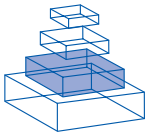

Trends in programming languages for neuroscience simulations

Andrew P. Davison[1]*, Michael L. Hines[2] and Eilif Muller[3]

[1] *Unité de Neurosciences Intégratives et Computationnelles, Centre National de la Recherche Scientifique, Gif sur Yvette, France*

[2] *Computer Science, Yale University, New Haven, CT, USA*

[3] *Laboratory for Computational Neuroscience, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*

Neuroscience simulators allow scientists to express models in terms of biological concepts, without having to concern themselves with low-level computational details of their implementation. The expressiveness, power and ease-of-use of the simulator interface is critical in efficiently and accurately translating ideas into a working simulation. We review long-term trends in the development of programmable simulator interfaces, and examine the benefits of moving from proprietary, domain-specific languages to modern dynamic general-purpose languages, in particular Python, which provide neuroscientists with an interactive and expressive simulation development environment and easy access to state-of-the-art general-purpose tools for scientific computing.

Keywords: Python, simulation, computational neuroscience

Edited by:
Rolf Kötter, Radboud University Nijmegen, The Netherlands

Reviewed by:
Felix Schürmann, Ecole Polytechnique Fédérale de Lausanne, Switzerland
Marc-Oliver Gewaltig, Honda Research Institute Europe GmbH, Germany
Volker Steuber, University of Hertfordshire, UK

**** Correspondence:***

***Andrew P. Davison** is a research scientist in the Unité de Neurosciences Intégratives et Computationnelles of the Centre National de la Recherche Scientifique, France. He is interested in large-scale, data-constrained, biologically-detailed modeling of neuronal networks; in bridging the gaps between different levels of modeling; and in the development of tools to facilitate collaborative modeling and reproducible simulations. He is co-founder of the NeuralEnsemble initiative (http://neuralensemble.org), which aims to promote collaborative software development in neuroscience. andrew.davison@unic.cnrs-gif.fr*

INTRODUCTION

Many models in computational neuroscience can be expressed by equations that have exact mathematical solutions, but a far greater number cannot, and approximate solutions must be found using numerical methods, a technique commonly referred to as simulation.

The earliest neuroscience simulations were perhaps those of Andrew Huxley, using a Brunsviga mechanical calculator (Hodgkin, 1976), for his Nobel prize-winning work with Alan Hodgkin on the action potential (Hodgkin and Huxley, 1952). Huxley performed his calculations by hand only because the Cambridge University electronic computer was undergoing an upgrade, but soon enough simulations of nerve cell membranes were being performed by computers, with programs written in languages such as FOCAL and FORTRAN (Moore, 1994).

The programs for such early simulations were designed from the ground up, with investigators concerning themselves with the technical details of early computers, the biological details of the system under study, and with devising efficient algorithms for numerically solving the differential equations (**Figure 1A**). With the maturation of the field, however, the details of the numerical methods began to be standardized, and an increase in research productivity could be achieved by hiding the details of solving the equations from the investigator, allowing them to focus on the biological concepts. Thus a number of general-purpose neuroscience simulation programs ("**simulators**") began to appear in the late 1980s and early 1990s (**Figure 1B**), such as CABLE (the forerunner of NEURON; Hines, 1989), GENESIS (Wilson et al., 1989), NODUS (De Schutter, 1989), Axontree (Manor et al., 1991), Nemosys (Eeckman et al., 1993) and SWIM (Ekeberg et al., 1994) (see Moore, 1994 for much more detail on this era).

The major advantages of using simulation software rather than writing simulation programs from scratch are: increased productivity, since

Simulator
A simulator is a computer program that numerically solves the equations used to represent a particular model, resulting in a simulation of the system being modelled.

Interpreted language
An interpreted language is one in which each line of source code is translated into machine code just before it is executed. In contrast, in a compiled language all the source code is translated at once, to be executed later. Interpreted languages allow interactive programming, i.e., a user can type a line of code, execute it, and then decide what to do next based on the results of that line.

Domain-specific language
In contrast to a general-purpose programming language, a domain-specific language is a language dedicated to a particular problem domain, which allows concepts from that domain to be expressed more clearly than with a general-purpose language. Examples outside neuroscience include spreadsheet formulas, the PostScript language for page rendering, and the typesetting language TEX.

NEURON simulation environment
NEURON is a simulator for modelling individual neurons and networks of neurons. It provides tools for conveniently building, managing, and using models in a way that is numerically sound and computationally efficient. It is particularly well-suited to problems that are closely linked to experimental data, especially those that involve cells with complex anatomical and biophysical properties.

much less code has to be written; fewer bugs, since the simulator will be used by many people, not just one or two, and hence bugs are encountered and fixed earlier; improved conceptual control of the simulation, since low-level computation and book-keeping are done by the simulator, allowing the user to focus on the scientific concepts.

There are several ways in which models and their environment (inputs, parameterization, instrumentation, output files, etc.) can be specified in a simulator: through a text-based configuration file, through a graphical interface, or through a special-purpose, domain-specific programming language, either compiled or, more commonly, **interpreted**. The advantages of configuration files or graphical interfaces are that the user need not have any knowledge of programming, and that it is much more difficult, or impossible, to introduce an error or inconsistency into the model (though, of course, the model defined by the user may differ from his or her intention). This is particularly important when using a simulator as an educational tool. For research, however, the greater flexibility introduced by a domain-specific programming language is often indispensable.

PROGRAMMING LANGUAGES FOR NEURAL SIMULATORS

Until very recently, with few exceptions, each simulator that offered the option of a **domain-specific programming language** (DSL) came with its own proprietary language, specific to that simulator.

These languages often started out restricted in scope, then gradually added functionality as the software was developed (Cannon et al., 2007). At a minimum, a DSL for a neural simulator needs to be able to represent neuroscience concepts, such as ion channels, synapses, dendrites, neurons, and to interact with the operating system by reading and writing files and accepting user input. Beyond this minimum, the following features are desirable: features for structured programming, at least functions/procedures and preferably classes and objects; a variety of data structures such as lists, associative arrays, matrices; a mathematical library; a graphical interface.

To illustrate this trend of gradual accumulation of features, consider the interpreter for the **NEURON simulation environment**. Hoc (Kernighan and Pike, 1984) was incrementally developed by those authors within the context of a tutorial on "Program Development" using standard UNIX software tools. As a language development example, Hoc had a syntax for expressions and control flow vaguely similar to the C language and we considered it more expressive than the BASIC-like interpreter, FOCAL, previously used in our lab for interactively calling functions and assigning/evaluating variables in C or FORTRAN compiled libraries. In 1984, the precursor to the NEURON simulation environment, CABLE, switched from FOCAL to Hoc for setup and control of neural simulations. A fundamental extension to Hoc syntax was made in the late 1980s in order to represent the notion of continuous cables, called sections. Sections are connected to form a tree shaped structure and their principle purpose is to allow the user to specify the physical properties of a neuron without regard for the purely numerical issue of how many compartments are used to represent each of the cable sections. In the early 1990s, Hoc syntax was again extended to provide some limited support for classes and objects, that is, data encapsulation and polymorphism, but not inheritance; useful containers for numerical data, such as vectors and matrices, were added and a graphical interface was developed.

Hoc is now a fairly full-featured, general programming language that serves its purpose well. However, it has turned out to be an orphan language limited to NEURON users and, along with all other DSLs for neural simulators, inevitably suffers in comparison with mainstream, general-purpose interpreted languages such as Python (http://www.python.org), Ruby (http://www.ruby-lang.org/) or Scheme (Abelson et al., 1998), or with general scientific programming environments such as MATLAB (The Mathworks, Inc.), which have hundreds of developers and many thousands of users in all domains of science and engineering. Furthermore, continuing development and maintenance of the general programming language features of a DSL steals significant time and effort from neurobiology domain-specific improvements.

Given these limitations and costs of domain-specific languages for simulators, the natural next step in the evolution of programming languages for neural simulators is to replace home-grown DSLs with a general purpose programming language, with neurobiology-specific concepts implemented in the general purpose language (**Figure 1C**). This relieves the simulator developer of the need to develop and maintain standard programming language features, connects both developers and users to a wider scientific and technical programming community, and in most cases enables an enhanced representation of domain-specific concepts, since modern, widely-used languages are almost inevitably more powerful and expressive than home-grown DSLs.

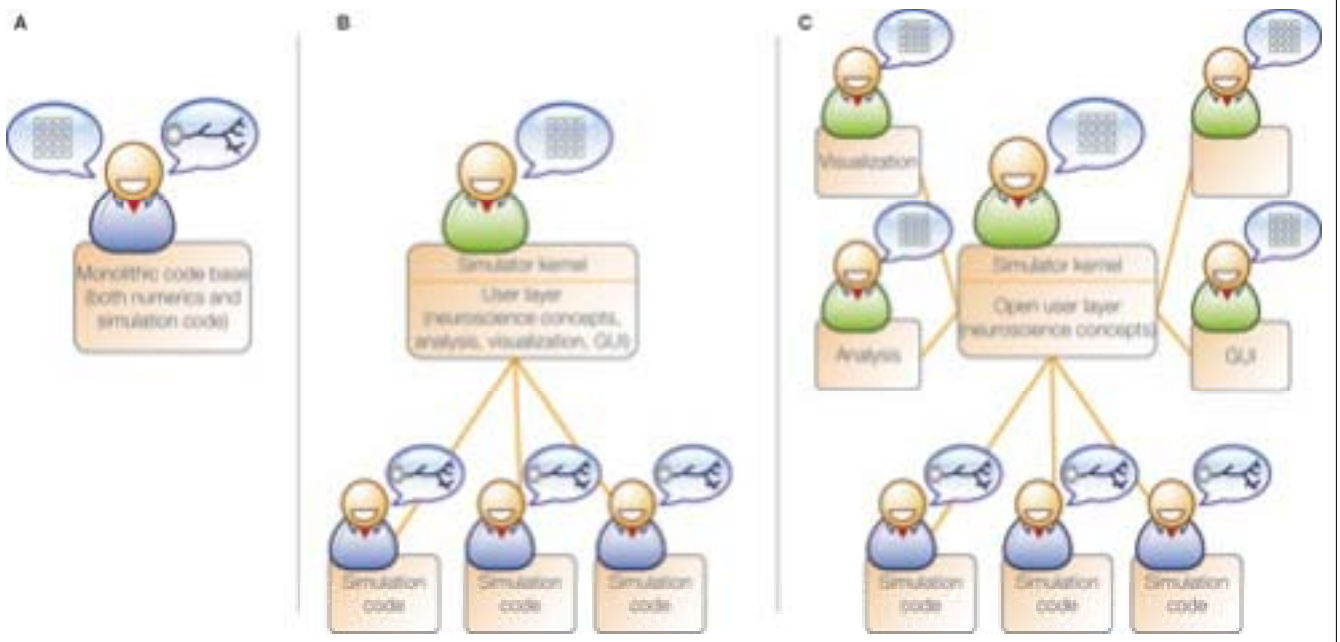

FIGURE 1 | (A) The earliest approach to computational simulations: the user (neuroscientist) is also the software developer, and simulation code is written from the ground up. **(B)** Development of general-purpose neural simulators. The simulator is developed by a small group of people, and has its own domain-specific language (DSL) for representing neuroscience concepts. The DSL also contains functionality for visualizing results and performing numerical analysis. The simulator is used by a larger group of scientists, who may not be programming experts, and who use the DSL to create their simulations. **(C)** The DSL is replaced or augmented by a general-purpose interpreted programming language. The simulator developers are able to concentrate on the domain-specific functionality, and to leverage tools for visualization, analysis, database access, network access, etc., developed by third-party developers external to neuroscience. The users benefit from greatly expanded functionality and easier extensibility of the simulator. *The programmer icon was originally created by David Vignoni, and is reused here under the terms of the GNU Lesser General Public Licence (LGPL) (http://www.gnu.org/licenses/lgpl.html). As a consequence, this image is also licensed under the LGPL. Copies of the image file are available on request from the corresponding author.*

Such a general purpose language should: (i) be interpreted, allowing interactive use of the simulator; (ii) be easy to learn, given that most of its users will be neuroscientists with little formal computer science training or experience; (iii) provide support for modularity, facilitating the construction and maintenance of complex programs; (iv) have a large scientific/engineering user base (not restricted to neuroscience), providing a ready-made library of tools for data analysis and visualization; (v) have a large general user base outside of science, providing general purpose tools for database access, graphical interfaces, network access, debugging, etc.

A number of languages, among them Python, Perl (http://www.perl.org), Ruby and Scheme, meet these criteria and would be suitable candidates for a simulator interface language. For a number of reasons, it is the Python programming language that has seen widespread uptake among simulator developers in recent years, resulting in the addition of Python interfaces to several existing simulators, including NEURON (Hines et al., 2009), NEST (Eppler et al., 2008), Nengo (Stewart et al., 2009), MOOSE (Ray and Bhalla, 2008), STEPS (Wils and De Schutter, 2009), Topographica (Bednar, 2009) and NCS (Drewes et al., 2009), and in the choice of Python as the sole or principal interface language for new simulators such as Brian (Goodman and Brette, 2008) and PCSIM (Pecevski et al., 2009).

The latter examples demonstrate that new simulators need not inevitably reprise the path described above, and nowadays can adopt a general-purpose language from the beginning. It is also possible to replace one general-purpose language with another, as was done in the rewrite of CSIM (which originally used MATLAB) to produce PCSIM, and by Topographica, which initially adopted Scheme before replacing it with Python.

PYTHON AS A PROGRAMMING LANGUAGE FOR NEURAL SIMULATION

Python is a dynamic object-oriented programming language that is widely used in both commercial and academic settings for systems integration, as a scripting language, as a web-development language, and for scientific computing.

SciPy
SciPy is perhaps the most important of many open-source packages for scientific computing that use the Python programming language. It is an excellent example of the sort of powerful tool that becomes available to simulator users when the simulator interface is a general-purpose programming language such as Python.

The advantages of using Python in the context of neuronal simulations are:

- it is an interpreted language, making interactive exploration of code or data possible, and providing immediate feedback to the user.
- clear, expressive syntax. This makes code easy to write and, perhaps more importantly, easy to read, facilitating sharing, debugging and re-use. Python code is generally concise enough to make it easy to see and understand the overall structure, but not so concise as to be confusing.
- powerful data structures such as lists and dictionaries are built-in to the language.
- it has an extremely flexible implementation of object-oriented programming, which is important in producing well-structured, reusable code, making it possible to create models of high complexity (almost inevitable in neuroscience) with non-complicated code.
- it has a large standard library, providing built-in, extensive functionality for data processing, database access, network programming, etc.
- a large number of freely-available, third-party libraries for graphical interfaces, scientific computing, etc., are available. Of particular note is the **SciPy** package (http://www.scipy.org), which provides extensive and high-speed facilities for manipulation of numerical data.
- it is easy to interface Python with code written in other programming languages. A common approach is to develop a user interface in Python, with its ease-of-use and rapid development time, and to implement computationally-expensive code in a fast, compiled language such as C, C++ or FORTRAN.
- it is easy to learn (Raymond, 2000), due mainly to the first three points in this list.

Although many of the above are also true of other programming languages, taken together Python seems to have the best combination. There is also a virtuous circle effect: as Python is more widely used in scientific computing, the range of available libraries, of teaching materials, and of expertise becomes wider, making it yet more attractive.

THE PYTHON INTERFACE TO NEURON

To give a concrete example of the use of Python in a neuroscience simulator, we present here the Python interface to NEURON, which coexists, and interoperates with, the original Hoc interpreter. NEURON with Python works on Windows, Mac OS X, Linux, and many other platforms such as the IBM Blue Gene/L/P and Cray XT3 supercomputers. Downloads and installation instructions can be found at http://www.neuron.yale.edu.

The fundamental objects for representing neurons in NEURON are the membrane section (an un-branched piece of a dendrite, axon or soma), and the membrane mechanism, which may either be inserted at a particular point in a section, as for synapses or electrodes, or distributed over the entire surface of the section, as for ion channels.

Each of these objects is represented by a Python class. In the following listing we create a section for the soma, a section for a dendrite, connect them together, insert Hodgkin-Huxley sodium and potassium channels in the soma, and place a synapse near the end of the dendrite:

```
>>> from neuron import h
>>> soma = h.Section()
>>> dend = h.Section()
>>> dend.connect(soma, 0, 0)
>>> soma.insert('hh')
>>> syn = h.ExpSyn(0.9, sec=dend)
```

This code is not very far removed from the code used to perform the same task in Hoc, the main difference being that Hoc has special keywords `create`, `connect` and `insert`, whereas the Python interface uses the standard syntax for creating a new object and calling "methods" (object-oriented programming terminology for functions that are bound to an individual object).

The gain in representing membrane sections as standard objects, rather than using special keywords to create and manipulate them, is that the full power of Python's object-oriented approach can be brought to bear, allowing sub-classes to inherit behaviour from their parent classes, encapsulation of data and functionality within the class, and allowing sections to be passed as arguments to functions, all of which lead to cleaner, easier-to-understand code. Hoc does have object-oriented capabilities, but they do not apply to sections, and do not support inheritance.

While the syntax improvements are valuable, a much greater benefit of moving from a special-purpose to a widely-used, general-purpose language is the availability of all the libraries and modules developed in the general-purpose language. This is described in the next section.

Despite these gains, the move away from the special-purpose language might nevertheless be negative overall if some of the capabilities of

the special-purpose language were lost, or if the expertise built-up by modellers in that language were no longer applicable. This is not the case for NEURON, since all the functionality of Hoc is still available through a special object representing the Hoc interpreter (`h` in the code example above). The `h` object allows us to use Hoc commands such as `create`, e.g.:

```
>>> h('create soma')
>>> h.soma
<nrn.Section object at 0x8194080>
```

and makes any of the classes defined in Hoc available to Python, such as the `ExpSyn` mechanism in the example above, or the important `Vector` class, which is used for recording, graphing and many other purposes. Through Python, Hoc `Vector` objects can be used in most cases where Numpy, Scipy, and Matplotlib (Hunter, 2007), the most important scientific modules, accept lists or arrays:

```
>>> from numpy import array
>>> l1 = [1, 2, 3, 4, 5]
>>> a1 = array(l1)
>>> v1 = h.Vector(l1)
>>> v2 = h.Vector(a1)
>>> a2 = array(v1)
>>> l2 = list(v2)
>>> from matplotlib.pylab import plot
>>> plot(v1)
```

This easy interoperability between Hoc and Python makes it easy to re-use existing code written in Hoc in a new simulation using Python, without needing to rewrite or convert the older code.

This has been a very brief introduction to using Python with NEURON. A more extensive description is given in Hines et al. (2009). Although the details differ, the general benefits described above apply equally to other simulators that use a general-purpose language, such as Python, as their user interface.

CURRENT AND FUTURE USES OF PYTHON

Python makes a vast library of third-party modules available to NEURON users for use in their simulations. We give here two examples: exporting data and importing XML. Suppose the user would like to export simulated voltage traces to a standard binary format for scientific data, such as HDF5 (http://www.hdfgroup.org/HDF5), for later analysis in Python, MATLAB, etc. The PyTables package (www.pytables.org) provides the required functionality. Supposing the membrane potential trace is stored in a vector `vm`, the following code saves it to HDF5:

```
>>> import tables, numpy
>>> h5 = tables.openFile('test.h5',
...                      'w')
>>> h5.createArray('/', 'V',
...                numpy.array(vm))
>>> h5.close()
```

MorphML (Crook et al., 2007) is an XML-based format for exchanging neuronal morphology data. An increasing number of neuron reconstructions are available in this format, and it was desirable to allow these morphologies to be imported into NEURON. Hoc does not provide any tools for processing XML data (it would be possible, but time-consuming to create them), but Python provides a number of such tools. By using Hoc and Python together, the process of adding MorphML support was greatly accelerated. Taking lines of code as a crude measure of development effort, we can compare the 1180 lines of NEURON's NeuroLucida v3 import tool, written purely in Hoc, to the 448 lines (78 lines of Hoc, 370 lines of Python) needed for MorphML import. A fuller description of the MorphML import is given in Hines et al. (2009).

The availability of Python interfaces for multiple simulators allows two or more simulators to be coupled via the interpreter to compose compound models, as explored in Ray and Bhalla (2008). With the addition of run-time simulator interaction, such as provided by the MUSIC library (Ekeberg and Djurfeldt, 2008), such interactions become possible in distributed computing environments on a large-scale while remaining controllable from a single interactive Python prompt (in the distributed case using the parallel capabilities of IPython). Moreover, it becomes possible to provide a unified meta-interface to Python-based simulators. PyNN (Davison et al., 2009) is one such meta-interface, and allows network models of point neurons (integrate-and-fire, single compartment Hodgkin-Huxley, etc.) to be simulated on NEURON, NEST, Brian and PCSIM without any modification of the code. Such a common interface facilitates model cross-checking, translation, evaluation of the optimal simulator for a given problem, and provides a simulator-agnostic foundation upon which to develop higher-level modelling abstractions.

In our opinion, the most promising future applications of Python in neuroscience simulation include the following:

- more expressive, well-structured and easy-to-understand models, as expanded on in previous sections;
- development of graphical user interfaces using the power of the most recent cross-platform GUI toolboxes such as Qt or GTK+, either replacing or complementing existing GUIs. The possibility of creating a single GUI for multiple simulators (like the existing neuroConstruct, Gleeson et al., 2007, but fully interactive) is also very interesting;
- integration of the simulator in a complete Python-based workflow for simulation projects, including stimulus generation, visualization, data analysis and databasing;
- support for declarative formats for neuroscience models (based on XML or other formats), such as NeuroML (Goddard et al., 2001, http://www.neuroml.org) or SBML (Hucka et al., 2003).

DISCUSSION

> "About half the time spent on a typical simulation project involves creating and tuning the model. Thus, a good user interface may contribute more to the overall efficiency of a project than pure computation speed."
>
> De Schutter (1992)

> "Increasingly, the real limit on what computational scientists can accomplish is how quickly and reliably they can translate their ideas into working code."
>
> Wilson (2006)

Available, affordable computer power and the amount of experimental data available to constrain models have both increased greatly in the 14 years that separate these two quotations. So, however, have the ambition of computational neuroscientists and the complexity of the simulations they develop, so that the influence of the simulator user interface on the efficiency and correctness of a neuronal simulation project is greater than ever.

In this article, we have coarsely sketched the history of simulator interface development, and have highlighted the most recent trend: for homegrown interfaces to be replaced by modern, powerful, general purpose programming languages, particularly Python.

This trend has a number of positive consequences:

- it frees up simulator developers to concentrate on neuroscience-specific features, leaving ancillary functionality to the general language.
- it makes available to both developers and users an extensive collection of tools for data analysis, visualization, debugging, testing, etc.
- it provides tools for well-structured programming, so that simulating complex models of complex neural structures need not imply complex, hard-to-understand code.
- competency gained in programming a simulator is transferrable to other domains of programming, both inside and outside science.
- where multiple simulators adopt the same general purpose language, as is the case with Python, the energy barrier for translating models between simulators is lowered. Each simulator still effectively has its own representations for neuroscience domain-specific concepts (but see Davison et al., 2009), but now all simulators can access the same data structures, and exploit the same built-in and external libraries. Furthermore, it becomes much easier to develop tools, for visualization or data analysis, that will work with any Python-supporting simulator.

From the point of view of the NEURON simulator, we recommend that new users of NEURON and those already familiar with Python should use Python rather than Hoc to develop new models. There is no need to rewrite legacy code in Python, as it will continue to work using the Hoc interpreter or mixed in with new Python code and accessed via the `h` object.

Our expectation is that the recent widespread adoption of Python for simulator interfaces will lead to accelerated progress in computational neuroscience. Although part of the complexity of neuroscience models comes from the unavoidable complexity of the neural systems under study, the extra complexity added by our software systems can certainly be reduced. Python alone is not a silver bullet that completely alleviates the problem of avoidable complexity in neuroscience modelling – simulation-based computational neuroscience must at the least also adopt other tools from mainstream software engineering – but it provides a solid foundation for developing readable, modular, well-structured, reusable models; developing and sharing tools for simulation project management, data analysis and visualisation, etc.; and leveraging the work of other scientific and engineering communities, without which we cannot hope to begin to tame the complexity of the brain.

ACKNOWLEDGMENTS

This work was supported by NIH grant NS11613, by the European Union under the Bio-inspired Intelligent Information Systems program, project reference IST-2004-15879 (FACETS), and by a grant from the Swiss National Science Foundation. We would also like to thank the reviewers, whose comments have been very helpful in improving this article.

REFERENCES

Abelson, H., Dybvig, R., Haynes, C., Rozas, G., Adams, N., Friedman, D., Kohlbecker, E., Steele, G., Bartley, D., Halstead, R., Oxley, D., Sussman, G., Brooks, G., Hanson, C., Pitman, K., and Wand, M. (1998). Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computat.* 11, 7–105. doi:10.1023/A:1010051815785.

Bednar, J. A. (2009). Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Front. Neuroinformatics* 3, 8. doi: 10.3389/neuro.11.008.2009.

Cannon, R. C., Gewaltig, M. O., Gleeson, P., Bhalla, U. S., Cornelis, H., Hines, M. L., Howell, F. W., Muller, E., Stiles, J. R., Wils, S., and De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.

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

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinformatics* 2, 11. doi: 10.3389/neuro.11.011.2008.

De Schutter, E. (1992). A consumer guide to neuronal modeling software. *Trends Neurosci.* 15, 462–464.

De Schutter, E. (1989). Computer software for development and simulation of compartmental models of neurons. *Comput. Biol. Med.* 19, 71–81.

Drewes, R., Zou, Q., and Goodman, P. H. (2009). Brainlab: a Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the NeoCortical Simulator. *Front. Neuroinformatics* 3, 16. doi: 10.3389/neuro.11.016.2009.

Eeckman, F. H., Theunissen, F. E., and Miller, J. P. (1993). NeMoSys: A neural modeling system. In: MASCOTS '93, Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems. San Diego, CA, USA, The Society for Computer Simulation, International, pp. 365–366.

Ekeberg, O. and Djurfeldt, M. (2008). MUSIC – multisimulation coordinator: request for comments. *Nat. Precedings*. doi: http://dx.doi.org/10.1038/npre.2008.1830.1.

Ekeberg, O., Hammarlund, P., Levin, B., and Lansner, A. (1994). SWIM – a simulation environment for realistic neural network modeling. In Neural Network Simulation Environments, J. Skrzypek, ed. (Hingham, MA, Kluwer), pp. 47–71.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. O. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinformatics* 2, 12. doi: 10.3389/neuro.11.012.2008.

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, N., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modeling in neuroscience. *Philoso. Trans. R. Soc. B* 356, 1209–1228.

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinformatics* 2, 5. doi: 10.3389/neuro.11.005.2008.

Hines, M. (1989). A program for simulation of nerve equations with branching geometries. *Int. J. Biomed. Comput.* 24, 55–68.

Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinformatics* 3, 1. doi: 10.3389/neuro.11.001.2009.

Hodgkin, A. L. (1976). Chance and design in electrophysiology: an informal account of certain experiments on nerve carried out between 1934 and 1952. *J. Physiol.* 263, 1–21.

Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117, 500–544.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S., Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hedley, W. J., Hodgman, T. C., Hofmeyr, J. H., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Le Novère, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsness, E. D., Nakayama, Y., Nelson, M. R., Nielsen, P. F., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., and Wang, J. (2003). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531.

Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *IEEE Comput. Sci. Eng.* 9, 90–95.

Kernighan, B., and Pike, R. (1984). The Unix Programming Environment. Englewood Cliffs, New Jersey, Prentice Hall.

Manor, Y., Goncazarowski, J., and Segev, I. (1991). Propagation of action potentials along complex axonal trees. Model and implementation. *Biophys. J.* 60, 1411–1423. doi: 10.1016/S0006-3495(91)82178-6.

Moore, J. W. (1994). Simulations with NEURON. Available at: http://neuron.duke.edu/userman/contents.html.

Pecevski, D., Natschläger, T., and Schuch, K. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Front. Neuroinformatics* 3, 11. doi: 10.3389/neuro.11.011.2009.

Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinformatics* 2, 6. doi: 10.3389/neuro.11.006.2008.

Raymond, E. S. (2000). Why Python? *Linux J.* Available at: www.linuxjournal.com/article/3882.

Stewart, T. C., Tripp, B., and Eliasmith, C. (2009). Python scripting in the Nengo simulator. *Front. Neuroinformatics* 3, 7. doi: 10.3389/neuro.11.007.2009.

Wils, S., and De Schutter, E. (2009). STEPS: modeling and simulating complex reaction-diffusion systems with Python. *Front. Neuroinformatics* 3, 15. doi: 10.3389/neuro.11.015.2009.

Wilson, G. (2006). Where's the real bottleneck in scientific computing? *Am. Sci.* 94, 5. doi: 10.1511/2006.1.5.

Wilson, M. A., Bhalla, U. S., Uhley, J. D., and Bower, J. M. (1989). GENESIS: a system for simulating neural networks. In Advances in Neural Information Processing Systems, D. Touretzky, ed. (San Mateo, CA, Morgan Kaufmann), pp. 485–492.

Conflict of Interest Statement: The authors declare that the research presented in this paper was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 31 July 2009; paper pending published: 04 September 2009; accepted: 02 October 2009; published: 15 December 2009.

*Citation: Front. Neurosci. (2009) **3**, 3: 374–380. doi: 10.3389/neuro.01.036.2009*