* makes the previously defined function *run_simulation* available. The *DataPlot* class is a specialized class to read in GENESIS data output and load it into a *wxPython* plot widget. The import of *wx* references a system wide install of *wxPython* and makes the GUI functions of *wxWidgets* available to our script.

```
1  import DataPlot
2  import example
3  import wx
4  from wx import xrc
```

To ensure correct system initialization via the method *OnInit*, our *G3App* class is declared to inherit the functions of the *wx.App* class.

```
5  class G3App(wx.App):
6
7      def OnInit(self):
```

After correct system initialization, application specific initialization can start. Inside the *OnInit* method we first load the XML resource file previously created using *wxFormBuilder*.

```
8          self.res = xrc.XmlResource('G3.xrc')
```

The GUI elements are then retrieved from the XRC specification and made available as Python objects. Each declared element can be retrieved via its name:

```
9          self.frame = self.res.LoadFrame(None, 'mainFrame')
10         self.durationTextCtrl = xrc.XRCCTRL(self.frame,'durationTextCtrl')
11         self.runButton = xrc.XRCCTRL(self.frame, 'runButton')
```

After retrieving the run button, we bind it to the method *OnRun* (given below). This translates the GUI event generated when the run button is clicked to an action that invokes the *OnRun* method.

```
12         self.frame.Bind(wx.EVT_BUTTON, self.OnRun, self.runButton)
```

The *OnRun* method reads a numerical value for the the simulation time from a text control widget (*durationTextCtrl*) and stores it in a variable. This variable is then passed to the function *run_ simulation*. After the simulation is complete a call to a *Plot* method is made. This displays the generated data in a *wxPython* plot widget.

```
13      def OnRun(self,evt):
14
15          simulation_time = float(self.durationTextCtrl.GetValue())
16          example.run_simulation(simulation_time)
17          self.Plot('/tmp/output')
```

The *Plot* method uses the *DataPlot* class to display G-3 data output with a *wxPython* plot widget. The *DataPlot* widget is part of the libraries of the **G-Tube**, a Python GUI under development for G-3.

```
18      def Plot(self,datafile):
19
20          plotwindow = wx.Frame(self.frame, -1, "Graph display", (480,300))
21          plotpanel = wx.Panel(plotwindow, -1)
22
23          self.dataplot = DataPlot.DataPlot(plotpanel, -1,
24                                            '/tmp/output',
25                                            'Example Plot',
26                                            'Time (Seconds)',
27                                            'Membrane Potential (Volts)')
28
29          vbox_sizer = wx.BoxSizer(wx.VERTICAL)
30          vbox_sizer.Add(self.dataplot, 1, wx.EXPAND)
31          plotpanel.SetAutoLayout(True)
32          plotpanel.SetSizer(vbox_sizer)
33          plotpanel.Layout()
34          plotwindow.Show()
```

The code of the GUI application (*G3App*) is terminated with a call to the main event loop of *wxPython*.

```
35  if __name__ == '__main__':
36      app = G3App(False)
37      app.MainLoop()
```

In this example we have shown how the CBI architecture defines a separation between GUI statements and peripheral code such as input and output specifications, and model construction. Besides allowing common GUI construction kits to be used for the development of research and educational projects, the approach also allows interfacing to more specialized GUI kits. This is illustrated with the following example.

3.4 Interfacing GENESIS with Blender

Blender (http://www.blender.org/) is a free open source 3D content creation suite available for all major operating systems that have Python enabled bindings. The Python environment of Blender has the restriction that the code must be run from inside the Blender specific Python interpreter. In doing this, Blender replaces the functionality otherwise provided by the **G-Shell**. It allows the state-of-the-art rendering functions of Blender to be used to validate and analyze models of the morphology of small dendritic segments obtained from electron microscopy data.

Over the last several years electron microscopy (EM) in conjunction with Reconstruct [Fiala, 2005] has been used to obtain precise morphologies of small segments of Purkinje cell dendrites [Huo et al., 2009, Cornelis et al., 2007].

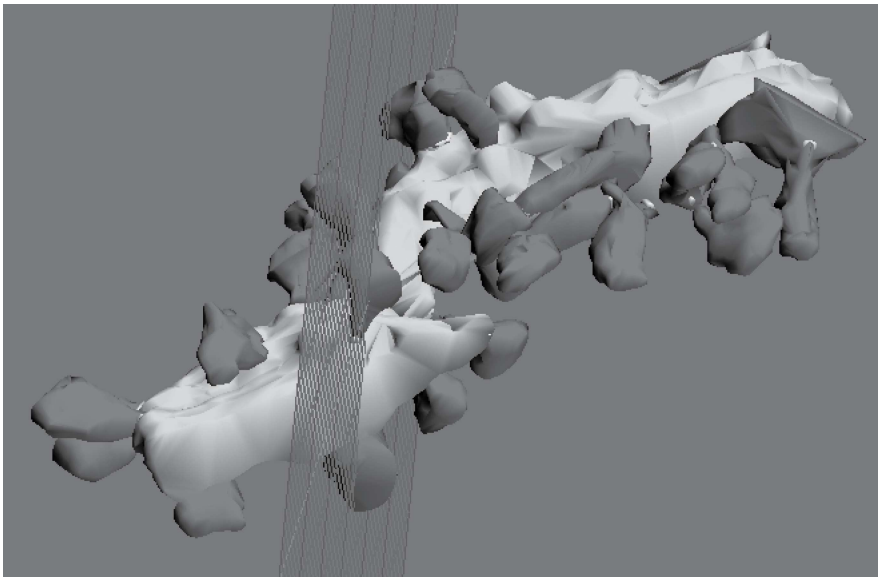

Figure 3: Blender image of Purkinje neuron dendritic segment.

The Reconstruct interface converts the Reconstruct application into a G-3 simulator component by making it CBI compliant. This allows Reconstruct data to be imported into the **Model Container**. The core of the interface implements geometrical transformation algorithms that convert EM contours provided by Reconstruct to equivalent cylinders suitable for cable modeling. The geometrical properties of the cylinders are stored in the native G-3 file format and algorithms provided by the **Model Container** link them with the cable parameters required by the mathematical solvers. A simulation can then be run with the *read* and *run* methods given above.

The necessary conversion algorithms are accessible from the **Model Container** via Python. The Python interface of Blender links it to the G-3 simulator, such that Blender is the first G-3 3D model inspection tool for EM data. As an example, the Python script developed above can be run from within the Blender environment. Also, via the same Python interface, simulations can be started based on the 3D image data.

Interactive visualization of reconstructed dendritic segments is a valuable method of model validation and is available with the interface of G-3 with Blender (see Figure 3). However, the development of small focused plugins allows for more than just these functions. For example, 3D measurement and manipulation of neuron morphology, computation of surface areas and volumes, and the generation of 3D crossections and 2D cuts also becomes possible.

4 Discussion

The Python and Perl bindings of the G-3 simulator embed similar concepts to the G-2 SLI, although their purpose is different. While the SLI had as major goals the integration of model components, running simulations, and output collection, the primary goal of scripting languages such as Perl and Python has become application integration.

Software libraries can either provide general support or can be tailored for specific scientific disciplines. Through the Neurospaces project, GENESIS now provides a series of independent software components that can be combined to support computational modeling in the neurosciences. Scripting languages such as Python and Perl then provide powerful integration tools to connect these software components to general purpose libraries for GUI application development, result visualization and data analysis.

4.1 From Monolithic Software Applications to Modular Frameworks

The development of the GENESIS simulator was initiated during the eighties through research projects that addressed specific scientific questions in computational neuroscience. Software development was then continued with a life cycle of research project extensions. For example the libraries for kinetic pathway modeling were added for projects investigating how signalling networks store learned behaviour [Bhalla and Iyengar, 1999] and how light regulates release from intracellular calcium stores for photoreception [Blackwell, 2000]. The fast implicit solver was developed with the specific focus of complex Purkinje cell modeling [De Schutter and Bower, 1994a, De Schutter and Bower, 1994b] and more recently synaptic learning rules have been implemented [Günay et al., 2008]. In principle such linear or single-threaded development processes can continue forever. However, repetitive extension of the GENESIS simulator with source code of diverse functions and origin ultimately made the code structure so complicated that it became difficult, if not impossible, to extend. Because of the density of the G-2 source code the application became become 'monolithic' and user contributions to simulation functionality were minimalized.

In the paradigm of the CBI federated software architecture, model parameters are stored and processed separately from stimulus protocols and the way simulations are run. This greatly facilitates the development and maintenance of individual software components for G-3. As examples of the added functionality this approach allows, we

have shown the ability of the **Model Container** to query the structure of a neuronal morphology, and how to run a simple model using **Heccer**. The CBI federated software architecture also defines clear boundaries for integration using existing scripting technology. For example, an interactive simulation of a Purkinje cell model after unitary synaptic stimulation can be connected to a predefined spike train stored in a file. In this way software improvements can be achieved using parallel software development processes for stand-alone software components rather than the more linear ones typical of centrally developed monolithic software applications.

4.2 Extensibility in The G-3 Software Federation

An important benefit of the CBI federated software architecture is that third party software libraries become available for users. For example, *wxFormBuilder* can be used to generate GUI bindings for *wxPython* and integrate them with the G-3 software platform.

To demonstrate the additional power of our approach, we have interfaced G-3 with Reconstruct and Blender. This novel software platform has been used for visual inspection and validation of reconstructed dendrites by connecting a model to the geometrical and analytical tools provided by the Blender plugin library. Further, we note that it is also possible to use Blender to instantiate neural simulations and, for example, to collect simulation output data for movie generation. We now give an overview of our ongoing efforts to interface the G-3 simulator with external libraries and applications.

Complementary functionality to that provided by interfacing G-3 with Blender would be available after interfacing G-3 with *neuro*Construct (http://www.neuroconstruct.org/), a software package designed to simplify the development of complex networks of biologically realistic neurons [Gleeson, 2005, Gleeson et al., 2007]. Implemented in Java, *neuro*Construct uses the latest NeuroML specifications (see http://www.neuroml.org/, http://www.morphml.org/), can be used to visually validate network layout and design [Crook et al., 2007], and can be connected to Python applications (e.g. see http://www.jython.o In principle this allows it to be integrated with other simulators that have Python bindings, including NEURON, NEST, and G-3.

A serial communication framework for event delivery of action potentials to postsynaptic targets has been developed. Called the Discrete Event System (DES), this software component is integrated with the mathematical solvers of G-3 using either Perl or Python. Because it is optimized for communication over serial hardware, DES can be extended to support communication frameworks for parallel hardware such as those provided by the MOOSE simulator [Ray and Bhalla, 2008] and the MUSIC framework [Djurfeldt et al., 2010].

The NeuroMorpho.Org database of neuronal morphologies (http://www.neuromorpho.org/) is a centrally curated inventory of digitally reconstructed neurons [Ascoli, 2006]. The digital reconstruction of neuronal arborization is an important step in the quantitative investigation of cellular neuroanatomy. Allowing extensive morphometric analysis, it is the first step in the implementation of biophysical models of electrophysiology. Direct interfacing with the functionality of the **Model Container** accelerates the development

of neuronal models by providing a direct link to data from experiments. Preliminary implementations of this functionality are now part of an automated test framework for G-3.

G-3 also significantly extends the ability of GENESIS to directly interact with experimental technologies such as open source dynamic clamp software. As an example, the modular approach taken by the RTXI platform for dynamic clamp [Bettencourt et al., 2008, Dorval et al., 2001] and the modular structure of G-3 mean that the solver, **Heccer**, can be directly integrated as an RTXI plug-in [Cornelis and Coop, 2010]. This greatly simplifies the required software development.

Ultimately, the extensibility of the CBI federated software architecture provides an extremely plastic environment within which independent components can be integrated with a scripting language of choice.

4.3 Implications for Neuronal Simulator Interoperability

The current generation of neural simulators can be characterized as software applications that support a user workflow extending from model construction to data analysis. Many of these simulators support Python bindings because of their ease of use [Pecevski et al., 2009] and simplicity [Goodman and Brette, 2008]. They range from Monte-Carlo simulators for reaction-diffusion systems [Wils and Schutter, 2009] and dedicated large network simulators [Eppler et al., 2008] to the general purpose NEURON and GENESIS 2 simulators [Hines et al., 2009, Bower and Beeman, 1998].

For these simulators interoperability is more easily implemented using one of the emerging standards for model exchange such as NeuroML [Goddard et al., 2001], NineML [Gortechn and PyNN [Davison et al., 2008]. While dedicated G-3 modules supporting the use of these interoperability standards are currently under development, the G-3 platform now also provides an alternative approach that uses scripting to connect neuroscience specific software to general purpose software and integrate it into a next generation neural simulator.

4.4 Federated Software Development in Neuroscience

Processes of software development have traditionally been described as either cathedral-style where there is a closed developer group under central direction and software releases are infrequent, or, alternatively, bazaar-style where the software is developed by volunteers and software releases occur early and often [Raymond, 2001, Brooks, 1995]. While cathederal-style software development leads to a single-threaded development cycle commonly used by commercial applications, the bazaar-style leads to multi-threaded development cycles of applications that come in different flavours [5].

Here, based on the CBI paradigm, we have outlined a solution for multi-threaded development of software components for neuroscience (for other examples of this ap-

[5] A typical example is the family of editors based on Emacs.

proach to neural simulation see [King et al., 2009, Nordlie and Plesser, 2009]). We have given examples that use Python and Perl.

Employed in this way, the modularized design of the G-3 simulator gives rise to an ecology of software components that can be glued together in a variety of ways providing for progressive federated software development.

Acknowledgements

We acknowledge Ja-Lyoung Joe of the College of Medicine, Wonkwang University, Republic of Korea, both for fruitful discussion and for his parallel work on a new Python implementation of GENESIS available from http://sourceforge.net/. We also thank the Computational Biology Initiative at UTSA (http://www.cbi.utsa.edu) for their excellent support when installing and updating G-3 on their computers.

Hugo Cornelis is partially supported by the CREA Financing program (CREA/07/027) of the K.U.Leuven, Belgium, EU. Both Hugo Cornelis and Allan D. Coop are partially supported by NIH grant 3 R01 NS049288-06S1 to James M Bower.

References

[08:, 2008] (2008). Simplified wrapper interface generator. World Wide Web.

[Ascoli, 2006] Ascoli, G. (2006). Mobilizing the base of neuroscience data: the case of neuronal morphologies. *Nature Rev. Neurosci.*, 7:318–324.

[Bettencourt et al., 2008] Bettencourt, J., Lillis, K., Stupin, L., and White, J. (2008). Effects of imperfect dynamic clamp: Computational and experimental results. *Journal of Neuroscience Methods*, 169(2):282–289.

[Bhalla et al., 1992] Bhalla, U., Bilitch, D., and Bower, J. (1992). Rallpacks: A set of benchmarks for neuronal simulators. *TRENDS in Neurosciences*, 15(11):453–458.

[Bhalla and Iyengar, 1999] Bhalla, U. and Iyengar, R. (1999). Emergent properties of networks of biological signaling pathways. *Science*, 283:381–387.

[Blackwell, 2000] Blackwell, K. (2000). Evidence for a distinct light-induced calcium-dependent potassium current in hermissenda crassicornis. *Journal of Computational Neuroscience*, 9(2):149–170.

[Bower and Beeman, 1998] Bower, J. M. and Beeman, D., editors (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* Springer-Verlag, New York, second edition.

[Brooks, 1995] Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition).* Addison-Wesley Professional, 2 edition.

[Cornelis and Coop, 2010] Cornelis, H. and Coop, A. D. (2010). Realtime tuning and verification of compartmental cell models using RTXI and GENESIS. In *Twentieth Annual Computational Neuroscience Meeting CNS*2010 July 2010.* Submitted.

[Cornelis and De Schutter, 2003] Cornelis, H. and De Schutter, E. (2003). Neurospaces : Separating modeling and simulation. *Neurocomputing*, 52–54:227–231.

[Cornelis and De Schutter, 2004] Cornelis, H. and De Schutter, E. (2004). Neurospaces parameter handling. *Neurocomputing*, 58–60:1079–1084.

[Cornelis et al., 2008] Cornelis, H., Edwards, M., Coop, A., and Bower, J. (2008). The CBI architecture for computational simulation of realistic neurons and circuits in the GENESIS 3 software federation. *BMC Neuroscience*, 9(S1):P88.

[Cornelis et al., 2007] Cornelis, H., Lu, H., Esquivel, A., and Bower, J. (2007). Modeling a single dendritic compartment using Neurospaces and GENESIS-3. *BMC Neuroscience*, 8(S2):P3.

[Crook et al., 2007] Crook, S., Gleeson, P., Howell, F., Svitak, J., and Silver, A. (2007). MorphML: Level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics*, 5(2):96–104.

[Davison et al., 2008] Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2.

[De Schutter and Bower, 1994a] De Schutter, E. and Bower, J. (1994a). An active membrane model of the cerebellar purkinje cell I. simulation of current clamps in slice. *Journal of Neurophysiology*, 71:375–400.

[De Schutter and Bower, 1994b] De Schutter, E. and Bower, J. M. (1994b). An active membrane model of the cerebellar purkinje cell II. simulation of synaptic responses. *Journal of Neurophysiology*, 71:401–419.

[Djurfeldt et al., 2010] Djurfeldt, M., Hjorth, J., Eppler, J. M., Dudani, N., Helias, M., Potjans, T. C., Bhalla, U. S., Diesmann, M., Kotaleski, J. H., and Ekeberg, O. (2010). Run-time interoperability between neuronal network simulators based on the music framework. *Neuroinformatics*, 8(1):43–60.

[Dorval et al., 2001] Dorval, A., Christini, D., and White, T. (2001). Real-time linux dynamic clamp: A fast and flexible way to construct virtual ion channels in living cells. *Annals of Biomededical Engineering*, 29:897–907.

[Eppler et al., 2008] Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: A convenient interface to the nest simulator. *Frontiers in neuroinformatics*, 2(12). DOI: 10.3389/neuro.11.012.2008.

[Fiala, 2005] Fiala, J. (2005). Reconstruct: a free editor for serial section microscopy. *Journal of Microscopy*, 218:52–61. http://www.bu.edu/neural/Reconstruct.html.

[Gleeson, 2005] Gleeson, P. (2005). Building 3D network models with neuroConstruct. World Wide Web. Tutorial at the Wam-Bam meeting, http://wambamm.org/WB05/Tutorials/advanced-tutorials/gleeson/index.html.

[Gleeson et al., 2007] Gleeson, P., Steuber, V., and Silver, R. A. (2007). neuroconstruct: A tool for modeling networks of neurons in 3d space. *Neuron*, 54:219–235.

[Goddard and Hood, 1998] Goddard, N. and Hood, G. (1998). Large-scale simulation using parallel genesis. In Bower, J. and Beeman, D., editors, *The Book of GENESIS*, chapter 21. Springer-Verlag, 2nd edition.

[Goddard et al., 2001] 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. *Philosophical Transactions of the Royal Society, Series B: Biological Sciences*, 356:1–20. Theme Issue organized and edited by Rolf

Kötter on "Neuroscience databases - tools for exploring brain structure-function relationships".

[Goodman and Brette, 2008] Goodman, D. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics*, 2(5). DOI: 10.3389/neuro.11.005.2008.

[Gortechnikov and the INCF NineML Task Force, 2010] Gortechnikov, A. and the INCF NineML Task Force (2010). The NineML user layer. In *Twentieth Annual Computational Neuroscience Meeting CNS*2010 July 2010.*, San Antonio, USA.

[Günay et al., 2008] Günay, C., Edgerton, J. R., , and Jaeger, D. (2008). Channel density distributions explain spiking variability in the globus pallidus: A combined physiology and computer simulation database approach. *J. Neurosci.*, 28:7476–7491.

[Hines et al., 2009] Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and python. *Frontiers in neuroinformatics*, 3(1). DOI: 10.3389/neuro.11.001.2009.

[Huo et al., 2009] Huo, L., Esquivel, A. V., and Bower, J. M. (2009). 3D electron microscopic reconstruction of segments of rat cerebellar purkinje cell dendrites receiving ascending and parallel fiber granule cell synaptic inputs. *The Journal of Comparative Neurology*, 514(6):583–94.

[King et al., 2009] King, J. G., Hines, M., Hill, S. L., Goodman, P. H., Markram, H., and Schürmann, F. (2009). A component-based extension framework for large-scale parallel simulations in NEURON. *Frontiers in Neuroinformatics*.

[Langtangen, 2004] Langtangen, H. P. (2004). *Python Scripting for Computational Science*. Springer-Verlag.

[Lee and Ware, 2007] Lee, J. and Ware, B. (2007). *Open Source Development with LAMP: Using Linux, Apache, MySQL, Perl, and PHP*. Addison-Wesley.

[Martelli, 2006] Martelli, A. (2006). *Python in a Nutshell*. O'Reilly Media, Inc.

[Nordlie and Plesser, 2009] Nordlie, E. and Plesser, H. E. (2009). Visualizing neuronal network connectivity with connectivity pattern tables. *Frontiers in Neuroinformatics*.

[O'Hara and Gomberg, 1988] O'Hara, R. and Gomberg, D. (1988). *Modern Programming Using REXX*. Prentice Hall. ISBN 0-13-597329-5.

[Ousterhout, 1994] Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley. ISBN 0-201-63337-X.

[Ousterhout, 1998] Ousterhout, J. K. (1998). Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31:23–30.

[Pecevski et al., 2009] Pecevski, D., Natschläger, T., and Schuch, K. (2009). PCSIM: A parallel simulation environment for neural circuits fully integrated with python. DOI: 10.3389/neuro.11.011.2009.

[Ray and Bhalla, 2008] Ray, S. and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in python for MOOSE. *Frontiers in Neuroinformatics*.

[Raymond, 2001] Raymond, E. S. (2001). *The Cathedral and the Bazaar*. O'Reilly Media.

[Thiruvathukal et al., 2001] Thiruvathukal, G. K., Christopher, T. W., and Shafaee, J. P. (2001). *Web Programming in Python: Techniques for Integrating Linux, Apache and MySQL*. Prentice Hall. ISBN: 0130410659.

[Tiobe Software, 2010] Tiobe Software (2010). Tiobe programming community index. World Wide Web. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[Valiente, 2009] Valiente, G. (2009). *Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R*. Addison-Wesley.

[Vanier, 1997] Vanier, M. C. (1997). A version of the neural simulator GENESIS that uses python. World Wide Web. http://www.cs.caltech.edu/~mvanier/hacking/pygenesis/pygenesis.tar.gz.

[Wall, 1999] Wall, L. (1999). *Perl Programmers Reference Guide*. Get a more recent reference.

[Wils and Schutter, 2009] Wils, S. and Schutter, E. D. (2009). STEPS: Modeling and simulating complex reaction-diffusion systems with python. *Frontiers in Neuroinformatics*, 3(15). DOI: 10.3389/neuro.11.015.2009.