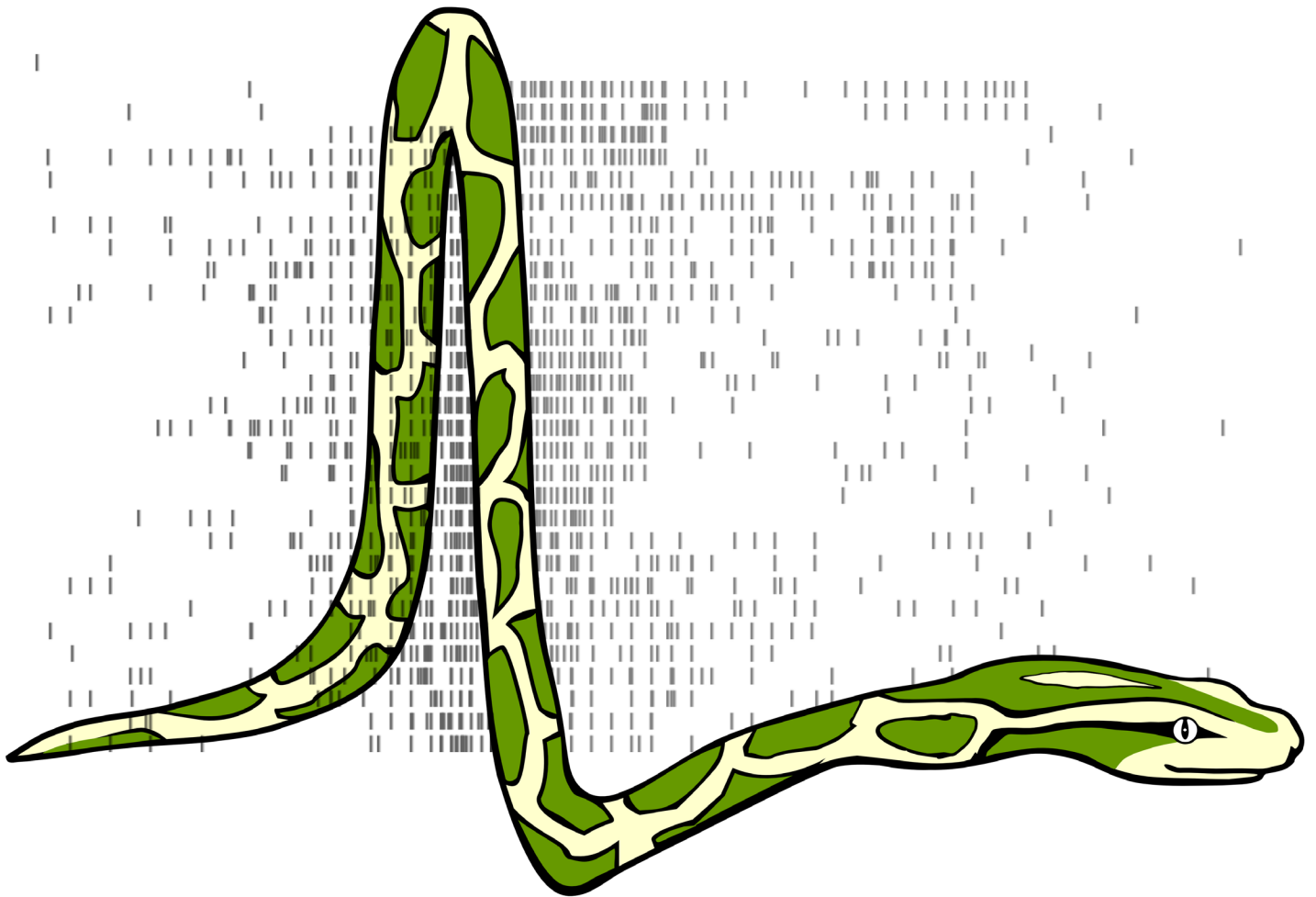


PYTHON IN NEUROSCIENCE

EDITED BY : Eilif Muller, James A. Bednar, Markus Diesmann,
Marc-Oliver Gewaltig, Michael Hines and Andrew P. Davison
PUBLISHED IN: Frontiers in Neuroinformatics





frontiers

Frontiers Copyright Statement

© Copyright 2007-2015 Frontiers Media SA. All rights reserved.

All content included on this site, such as text, graphics, logos, button icons, images, video/audio clips, downloads, data compilations and software, is the property of or is licensed to Frontiers Media SA ("Frontiers") or its licensees and/or subcontractors. The copyright in the text of individual articles is the property of their respective authors, subject to a license granted to Frontiers.

The compilation of articles constituting this e-book, wherever published, as well as the compilation of all other content on this site, is the exclusive property of Frontiers. For the conditions for downloading and copying of e-books from Frontiers' website, please see the Terms for Website Use. If purchasing Frontiers e-books from other websites or sources, the conditions of the website concerned apply.

Images and graphics not forming part of user-contributed materials may not be downloaded or copied without permission.

Individual articles may be downloaded and reproduced in accordance with the principles of the CC-BY licence subject to any copyright or other notices. They may not be re-sold as an e-book.

As author or other contributor you grant a CC-BY licence to others to reproduce your articles, including any graphics and third-party materials supplied by you, in accordance with the Conditions for Website Use and subject to any copyright notices which you include in connection with your articles and materials.

All copyright, and all rights therein, are protected by national and international copyright laws.

The above represents a summary only. For the full conditions see the Conditions for Authors and the Conditions for Website Use.

ISSN 1664-8714

ISBN 978-2-88919-608-1

DOI 10.3389/978-2-88919-608-1

About Frontiers

Frontiers is more than just an open-access publisher of scholarly articles: it is a pioneering approach to the world of academia, radically improving the way scholarly research is managed. The grand vision of Frontiers is a world where all people have an equal opportunity to seek, share and generate knowledge. Frontiers provides immediate and permanent online open access to all its publications, but this alone is not enough to realize our grand goals.

Frontiers Journal Series

The Frontiers Journal Series is a multi-tier and interdisciplinary set of open-access, online journals, promising a paradigm shift from the current review, selection and dissemination processes in academic publishing. All Frontiers journals are driven by researchers for researchers; therefore, they constitute a service to the scholarly community. At the same time, the Frontiers Journal Series operates on a revolutionary invention, the tiered publishing system, initially addressing specific communities of scholars, and gradually climbing up to broader public understanding, thus serving the interests of the lay society, too.

Dedication to Quality

Each Frontiers article is a landmark of the highest quality, thanks to genuinely collaborative interactions between authors and review editors, who include some of the world's best academicians. Research must be certified by peers before entering a stream of knowledge that may eventually reach the public - and shape society; therefore, Frontiers only applies the most rigorous and unbiased reviews.

Frontiers revolutionizes research publishing by freely delivering the most outstanding research, evaluated with no bias from both the academic and social point of view.

By applying the most advanced information technologies, Frontiers is catapulting scholarly publishing into a new generation.

What are Frontiers Research Topics?

Frontiers Research Topics are very popular trademarks of the Frontiers Journals Series: they are collections of at least ten articles, all centered on a particular subject. With their unique mix of varied contributions from Original Research to Review Articles, Frontiers Research Topics unify the most influential researchers, the latest key findings and historical advances in a hot research area! Find out more on how to host your own Frontiers Research Topic or contribute to one as an author by contacting the Frontiers Editorial Office: researchtopics@frontiersin.org

PYTHON IN NEUROSCIENCE

Topic Editors:

Eilif Muller, Ecole Polytechnique Fédérale de Lausanne, Switzerland

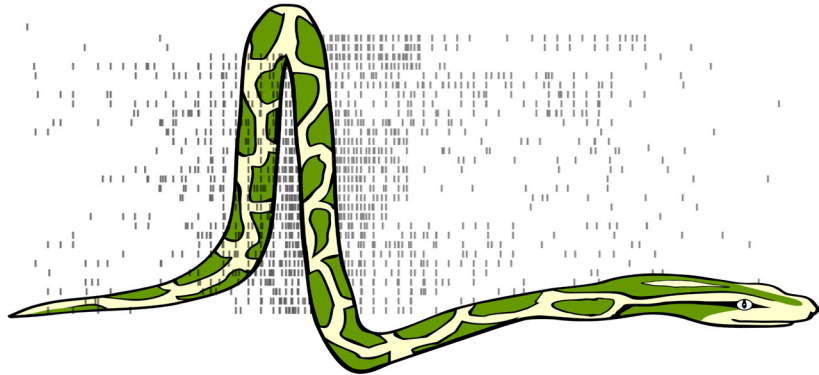
James A. Bednar, University of Edinburgh, UK

Markus Diesmann, Jülich Research Center and Jülich Aachen Research Alliance, Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6), Jülich, Germany; RWTH Aachen University, Germany

Marc-Oliver Gewaltig, Ecole Polytechnique Fédérale de Lausanne, Switzerland

Michael Hines, Yale University, USA

Andrew P. Davison, Centre National de la Recherche Scientifique, Gif sur Yvette, France



“Python action potential” by Yaroslav Halchenko and Andrew P. Davison. Licenced under the Creative Commons Attribution-Share Alike (CC BY-SA) 3.0 licence. Partially based on “Snake Brain” by Arno Klein and Michael Hanke.

Python is rapidly becoming the de facto standard language for systems integration. Python has a large user and developer-base external to the neuroscience community, and a vast module library that facilitates rapid and maintainable development of complex and intricate systems.

In this Research Topic, we highlight recent efforts to develop Python modules for the domain of neuroscience software and neuroinformatics:

- simulators and simulator interfaces
- data collection and analysis
- sharing, re-use, storage and databasing of models and data

- stimulus generation
- parameter search and optimization
- visualization
- VLSI hardware interfacing

Moreover, we seek to provide a representative overview of existing mature Python modules for neuroscience and neuroinformatics, to demonstrate a critical mass and show that Python is an appropriate choice of interpreter interface for future neuroscience software development.

Citation: Muller, E., Bednar, J. A., Diesmann, M., Gewaltig, M.-O., Hines, M., Davison, A. P., eds. (2015). Python in Neuroscience. Lausanne: Frontiers Media.
doi: 10.3389/978-2-88919-608-1

Table of Contents

- 06 *Python in neuroscience***
Eilif Muller, James A. Bednar, Markus Diesmann, Marc-Oliver Gewaltig, Michael Hines and Andrew P. Davison
- 10 *STEPS: modeling and simulating complex reaction-diffusion systems with Python***
Stefan Wils and Erik De Schutter
- 18 *Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system***
Daniel Brüderle, Eric Müller, Andrew Davison, Eilif Muller, Johannes Schemmel and Karlheinz Meier
- 28 *Near-infrared neuroimaging with NinPy***
Gary E. Strangman, Quan Zhang and Thomas Zeffiro
- 41 *Network features and pathway analyses of a signal transduction cascade***
Ryoji Yanashima, Noriyuki Kitagawa, Yoshiya Matsubara, Robert Weatheritt, Kotaro Oka, Shinichi Kikuchi, Masaru Tomita and Shun Ishizaki
- 51 *Brainlab: a Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the NeoCortical Simulator***
Rich Drewes, Quan Zou and Philip H. Goodman
- 61 *PCSIM: a parallel simulation environment for neural circuits fully integrated with Python***
Dejan Pecevski, Thomas Natschläger and Klaus Schuch
- 76 *OpenElectrophy: an electrophysiological data- and analysis-sharing framework***
Samuel Garcia and Nicolas Fourcaud-Trocmé
- 86 *DataViewer3D: an open-source, cross-platform multi-modal neuroimaging data visualization tool***
André Gouws, Will Woods, Rebecca Millman, Antony Morland and Gary Green
- 104 *Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components***
James A. Bednar
- 113 *Python scripting in the Nengo simulator***
Terrence C. Stewart, Bryan Tripp and Chris Eliasmith
- 122 *Technical integration of hippocampus, basal ganglia and physical models for spatial navigation***
Charles Fox, Mark Humphries, Ben Mitchinson, Tamas Kiss, Zoltan Somogyvari and Tony Prescott

- 133 *Python for information theoretic analysis of neural data***
Robin A. A. Ince, Rasmus S. Petersen, Daniel C. Swan and Stefano Panzeri
- 148 *OMPC: an open-source MATLAB®-to-Python compiler***
Peter Jurica and Cees van Leeuwen
- 157 *PyMVPA: a unifying approach to the analysis of neuroscientific data***
Michael Hanke, Yaroslav O. Halchenko, Per B. Sederberg, Emanuele Olivetti, Ingo Fründ, Jochem W. Rieger, Christoph S. Herrmann, James V. Haxby, Stephen José Hanson and Stefan Pollmann
- 170 *PyNEST: A convenient interface to the NEST simulator***
Jochen Martin Eppler, Moritz Helias, Eilif Muller, Markus Diesmann and Marc-Oliver Gewaltig
- 182 *NEURON and Python***
Michael L. Hines, Andrew P. Davison and Eilif Muller
- 194 *Python for large-scale electrophysiology***
Martin Spacek, Tim Blanche and Nicholas Swindale
- 204 *PyNN: a common interface for neuronal network simulators***
Andrew P. Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet and Pierre Yger
- 214 *Generating stimuli for neuroscience using PsychoPy***
Jonathan W. Peirce
- 222 *Modular toolkit for Data Processing (MDP): a Python data processing framework***
Tiziano Zito, Niko Wilbert, Laurenz Wiskott and Pietro Berkes
- 229 *PyMOOSE: interoperable scripting in Python for MOOSE***
Subhasis Ray and Upinder S. Bhalla
- 245 *A Python analytical pipeline to identify prohormone precursors and predict prohormone cleavage sites***
Bruce R. Southey, Jonathan V. Sweedler and Sandra L. Rodriguez-Zas
- 254 *Brian: a simulator for spiking neural networks in Python***
Dan Goodman and Romain Brette
- 264 *Vision Egg: an open-source library for realtime visual stimulus generation***
Andrew D. Straw

Python in neuroscience

Elif Muller¹, James A. Bednar², Markus Diesmann^{3,4,5}, Marc-Oliver Gewaltig¹, Michael Hines⁶ and Andrew P. Davison^{7*}

¹ Center for Brain Simulation, Ecole Polytechnique Fédérale de Lausanne, Geneva, Switzerland, ² Institute for Adaptive and Neural Computation, University of Edinburgh, Edinburgh, UK, ³ Jülich Research Center and Jülich Aachen Research Alliance, Institute of Neuroscience and Medicine (INM-6) and Institute for Advanced Simulation (IAS-6), Jülich, Germany, ⁴ Department of Psychiatry, Psychotherapy and Psychosomatics, Medical Faculty, RWTH Aachen University, Aachen, Germany, ⁵ Department of Physics, Faculty 1, RWTH Aachen University, Aachen, Germany, ⁶ Department of Neurobiology, Yale University, New Haven, CT, USA, ⁷ Neuroinformatics group Unité de Neurosciences, Information et Complexité, Centre National de la Recherche Scientifique, Gif sur Yvette, France

Keywords: python language, software development, scientific computing, interoperability, collaboration

This Research Topic of Frontiers in Neuroinformatics is dedicated to the memory of Rolf Kötter (1961–2010), who was the Frontiers Associate Editor responsible for this Research Topic, and who gave us considerable support and encouragement during the process of conceiving and launching the Topic, and throughout the reviewing process.

Computation is becoming essential across all sciences, for data acquisition and analysis, automation, and hypothesis testing via modeling and simulation. As a consequence, software development is becoming a critical scientific activity. Training of scientists in programming, software development, and computational thinking (Wilson, 2006), choice of tools, community-building and interoperability are all issues that should be addressed, if we wish to accelerate scientific progress while maintaining standards of correctness and reproducibility.

The Python programming language in particular has seen a surge in popularity across the sciences, for reasons which include its readability, modularity, and large standard library. The use of Python as a scientific programming language began to increase with the development of numerical libraries for optimized operations on large arrays in the late 1990s, in which an important development was the merging of the competing Numeric and Numarray packages in 2006 to form NumPy (Oliphant, 2007). As Python and NumPy have gained traction in a given scientific domain, we have seen the emergence of domain-specific ecosystems of open-source Python software developed by scientists. It became clear to us in 2007 that we were on the cusp of an emerging *Python in neuroscience* ecosystem, particularly in computational neuroscience and neuroimaging, but also in electrophysiological data analysis and in psychophysics.

Two major strengths of Python are its modularity and ability to easily “glue” together different programming languages, which together facilitate the interaction of modular components and their composition into larger systems. This focus on reusable components, which has proven its value in commercial and open-source software development (Brooks, 1987), is, we contend, essential for scientific computing in neuroscience, if we are to cope with the increasingly large amounts of data being produced in experimental labs, and if we wish to understand and model the brain in all its complexity.

We therefore felt that it was timely and important to raise awareness of the emerging Python in Neuroscience software ecosystem amongst researchers developing Python-based tools, but also in the larger neuroscience community.

Our goals were several-fold:

- establish a critical mass for Python use and development in the eyes of the community;
- encourage interoperability and collaboration between developers;
- expose neuroscientists to the new Python-based tools now available.

OPEN ACCESS

Edited and reviewed by:

Sean L. Hill,
International Neuroinformatics
Coordinating Facility, Sweden

*Correspondence:

Andrew P. Davison,
andrew.davison@unic.cnrs-gif.fr

Received: 20 March 2015

Accepted: 28 March 2015

Published: 14 April 2015

Citation:

Muller E, Bednar JA, Diesmann M,
Gewaltig M-O, Hines M and Davison
AP (2015) Python in neuroscience.
Front. Neuroinform. 9:11.
doi: 10.3389/fninf.2015.00011

From this was born the idea for a Research Topic in *Frontiers in Neuroinformatics* on “Python in Neuroscience” to showcase those projects we were aware of, and to give exposure to projects of which we were not aware. Although it may seem strange at first glance to center a Research Topic around a tool, rather than around a scientific problem, we feel it is justified by the increasingly critical role of scientific programming in neuroscience research, and by the particular strengths of the Python language and the broader Python scientific computing ecosystem.

Collected in this Research Topic are 24 articles describing some ways in which neuroscience researchers around the world are turning to the Python programming language to get their job done faster and more efficiently.

Overview of the Research Topic

We will now briefly summarize the 24 articles in the Research Topic, drawing out common themes.

Both Southey et al. (2008) and Yanashima et al. (2009) use Python for bioinformatics applications, but in very different areas. Yanashima et al. have developed a Python package for graph-theoretical analysis of biomolecular networks, BioNetpy, and employed it to investigate protein networks associated with Alzheimer’s disease. Southey et al.’s study demonstrates the wide breadth of application of Python, and the large number of high quality scientific libraries available, combining existing tools for bioinformatics, machine learning and web development to build an integrated pipeline for identification of prohormone precursors and prediction of prohormone cleavage sites.

Jurica and van Leeuwen (2009) address the needs of scientists who already have significant amounts of code written in MATLAB® and who wish to transfer this to Python. They present OMPC, which uses syntax adaptation and emulation to allow transparent import of existing MATLAB® functions into Python programs.

Three articles reported on new tools in the domain of neuroimaging. Hanke et al. (2009) report on PyMVPA, a Python framework for machine learning-based data analysis, and its application to analysis of fMRI, EEG, MEG, and extracellular electrophysiology recordings. Gouws et al. (2009) describe DataViewer3D, a Python application for displaying and integrating data from multiple neuroimaging modalities, showcasing Python’s abilities to easily interface with libraries written in other languages, such as C++, and to integrate them into user-friendly systems. Strangman et al. (2009) emphasize the advantages of Python for “*swift prototyping followed by efficient transition to stable production systems*” in their description of NinPy, a toolkit for near-infrared neuroimaging.

Zito et al. (2009) and Ince et al. (2009) both report on the use of Python for general purpose data analysis, with a focus on machine learning and information theory respectively. Zito et al. have developed MDP, the Modular toolkit for Data Processing, a collection of computationally efficient data analysis modules that can be combined into complex pipelines. MDP was originally developed for theoretical research in neuroscience, but has broad application in general scientific data analysis and in teaching. Ince et al. (2009) describe the use of Python for

information-theoretic analysis of neuroscience data, outlining algorithmic, statistical and numerical challenges in the application of information theory in neuroscience, and explaining how the use of Python has significantly improved the speed and domain of applicability of the algorithms, allowing more ambitious analyses of more complex data sets. Their code is available as an open-source package, pyEntropy.

Three articles report on tools for visual stimulus generation, for use in visual neurophysiology and psychophysics experiments. Straw (2008) describes VisionEgg, while Peirce (2009) presents PsychoPy, both of which are easy-to-use and easy-to-install applications that make use of OpenGL to generate temporally and spatially precise, arbitrarily complex visual stimulation protocols. Python is used to provide a simple, intuitive interface to the underlying graphics libraries, to provide a graphical user interface, and to interface with external hardware. PsychoPy can also generate and deliver auditory stimuli. Spacek et al. (2009) also report on a Python library for visual stimulus generation, as part of a toolkit for the acquisition and analysis of highly parallel electrophysiological recordings from cat and rat visual cortex. The other two components in the toolkit are for electrophysiological waveform visualization and spike sorting; and for spike train and stimulus analysis. The authors note “*The requirements and solutions for these projects differed greatly, yet we found Python to be well suited for all three.*”

Also in the domain of electrophysiology, Garcia and Fourcaud-Trocmé (2009) describe OpenElectrophy, an application for efficient storage and analysis of large electrophysiology datasets, which includes a graphical user interface for interactive visualization and exploration and a library of analysis routines, including several spike-sorting methods.

By far the largest contribution to the Research Topic came from the field of modeling and simulation, with 12 articles on the topic. Nine of these articles present neuroscience simulation environments with Python scripting interfaces. In most cases, the Python interface was added to an existing simulator written in a compiled language such as C++. This was the case for NEURON (Hines et al., 2009), NEST (Eppler et al., 2009), PCSIM (Pecovski et al., 2009), Nengo (Stewart et al., 2009), MOOSE (Ray and Bhalla, 2008), STEPS (Wils and De Schutter, 2009) and NCS (Drewes et al., 2009). However, as the articles by Goodman and Brette (2008) on the Brian simulator and Bednar (2009) on the Topographica simulator demonstrate, it is also possible to develop new simulation environments purely in Python, making use of the vectorization techniques available in the underlying NumPy package to obtain computational efficiency. The range of modeling domains of these simulators is wide, from stochastic simulation of coupled reaction-diffusion systems (STEPS), through simulation of morphologically detailed neurons and networks (NEURON, MOOSE), highly-efficient large-scale networks of spiking point neurons (NEST, PCSIM, NCS, Brian) to population coding or point-neuron models of large brain regions (Nengo, Topographica). Note that although we have categorized each simulator by its main area of application, most of these tools support modeling at a range of scales and levels of detail: Bednar (2009), for example, describes the integration of a

spiking NEST simulation as one component in a Topographica simulation.

The addition of Python interfaces to such a large number of widely used simulation environments suggested a huge opportunity to enhance interoperability between different simulators, making use of the common scripting language, which in turn has the potential to enhance the transfer of technology, knowledge and models between users of the different simulators, and to promote model reuse. Davison et al. (2009a) describe PyNN, a common Python interface to multiple simulators, which enables the same modeling and simulation script to be run on any supported simulator without modification. At the time of writing, PyNN supports NEURON, NEST, PCSIM and Brian, with MOOSE support under development. The existence of such a common “meta-simulator” then makes it much easier for scientists developing new, hardware-based approaches to neural simulation to engage with the computational neuroscience community, as evidenced by the article by Brüderle et al. (2009) on interfacing a novel neuromorphic hardware system with PyNN.

Finally, Fox et al. (2009) describe the possibilities when one is not limited to a single simulator, but can use Python to integrate multiple models into a brain-wide system. In their development of an integrated basal ganglia-hippocampal formation model for spatial navigation and its embodiment in a simulated robotic environment, Fox et al. found that Python offers “a significant reduction in development time, without a corresponding significant increase in execution time.”

It is important to note that most or all of the Python tools and libraries described in the Research Topic are open source and hence free to download, use and extend.

Discussion

This editorial is being written 6 years after the first articles in the Research Topic were published. It is with the benefit of considerable hindsight, therefore, that we can confidently say that our goals in launching this Research Topic—to establish a critical mass for Python use and development in the eyes of the community and to encourage interoperability and collaboration between developers—have been met or exceeded.

References

- Antolík, J., and Davison, A. P. (2013). Integrated workflows for spiking neuronal network simulations. *Front. Neuroinform.* 7:34. doi: 10.3389/fninf.2013.00034
- Bednar, J. A. (2009). Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Front. Neuroinform.* 3:8. doi: 10.3389/neuro.11.008.2009
- Brooks, F. P. Jr. (1987). No silver bullet: essence and accidents of software engineering. *Computer* 20, 10–19. doi: 10.1109/MC.1987.1663532
- Brüderle, D., Müller, E., Davison, A. P., Muller, E., Schemmel, J., and Meier, K. (2009). Establishing a novel modeling tool: a Python-based interface for a neuromorphic hardware system. *Front. Neuroinform.* 3:17 doi: 10.3389/neuro.11.017.2009
- Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecovski, D., et al. (2009a). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:11. doi: 10.3389/neuro.11.011.2008
- Davison, A. P., Hines, M., and Muller, E. (2009b). Trends in programming languages for neuroscience simulations. *Front. Neurosci.* 3, 374–380. doi: 10.3389/neuro.01.036.2009
- Djurfeldt, M. (2012). The connection-set algebra—a novel formalism for the representation of connectivity structure in neuronal network models. *Neuroinformatics* 10, 287–304. doi: 10.1007/s12021-012-9146-1
- Djurfeldt, M., Davison, A. P., and Eppler, J. M. (2014). Efficient generation of connectivity in neuronal networks from simulator-independent descriptions. *Front. Neuroinform.* 8:43. doi: 10.3389/fninf.2014.00043
- Drewes, R. P., 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. Neuroinform.* 3:16. doi: 10.3389/neuro.11.016.2009
- Einevoll, G. T. (2009). Sharing with Python. *Front. Neurosci.* 3, 334–335. doi: 10.3389/neuro.01.037.2009

The average number of citations per article for the Research Topic as a whole is 54, or approximately 9 per year, using figures from Google Scholar. Although citation counts from Google Scholar tend to be higher than those from Journal Citation Reports so the numbers are not directly comparable, this compares favorably with the impact factors of well respected journals such as Journal of Neuroscience or PLoS Computational Biology. Some of the articles were much more highly cited, with three of them being cited more than 20 times per year, on average, over the period. Four of the articles were chosen to “climb the tier” in the Frontiers system, and were followed up by Focused Review articles in Frontiers in Neuroscience (Davison et al., 2009b; Goodman and Brette, 2009; Hanke et al., 2010; Ince et al., 2010), another was the subject of a commentary (Einevoll, 2009).

Concerning the goals of interoperability and collaboration, several articles in a follow-up volume *Python in Neuroscience II* attest to the degree to which the developers of different tools have worked together, and prioritized interoperability in recent years. For example, the developers of OpenElectrophy (Garcia and Fourcaud-Trocmé, 2009) and the community around PyNN (Davison et al., 2009a) formed the nucleus of an effort to develop a baseline Python representation for electrophysiology data, which resulted in the Neo project, reported in the *Python in Neuroscience II* Research Topic (Garcia et al., 2014) together with two of the several projects which build on Neo (Pröpper and Obermayer, 2013; Sobolev et al., 2014). A new workflow system for computational neuroscience, Mozaik (Antolík and Davison, 2013) builds on both PyNN and Topographica (Bednar, 2009). PyNEST (Eppler et al., 2009) and PyNN developers collaborated with the INCF to improve the interoperability between these tools (Djurfeldt et al., 2014) when using the Connection Set Algebra (Djurfeldt, 2012). Finally, a number of tools have been built on the Python interface to NEURON (Hines et al., 2009), including morphforge (Hull and Willshaw, 2014) and LFPy (Lindén et al., 2014).

Observing the rapid growth in adoption of Python in neuroscience over the last 6 years, which appears to continue to accelerate, it is clear that Python is here to stay, which augurs well for the growth, productivity, and rigor of computational methods in neuroscience.

- Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M. O. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2008
- Fox, C. W., Humphries, M. D., Mitchinson, B., Kiss, T., Somogyi, Z., and Prescott, T. J. (2009). Technical integration of hippocampus, basal ganglia and physical models for spatial navigation. *Front. Neuroinform.* 3:6. doi: 10.3389/neuro.11.006.2009
- Garcia, S., and Fourcaud-Trocme, N. (2009). OpenElectrophy: an electrophysiological data- and analysis-sharing framework. *Front. Neuroinform.* 3:14. doi: 10.3389/neuro.11.014.2009
- Garcia, S., Guarino, D., Jaillet, F., Jennings, T., Pröpper, R., Rautenberg, P. L., et al. (2014). Neo: an object model for handling electrophysiology data in multiple formats. *Front. Neuroinform.* 8:10. doi: 10.3389/fninf.2014.00010
- Goodman, D. F., and Brette, R. (2009). The Brian simulator. *Front. Neurosci.* 3, 192–197. doi: 10.3389/neuro.01.026.2009
- Goodman, D. F. M., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2:5. doi: 10.3389/neuro.11.005.2008
- Gouws, A. D., Woods, W., Millman, R. E., Morland, A. B., and Green, G. G. R. (2009). Dataviewer3D: an open-source, cross-platform multi-modal neuroimaging data visualization tool. *Front. Neuroinform.* 2:9. doi: 10.3389/neuro.11.009.2009
- Hanke, M., Halchenko, Y. O., Haxby, J. V., and Pollmann, S. (2010). Statistical learning analysis in neuroscience: aiming for transparency. *Front. Neurosci.* 4, 38–43. doi: 10.3389/neuro.01.007.2010
- Hanke, M., Halchenko, Y. O., Sederberg, P. B., Olivetti, E., Fründ, I., Rieger, J. W., et al. (2009). PyMVPA: a unifying approach to the analysis of neuroscientific data. *Front. Neuroinform.* 3:3. doi: 10.3389/neuro.11.003.2009
- Hines, M., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinform.* 3:1. doi: 10.3389/neuro.11.001.2009
- Hull, M. J., and Willshaw, D. J. (2014). Morphforge: a toolbox for simulating small networks of biologically detailed neurons in Python. *Front. Neuroinform.* 7:47. doi: 10.3389/fninf.2013.00047
- Ince, R. A. A., Mazzoni, A., Petersen, R. S., and Panzeri, S. (2010). Open source tools for the information theoretic analysis of neural data. *Front. Neurosci.* 4, 62–70. doi: 10.3389/neuro.01.011.2010
- Ince, R. A. A., Petersen, R. S., Swan, D. C., and Panzeri, S. (2009). Python for information theoretic analysis of neural data. *Front. Neuroinform.* 3:4. doi: 10.3389/neuro.11.004.2009
- Jurica, P., and van Leeuwen, C. (2009). OMPC: an open-source MATLAB®-to-Python compiler. *Front. Neuroinform.* 3:5. doi: 10.3389/neuro.11.005.2009
- Lindén, H., Hagen, E., Łęski, S., Norheim, E. S., Pettersen, K. H., and Einevoll, G. T. (2014). LFPy: a tool for biophysical simulation of extracellular potentials generated by detailed model neurons. *Front. Neuroinform.* 7:41. doi: 10.3389/fninf.2013.00041
- Oliphant, T. E. (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20. doi: 10.1109/MCSE.2007.58
- Pecevski, D., Natschläger, T., and Schuch, K. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Front. Neuroinform.* 3:11. doi: 10.3389/neuro.11.011.2009
- Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Front. Neuroinform.* 2:10. doi: 10.3389/neuro.11.010.2008
- Pröpper, R., and Obermayer, K. (2013). Spyke Viewer: a flexible and extensible platform for electrophysiological data analysis. *Front. Neuroinform.* 7:26. doi: 10.3389/fninf.2013.00026
- Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2:6. doi: 10.3389/neuro.11.006.2008
- Sobolev, A., Stoewer, A., Pereira, M., Kellner, C. J., Garbers, C., Rautenberg, P. L., et al. (2014). Data management routines for reproducible research using the G-Node Python Client library. *Front. Neuroinform.* 8:15. doi: 10.3389/fninf.2014.00015
- Southey, B., Sweedler, J., and Rodriguez-Zas, S. (2008). A Python analytical pipeline to identify prohormone precursors and predict prohormone cleavage sites. *Front. Neuroinform.* 2:7. doi: 10.3389/neuro.11.007.2008
- Spacek, M. A., Blanche, T., and Swindale, N. (2009). Python for large-scale electrophysiology. *Front. Neuroinform.* 2:9. doi: 10.3389/neuro.11.009.2008
- Stewart, C., Tripp, B., and Elias Smith, C. (2009). Python scripting in the Nengo simulator. *Front. Neuroinform.* 2:7. doi: 10.3389/neuro.11.007.2009
- Strangman, G. E., Zhang, Q., and Zeffiro, T. (2009). Near-infrared neuroimaging with Nipy. *Front. Neuroinform.* 2:12. doi: 10.3389/neuro.11.012.2009
- Straw, A. D. (2008). Vision egg: an open-source library for realtime visual stimulus generation. *Front. Neuroinform.* 2:4. doi: 10.3389/neuro.11.004.2008
- Wilson, G. (2006). Software carpentry: getting scientists to write better code by making them more productive. *Comput. Sci. Eng.* 8, 66–69. doi: 10.1109/MCSE.2006.122
- Wils, S., and De Schutter, E. (2009). STEPS: modeling and simulating complex reaction-diffusion systems with Python. *Front. Neuroinform.* 3:15. doi: 10.3389/neuro.11.015.2009
- Yanashima, R., Kitagawa, N., Matsubara, Y., Weatheritt, R., Oka, K., Kikuchi, S., et al. (2009). Network features and pathway analyses of a signal transduction cascade. *Front. Neuroinform.* 2:13. doi: 10.3389/neuro.11.013.2009
- Zito, T., Wilbert, N., Wiskott, L., and Berkes, P. (2009). Modular toolkit for data processing (MDP): a Python data processing framework. *Front. Neuroinform.* 2:8. doi: 10.3389/neuro.11.008.2008

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

Copyright © 2015 Muller, Bednar, Diesmann, Gewaltig, Hines and Davison. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) or licensor are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.



STEPS: modeling and simulating complex reaction-diffusion systems with Python

Stefan Wils^{1,2} and Erik De Schutter^{1,2*}

¹ Theoretical Neurobiology, University of Antwerp, Belgium

² Computational Neuroscience Unit, Okinawa Institute of Science and Technology, Japan

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Jeanette Koteleski, Karolinska Institute,
Sweden

Kim Avrama Blackwell, George Mason
University, Krasnow Institute, USA

*Correspondence:

Erik De Schutter, Theoretical
Neurobiology, Biomedical Sciences
Department, University of Antwerp,
Universiteitsplein 1, 2610 Wilrijk,
Belgium.
e-mail: erik@tnb.ua.ac.be

We describe how the use of the Python language improved the user interface of the program STEPS. STEPS is a simulation platform for modeling and stochastic simulation of coupled reaction-diffusion systems with complex 3-dimensional boundary conditions. Setting up such models is a complicated process that consists of many phases. Initial versions of STEPS relied on a static input format that did not cleanly separate these phases, limiting modelers in how they could control the simulation and becoming increasingly complex as new features and new simulation algorithms were added. We solved all of these problems by tightly integrating STEPS with Python, using SWIG to expose our existing simulation code.

Keywords: Python, software, simulator, reaction kinetics, 3D diffusion, signaling pathway, scripting

INTRODUCTION

Computational modeling and simulation of signaling pathways has become a valuable and established tool for studying the molecular aspects of biological systems (Bhalla, 2004; Doi et al., 2005; Holmes, 2000; Kuroda et al., 2001; Lindskog et al., 2006; Miller et al., 2005; Smolen et al., 2006; Stefan et al., 2008). Modeling such systems consists of identifying the molecular players and describing the stoichiometry and rate constants of their chemical interactions. The resulting system is then often simulated by converting it to a set of coupled ordinary differential equations that can be numerically integrated (Press et al., 2007).

It has long been acknowledged that the discrete nature of reaction events, caused by the very low numbers of key molecules being present, can make biological reaction systems noisy and affect their behavior on a macroscopic level. This aspect can be brought into the simulation by adding noise terms to the differential equations (Kloeden and Platen, 1999; van Kampen, 2007), or more commonly, by simulating the system with Gillespie's Stochastic Simulation Algorithm or SSA (Gillespie, 1977) or one of its derivations (Gillespie, 2007).

For some pathways, however, even more realism is needed. One such case is when the spatial organization and morphology of the cell is known to play an active role in controlling the pathway, e.g. through chemical compartmentalization, spatial gradients and by various transport processes and diffusion (Lemerle et al., 2005). Such cases are common in neurons because of their complex dendritic arborization (Santamaria et al., 2006), but of course are not limited to them.

In order to study systems at the level where stochasticity, spatial gradients within complex boundary conditions and diffusion all come into play at the same time, we have developed a simulation platform called STEPS (STochastic Engine for Pathway Simulation) that uses an extension of Gillespie's SSA to deal with diffusion

of molecules in 3-dimensional reconstructions of neuronal morphology and tissue (Wils and De Schutter, 2009). STEPS computes reactions occurring between diffusing molecules in volumes, and, in addition, also surface reactions to simulate channel fluxes and ligand-receptor binding. Our algorithm differs from a similar approach described in Elf and Ehrenberg (2004) mainly in that it is based upon the use of tetrahedral meshes which are particularly well-suited for representing biological morphology and that we avoid the use of a heap structure.

In this paper, based on a presentation made at the FACETS CodeJam #2 workshop 'Building the meta-simulator tool-chain: leveraging Python for a robust and efficient workflow in computational neuroscience', describes how Python scripting is used for working with models in STEPS. We also show how, in this particular problem domain, adding Python scripting improved the quality and maintainability of STEPS in a fundamental way.

SOFTWARE

BRIEF DESCRIPTION OF STEPS ALGORITHM

Stochastic simulation of reaction-diffusion processes can occur in a number of ways. One way is to track each reacting molecule as an independent particle that undergoes Brownian motion and occasionally collides with one of the other tracked molecules. This is the approach taken by such programs as M-Cell (Stiles and Bartol, 2001) and Smoldyn (Andrews and Bray, 2004).

Another approach is voxel-based; here one keeps track of how much molecules are present from any given species within a set of small volumes. By keeping these reaction volumes or voxels small enough, we can state that the concentration gradients within each voxel are negligible: the voxel is approximately well-mixed. Then we can apply SSA (Gillespie, 1976) by adding an extra reaction rule for each type of molecule for its diffusion step from one voxel to a neighboring one. Thus SSA handles both diffusion processes

and reaction mechanisms from within one single simulation framework. Though this approach is abstracted more from the underlying physical mechanisms than modeling Brownian motion, it offers a number of advantages. Because diffusion is uncoupled from chemical reaction, the modeler can decide for each type of molecule, considering the timeframe being simulated, whether it makes sense to implement diffusion or not. At the coding level, much less bookkeeping is necessary because one does not track individual molecules, giving rise to leaner and potentially faster code. It also facilitates combining SSA with approximate, faster methods such as tau-leaping (Gillespie, 2001).

STEPS simulates molecular reaction-diffusion in volumes which are bounded by membranes. These membranes can contain stationary reacting molecules, including channel proteins. To simulate the behavior of these systems, STEPS adapts the Direct Reaction Method version of SSA (Gillespie, 1976) for large systems by storing the propensity values for each process in a search tree. STEPS 0.4 implements two distinct stochastic solvers: a spatial solver (called *tetexact*) and an auxiliary well-mixed solver (called *wmdirect*, this does not model diffusion). Such well-mixed solvers are useful assistants because setting up a spatial model can benefit greatly from analyzing and tuning parts of the biochemical model under simpler conditions (Wils and De Schutter, 2009). In the future additional solvers will be added, including a deterministic one (based on Runge-Kutta integration; Press et al., 2007) and an extension of *tetexact* that includes diffusion in membranes.

STEPS WORKFLOW

Figure 1 shows a typical workflow for developing and simulating a 3-dimensional reaction-diffusion system and how the different phases can relate to each other. The first step, biochemical modeling, consists of describing reaction stoichiometry and selecting reaction rates and diffusion constants. Since this is independent to a large degree of the actual algorithm that will be used for simulation (i.e. numerical integration of ODE's vs stochastic simulation; with or without diffusion; ...), it is common practice to import and

compose this type of information from previous modeling efforts through formats such as SBML (Hucka et al., 2003)¹.

Mesh generation, or more generally speaking describing the geometric boundaries of the problem, is another step. Since tetrahedral meshes are supported both by stochastic solvers (Wils and De Schutter, 2009) as well as more traditional methods based on numerical integration of systems of partial differential equations (Ferziger and Peric, 2002), they are fairly independent of the algorithm that will be used at a later stage. In addition, a mesh can be reused with multiple modeling and simulation studies, a distinct advantage considering that their generation can be a rather elaborate task, especially for meshes based on imaging data (Means et al., 2006).

Because of their independence, the previous two phases can easily be performed in parallel, or even by separate groups. The only point where everything needs to come together and link up, is at the start of the third phase: running a simulation. This phase is the focus of STEPS and will be detailed below.

The fourth and final phase is the most important and daunting of all: collecting the simulation results, analyzing them and, if necessary, readjusting the biochemical model. Even more than was the case with the first two phases, different modelers will want to rely on different tools for this task. A logical option for STEPS modeling results are the many packages already available for Python (Scipy, Matplotlib, ...).

In the rest of this section, we will implement the simple toy model in Figure 2 to examine in more detail how different STEPS packages support each of the first three phases of our modeling cycle independently. We will show how easy it is to go from well-mixed to spatial simulations and back. We will then conclude our discussion of STEPS by looking at it from an architectural point of view and discuss the multiple roles that Python plays in allowing STEPS users to combine all the components of this cycle into a modeling pipeline.

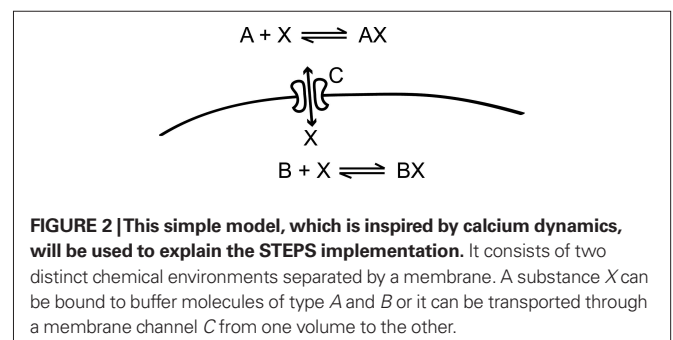
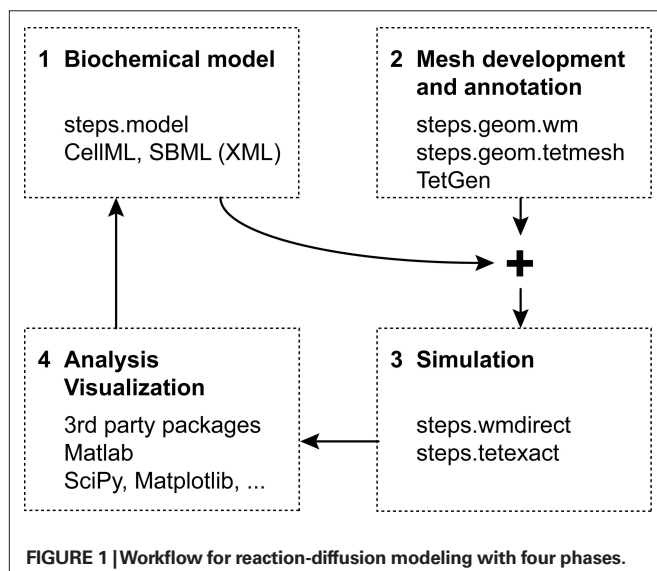
BIOCHEMICAL MODEL DESCRIPTION

The objects that together define the biochemical aspects of a STEPS model are written directly in Python and are grouped in package *steps.model*. The following snippet of Python code shows how to implement the simple toy model from Figure 2 using these objects:

```
from steps.model import *

# Create the model m
m = Model()
```

¹<http://www.sbml.org>



```

# Define all species of molecules in the model m
a = Spec('A', m)
b = Spec('B', m)
c = Spec('C', m)
x = Spec('X', m)
ax = Spec('AX', m)
bx = Spec('BX', m)

# Set up volume A of m and define all reactions in A
vs_ak = Volsys('A_kin', m)
# partners, right hand side partners and a rate
# constant.
# hand side partners and a rate constant.
ax_f = Reac('AX_f', vs_ak, lhs=[a,x], rhs=[ax],\
    kcst = 1.0e8)
ax_b = Reac('AX_b', vs_ak, lhs=[ax], rhs=[a,x],\
    kcst = 1.0e3)
# Set a diffusion constant for x
vs_ak_xdiff = Diff('AX_xdiff', vs_ak, x,\
    dcst = 0.065e-9)

# Set up volume B of m and define all reactions in B
vs_bk = Volsys('B_kin', m)
bx_f = Reac('BX_f', vs_bk, lhs=[b,x], rhs=[bx],\
    kcst = 2.0e8)
bx_b = Reac('BX_b', vs_bk, lhs=[bx], rhs=[b,x],\
    kcst = 2.0e3)
vs_bk_xdiff = Diff('BX_xdiff', vs_bk, x,\
    dcst = 0.065e-9)

# Set up simple membrane channel kinetics for C. With
# surface reactions, the reactants (lhs) and products
# (rhs) have to be marked as being located on the
# inside (i), outside (o) or surface (s) of the
# membrane.
ss_cchan = Surfsys('C_chan', m)
c_xflux_f = SReac('C_Xflux_f', ss_cchan, vlhs=[x],\
    slhs=[c], orhs=[x], srhs=[c])
c_xflux_f.kcst = 10.0e6
c_xflux_b = SReac('C_Xflux_b', ss_cchan, vlhs=[x],\
    slhs=[c], irhs=[x], srhs=[c])
c_xflux_b.kcst = 10.0e6

```

As one can see the model is created through a series of Python function calls that map onto STEPS code (see **Figure 4**). Volume systems (objects of class *Volsys*) describe the chemical properties of volume solutions, which comprise the stoichiometry and rate constants of reaction channels and the diffusion constants for all diffusing species in that solution. Surface systems (objects of class *Surfsys*) describe the chemical properties of membranes, such as ligand-receptor binding and unbinding or channel currents. Note that some information is given implicitly: because no diffusion constants are supplied for the molecular species *A*, *B* and *AX*, *BX* these are considered immobile.

Demonstrating the independence between the model construction phases mentioned earlier, this code shows that this level of description is completely separate from the geometry or the spatial 'location' of these volume and surface systems, and of the initial and boundary conditions or simulation events. *Volsys* and *Surfsys* are essentially just static template objects that group together related reaction rules and that will, at a later point in time, be *instantiated* on the actual simulation geometry. This uncoupling, which is somewhat

different from the approach used in SBML where the kinetic equations are usually mixed with compartment definitions and initial conditions, makes it easy for modelers to compose and recombine their biochemical models with different geometric descriptions. Since the objects themselves are in the end still just static hierarchies, a linking point with formats such as SBML or CellML² remains.

3D BOUNDARIES: TETRAHEDRAL MESHES

STEPS uses unstructured, tetrahedral meshes (Ferziger and Peric, 2002; see **Figure 3A** for an example) to describe the geometric domain in 3-dimensional detail. In these meshes, elements are not numbered along principal axes and do not have to be perfectly regular, allowing them to adapt to the local level of detail and to follow an arbitrary set of domain boundaries rather smoothly. We will not describe the Python scripting (*steps.mesh*) in detail, but instead focus on the conceptual approach.

To organize the simulation space into biological structures STEPS uses the notion of 'compartment' for volumes and 'patches' for surfaces. For example, compartments can represent physical regions such as the cytoplasm, ER lumen or cellular exterior. In order to be useful for a simulation, the tetrahedral mesh has to be annotated so that each tetrahedron is assigned to a 'compartment' (objects of class *Comp*) and each triangle is assigned to a 'patch' (objects of class *Patch*). When these objects are used directly, instead of a mesh, it is possible to describe a well-mixed geometry that can be used in well-mixed simulations, similar to the compartments found in SBML.

Eventually, *Comp* and *Patch* objects will refer to one or more volume systems or surface systems, respectively. As detailed in the next section, these references are resolved during the initialization phase of a simulation, when a model description is combined with a mesh object. At any point prior to simulation, however, these references are stored simply as string values, allowing users to manipulate meshes independently of any biochemical model. A mesh can be stored with or without such references, making it easy to reuse a mesh for simulating many different biochemical models.

As is the case with the objects of package *steps.model*, meshes are Python objects that can be manipulated using Python scripts or from the Python command line. It therefore becomes easier to automate many tasks and to write custom importers or exporters for various forms of 3D data. Currently, STEPS directly supports importing meshes from the freely available tetmesh generator TetGen³.

In the following snippet of code, we load a previously generated mesh (**Figure 3A**) stored in an archive. The mesh, which is used for demonstration purposes only, consists of two cylindrical compartments (called *outer* and *inner*) separated by a membrane patch called *imem*. This could represent, for example, a segment of dendrite with endoplasmic reticulum in its center. We link the mesh to our toy model by assigning these compartments and patches to volume systems and surface systems as needed.

```

# load the annotated mesh from a Python pickled archive
meshf = open('cyl.dat')
mesh = pickle.load(meshf)

```

²<http://www.cellml.org>

³<http://tetgen.berlios.de>

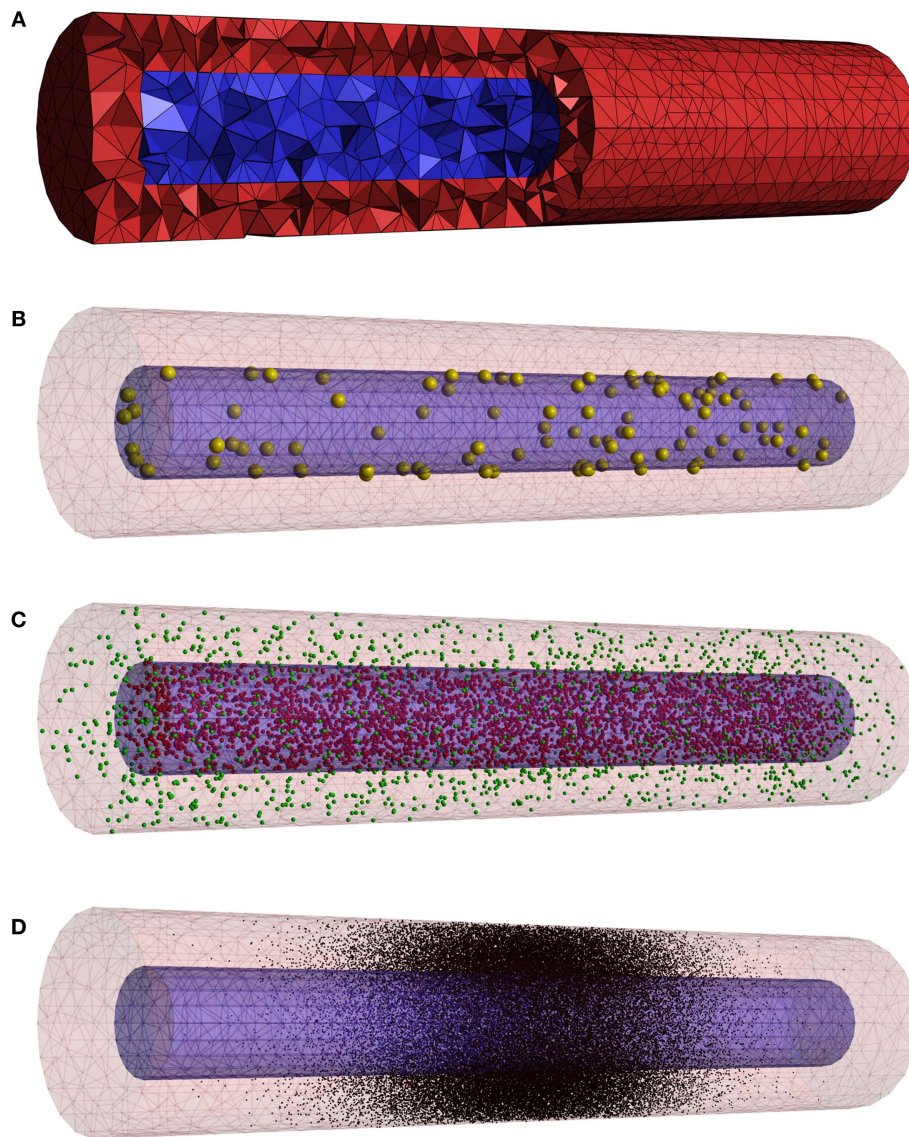


FIGURE 3 | The mesh used in the code examples for setting up initial conditions. (A) This opaque view with a cut-out shows that 21090 tetrahedrons are used to describe one cylinder surrounding another one. **(B)** Membrane channels *C* are distributed randomly over the inner membrane. **(C)** A uniform initial distribution for molecules *A* and *B*. **(D)** A Gaussian distribution in the center of the outer cylinder for *X*.

```
meshf.close()
# assign volume and surface systems to different parts
# of the mesh
mesh.getComp('outer').addVolsys('A_kin')
mesh.getComp('inner').addVolsys('B_kin')
mesh.getPatch('imem').addSurfsys('C_chan')
```

Notice that we only needed three line of code to perform this link to a fairly complex mesh.

RUNNING A SIMULATION

The third phase in the modeling cycling is to simulate the model with a numerical solver. To do this in STEPS, a solver object must be created. This basically consists of one line of code in which this object is created and initialized with the biochemical model, a

geometric description and a random number generator. It is from within the constructor of this solver object that all references from the *Comp* and *Patch* objects to the *Volsys* and *Surfsys* objects are resolved in order to create the appropriate data structures needed to represent the state of the simulation.

```
rng = steps.rng.create('mt19937')
rng.initialize(datetime.datetime.now().microsecond)
sim = steps.tetexact.Solver(m, mesh, rng)
# Make the simulator ready for action.
sim.reset()
```

The spatial solver (*steps.tetexact*) used in this example only accepts tetrahedral meshes, whereas the well-mixed solver (*steps.wmdirect*) can accept both a well-mixed description or a tetmesh,

from which the well-mixed features can be transparently extracted. Note that in the latter case the definition of diffusion constants in our toy model would be ignored automatically. The only change needed for setting up a well-mixed simulation would be in the third line of the code example, where *steps.wmDirect.Solver* would be evoked instead.

All current and future solver objects, regardless of their underlying algorithm or of their spatial or well-mixed nature, provide the same API through which the modeler can access the internal state of the simulation from within Python, in order to set initial conditions and to control the simulation. This internal state includes the local amount of molecules for different species, but also whether these species are buffered, the reaction and diffusion constants and whether reaction channels are active or not. All of these properties can be manipulated for individual tetrahedrons and triangles (in mesh-based solvers), or for entire compartments and patches at a time (in both mesh-based and well-mixed solvers). In **Figures 3B–D** we show three possible initial conditions for the concentration of *X* from our toy model. We first inject 100 channels of species *C* in the inner membrane *imem* (**Figure 3B**):

```
sim.setPatchCount('imem', 'C', 100)
```

Next, we inject 1500 molecules of species *A* in the *outer* compartment and set the concentration of species *B* in the inner compartment to 1 μM , spread out uniformly (**Figure 3C**):

```
sim.setCompCount('outer', 'A', 1500)
sim.setCompConc('inner', 'B', 1.0e-6)
```

Note that in both examples the position of channels or molecules is automatically randomized with uniform distributions. Because the API also allows access to the simulation state at the level of individual tetrahedrons, we can program arbitrarily complex initial conditions and runtime events. In the next piece of code we show how this can be used to generate a normally distributed pulse injection of *X* in the *outer* compartment with a given peak amplitude and width centered in the middle (**Figure 3D**):

```
# Set the concentration of a species in a compartment
# using a 3D density function.
def setCompConcDensity(sim, mesh, compname, specname,
    conc, dens, sampling=10):
    r = steps.rng.create('mt19937')
    r.initialize(datetime.datetime.now().microsecond)
    # Loop over all tetrahedrons of the requested
    # compartment
    for t in mesh.getComp(compname).tets:
        # Generate a number of random points in the
        # current tetrahedron and use these points
        # to sample the density function.
        dens2 = dens(t.getRanPnt(r, sampling)).mean()
        # Set the concentration in the tetrahedron to
        # the product of mean density value and the
        # peak concentration.
        sim.setTetConc(t.idx, specname, conc * dens2)

# Example of a density function which generates a
# Gaussian distribution
```

```
def dGaussian(p):
    m = 0.0
    s = 1.0e-6
    m2 = (p[:,0]-m)
    return np.exp(-(m2*m2) / (2*s*s))
```

```
# use both functions to set the initial conditions
setCompConcDensity(sim, mesh, 'outer', 'X', 20.0e-6, \
dGaussian)
```

These examples show the great flexibility that Python offers in setting up initial conditions for the simulation. In addition, the API also features the actual control functions that allow one to reset a simulation, to advance the simulation to some future time and to sample the simulation state.

WHY PYTHON?

To understand the design of the STEPS software package, a short history is useful. An earlier incarnation of STEPS consisted of a single standalone C++ application. Being focused on the simulation algorithm itself, not much thought was given to issues related to model description and simulation control and these aspects were put together in a single custom XML-based format. We didn't use SBML at the time because it lacked support for models with detailed 3D features. Meshes had to be stored in a separate custom data format and were referenced by filename from within these XML input files.

The limitations of our first implementation became apparent rather quickly. We discovered that, because of the spatial aspects, describing the initial state of a 3D reaction-diffusion system is more complicated than describing the initial state of a well-mixed simulation. People might not just want to set initial values in compartments as a whole, but inject molecules or manipulate rate constants using more sophisticated geometric patterns, for instance using a Gaussian distribution to mimic the result of a laser uncaging event (Wang and Augustine, 1995; see **Figure 3D**). Sometimes the simulation might require this release pattern to be confined to a particular compartment; other simulations might want the pattern to be applied globally.

Coming up with an XML-based way of describing a wide range of in-simulation events, a problem similar and closely related to the problem of setting up initial conditions, and output generation proved to be quite difficult. By far the most common use case would be to have events occur on specific times during the course of a simulation. But what if an event would have to depend on some condition being met, such as the concentration of some species reaching a threshold? We ended up with an increasingly rich fauna of trigger, action and output objects which covered many possibilities, but which was complicated and costly to maintain and in the end still left many rare but sensible use cases uncovered.

When at some point we also started thinking about supporting well-mixed solvers directly from within STEPS, we decided that our old approach had reached its limits and set out to redesign STEPS by integrating it closely with a fully-featured scripting language. Python was chosen because it is a mature language, simple to learn and already had a widespread user base in the computational sciences, with a wide selection of third-party packages and documentation to match. As described above, its object oriented

features allowed us to express the relationship between well-mixed and spatial models in a way that facilitates switching between the corresponding classes of simulators. Python's excellent XML features will allow us to keep up with projects such as SBML when their support for spatial modeling matures. Finally, Python can be used to integrate many miscellaneous tasks related to simulation that would otherwise typically be done with shell scripting. Examples are copying files to their right location, cleaning up, initiating a data processing or compacting method directly after a simulation finishes, etc.

The redesign was a major effort. The only part that could be reused from the old STEPS was the core simulator code, i.e. the solver currently known as tetexact. Everything else had to be rewritten following the modeling workflow described in **Figure 1**. We designed the solver API mentioned above and implemented it for our two current solvers. These API implementations were then exposed to Python using SWIG⁴, where they were further wrapped in a Python-side Solver base class that performs argument checking and provides some extra higher-level functionality. Much of the code for setting up a solver is the same for all current and future solvers and was therefore put in a shared set of C++ files. This reduces the amount of 'plumbing code' that needs to be written for a new solver, while still allowing considerable freedom in choosing the ultimate algorithm-specific internal data structures.

The main flaw of our first version of Pythonizing STEPS, as shown in **Figure 4**, is the many layers that have to be passed to go from calling a solver object method to the actual solver code and back. This may become a performance bottleneck when one is running a simulation that is interrupted repeatedly over small time intervals. This problem may be resolved in several ways. We can recode the Python-side *Solver* class, which is shared by all solvers, in C++ and derive an actual individual solver by overriding protected virtual methods. To avoid even the cost of virtual calls in this scenario, we can employ the Curiously Recurring Template Pattern (Vandevoorde and Josuttis, 2003). Alternatively, we can switch from SWIG to Boost.Python⁵, an ingenious method of exposing C++

code to Python that does not result in a Python-side shadow class, as is the case with SWIG.

DISCUSSION

We have described how STEPS mixes C++ with Python scripting to give modelers greater freedom in setting up and simulating a model, while maintaining the efficiency of compiled and optimized C++ code. We described how going the extra mile to make a scientific simulator fully scriptable in this way has considerable advantages. Because of the many scientific computing packages already available for Python, computational scientists are encouraged to develop sophisticated pipelines in which modeling, simulation and even post processing and visualization are highly automated. In addition, we find that the neural simulators such as Neuron (Carnevale and Hines, 2006) and Moose⁶ have committed to supporting Python, leading some to forward the challenging but intriguing possibility of using Python to actually 'glue' together simulations (Cannon et al., 2007). One should keep in mind, however, that naively using an interpreted language like Python to exchange and map state information between simulators at each time step might quickly run into performance and numerical issues that could be avoided only by deeper integration at the algorithmic level. Alternatives like the MUSIC project (Ekeberg and Djurfeldt, 2008) might therefore be better suited for this.

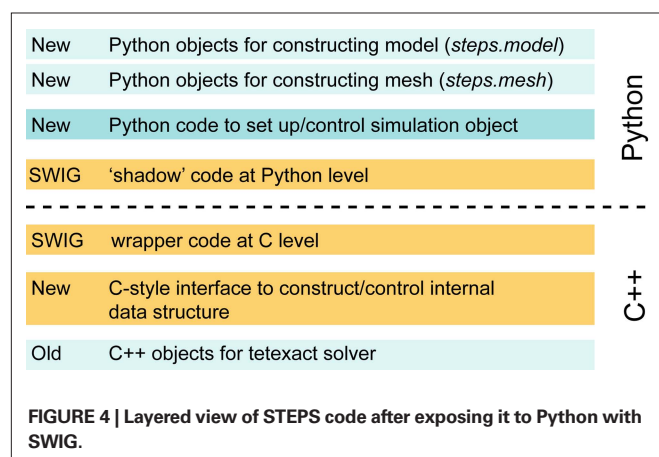
Like many before us, we have successfully used SWIG to expose our existing C++ simulation core to Python. The main technical issue that we encountered is the many layers between the user script and the C++ code which, as mentioned, can be resolved by porting the solver interface to C++ and possibly by switching to Boost.Python.

In the specific context of modeling 3D reaction-diffusion simulations we found that using Python had a large advantage for describing a complex internal state. There are many ways in which a biologist might want to set up and control this state and sample it for output. Switching to a scripting language allowed us to eliminate a great deal of complexity that was ultimately caused by sticking to a static, purely declarative input format in which model and simulation were thoughtlessly mixed. Since maintaining a backwards compatible API of basic getter/setter functions is less of an effort than designing and maintaining an increasingly 'baroque' set of trigger, action and output objects, we expect that this investment will keep paying off as STEPS keeps growing by adding more solvers and more capabilities. In other words, our switch to Python has actually saved us quite some time.

Finally, we believe that our experience suggests that a language like Python, as was proposed earlier in Cannon et al. (2007), can play a positive role in supporting the development of formal standards for sharing scientific models. Mirroring the requirements of understanding biology itself, biological simulators will necessarily become more complex and will be able to simulate more and more aspects of the living cell. Codes such as M-Cell (Stiles and

⁴<http://www.swig.org>

⁵<http://www.boost.org>



⁶<http://moose.sourceforge.net>

Bartol, 2001), MesoRD (Hattne et al., 2005), Smoldyn (Andrews and Bray, 2004) and also STEPS expand on the idea of ODE-based, well-mixed simulations of reaction kinetics by adding stochasticity and spatial processes such as diffusion. But this is only the beginning. The future will see developments such as simulations of electrophysiological phenomena in high 3D detail or full electrodiffusion (Lopreore et al., 2008), volume-occupying molecules (Gillespie et al., 2007; Schnell and Turner, 2004), dynamic meshes whose shape is controlled by simulated chemistry and, as mentioned earlier, possibly even the integration of simulators that work on different scales.

The designers of formal standards, such as SBML, can not be expected to keep up with these new trends as they come out, and still maintain a clean standard. This fact flows from a fundamental tension between on the one hand having a clean, simulator-independent standard for publishing models, and on the other hand the turbulent, seemingly endless expansion of exactly what is required in a biological model to be relevant

and how to breathe it all to life on a computer. The advantages of having such standards is obviously too great to discard (Bergmann and Sauro, 2008), and successes have been achieved to where classes of modeling efforts have sufficiently crystallized, together with the methods to simulate them (Hucka et al., 2003). The combination of Python and XML eases this tension by allowing projects that explore new types of simulations to mature independently from the standards for model sharing. It allows them to catch up with each other whenever and wherever it makes sense to do so.

ACKNOWLEDGEMENTS

This work was supported by grants from GOA (UA, Belgium), HFSP and OIST (Japan).

SUPPLEMENTARY MATERIAL

Our software is released under the GNU public license and can be downloaded from <http://sourceforge.net/projects/steps>.

REFERENCES

- Andrews, S. S., and Bray, D. (2004). Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Phys. Biol.* 1, 137–151.
- Bergmann, F. T., and Sauro, H. M. (2008). Comparing simulation results of SBML capable simulators. *Bioinformatics* 24, 1963–1965.
- Bhalla, U. S. (2004). Models of cell signaling pathways. *Curr. Opin. Genet. Dev.* 14, 375–381.
- 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.
- Carnevale, T. C., and Hines, M. L. (2006). *The NEURON Book*. Cambridge, Cambridge University Press.
- Doi, T., Kuroda, S., Michikawa, T., and Kawato, M. (2005). Spike-timing detection by calcium signaling pathways of cerebellar Purkinje cells in different forms of long-term depression. *J. Neurosci.* 25, 950–961.
- Ekeberg, Ö., and Djurfeldt, M. (2008). MUSIC – Multisimulation Coordinator: Request For Comments. *Nature Precedings*. Available at: <http://dx.doi.org/10.1038/npre.2008.1830.1>.
- Elf, J., and Ehrenberg, M. (2004). Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Syst. Biol.* 1, 230–236.
- Ferziger, J. H., and Peric, M. (2002). *Computational Methods for Fluid Dynamics*, 3rd Edn. Berlin, Springer-Verlag.
- Gillespie, D. T. (1976). A general method for numerically simulating the stochastic time evolution of coupled chemical species. *J. Comput. Phys.* 22, 403–434.
- Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.* 81, 2340–2361.
- Gillespie, D. T. (2001). Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys.* 115, 1716–1733.
- Gillespie, D. T. (2007). Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.* 58, 35–55.
- Gillespie, D. T., Lampoudi, S., and Petzold, L. R. (2007). Effect of reactant size on discrete stochastic chemical kinetics. *J. Chem. Phys.* 126, 034302.
- Hattne, J., Fange, D., and Elf, J. (2005). Stochastic reaction-diffusion simulation with MesoRD. *Bioinformatics* 21, 2923–2924.
- Holmes, W. R. (2000). Models of calmodulin trapping and CaM kinase II activation in a dendritic spine. *J. Comput. Neurosci.* 8, 65–68.
- 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., Mjølness, 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.
- Kloeden, P. E., and Platen, E. (1999). *Numerical Solution of Stochastic Differential Equations*, 3rd Edn. Berlin, Springer-Verlag.
- Kuroda, S., Schweighofer, N., and Kawato, M. (2001). Exploration of signal transduction pathways in cerebellar long-term depression by kinetic simulation. *J. Neurosci.* 21, 5693–5702.
- Lemerle, C., Di Ventura, B., and Serrano, L. (2005). Space as the final frontier in stochastic simulations of biological systems. *FEBS Lett.* 579, 1789–1794.
- Lindskog, M., Kim, M., Wikström, M. A., Blackwell, K. T., and Kotaleski, J. H. (2006). Transient calcium and dopamine increase PKA activity and DARPP-32 phosphorylation. *PLoS Comput. Biol.* 2, e119.
- Lopreore, C. L., Bartol, T. M., Coggan, J. S., Keller, D. X., Sosinsky, G. E., Ellisman, M. H., and Sejnowski, T. J. (2008). Computational modeling of three-dimensional electrodiffusion in biological systems: applications to the node of Ranvier. *Biophys. J.* 95, 2624–2635.
- Means, S., Smith, A. J., Shepherd, J., Shadid, J., Fowler, J., Wojcikiewicz, R. J. H., Mazel, T., Smith, G. D., and Wilson, B. S. (2006). Reaction diffusion modeling of calcium dynamics with realistic ER geometry. *Biophys. J.* 91, 537–557.
- Miller, P., Zhabotinsky, A. M., Lisman, J. E., and Wang, X. J. (2005). The stability of a stochastic CaMKII switch: dependence on the number of enzyme molecules and protein turnover. *PLoS Biol.* 3, e107.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes in C: The Art of Scientific Computing*, 3rd Edn. Cambridge, Cambridge University Press.
- Santamaria, F., Wils, S., De Schutter, E., and Augustine, G. J. (2006). Anomalous diffusion in Purkinje cell dendrites caused by spines. *Neuron* 52, 635–648.
- Schnell, S., and Turner, T. E. (2004). Reaction kinetics in intracellular environments with macromolecular crowding: simulations and rate laws. *Prog. Biophys. Mol. Biol.* 85, 235–260.
- Smolen, P., Baxter, D. A., and Byrne, J. H. (2006). A model of the roles of essential kinases in the induction and expression of late long-term potentiation. *Biophys. J.* 90, 2760–2775.
- Stefan, M. I., Edelstein, S. J., and Le Novère, N. (2008). An allosteric model of calmodulin explains differential activation of PP2B and CaMKII. *Proc. Natl. Acad. Sci. U.S.A.* 105, 10768–10773.
- Stiles, J. R., and Bartol, T. M. (2001). Monte Carlo methods for simulating realistic synaptic microphysiology using MCell. In *Computational Neuroscience: Realistic Modeling for Experimentalists*, E. De Schutter, ed. (Boca Raton, CRC Press).
- van Kampen, N. G. (2007). *Stochastic Processes in Physics and Chemistry*, 3rd Edn. Amsterdam, Elsevier.
- Vandevoorde, D., and Josuttis, N. M. (2003). *C++ Templates: The Complete Guide*. Reading, MA, Addison-Wesley.

- Wang, S. S., and Augustine, G. J. (1995). Confocal imaging and local photolysis of caged compounds: dual probes of synaptic function. *Neuron* 15, 755–760.
- Wils, S., and De Schutter, E. (2009). STEPS: an algorithm for stochastic simulation of reaction-diffusion systems using tetrahedral meshes. In preparation.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 17 September 2008; paper pending published: 11 November 2008; accepted: 09 May 2009; published online: 29 June 2009.
- Citation: Wils S and De Schutter E (2009) STEPS: modeling and simulating complex reaction-diffusion systems with Python. *Front. Neuroinform.* (2009) 3:15. doi:10.3389/neuro.11.015.2009
- Copyright © 2009 Wils and De Schutter. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system

Daniel Brüderle^{1*}, Eric Müller^{1†}, Andrew Davison², Eilif Müller³, Johannes Schemmel¹ and Karlheinz Meier¹

¹ Kirchhoff Institute for Physics, University of Heidelberg, Heidelberg, Germany

² Unité de Neurosciences Intégratives et Computationnelles, CNRS, Gif sur Yvette, France

³ Laboratory of Computational Neuroscience, EPFL, Lausanne, Switzerland

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Bernabe Linares-Barranco, Instituto de
Microelectrónica de Sevilla, Spain
Adrian Whatley, University of Zurich,
Switzerland

*Correspondence:

Daniel Brüderle, Kirchhoff Institute for
Physics, Im Neuenheimer Feld 227,
69120 Heidelberg, Germany.
e-mail: bruederle@kip.uni-heidelberg.de

[†]Daniel Brüderle and Eric Müller have
contributed equally to this work.

Neuromorphic hardware systems provide new possibilities for the neuroscience modeling community. Due to the intrinsic parallelism of the micro-electronic emulation of neural computation, such models are highly scalable without a loss of speed. However, the communities of software simulator users and neuromorphic engineering in neuroscience are rather disjoint. We present a software concept that provides the possibility to establish such hardware devices as valuable modeling tools. It is based on the integration of the hardware interface into a simulator-independent language which allows for unified experiment descriptions that can be run on various simulation platforms without modification, implying experiment portability and a huge simplification of the quantitative comparison of hardware and simulator results. We introduce an accelerated neuromorphic hardware device and describe the implementation of the proposed concept for this system. An example setup and results acquired by utilizing both the hardware system and a software simulator are demonstrated.

Keywords: neuromorphic, VLSI, hardware, software, modeling, computational neuroscience, Python, PyNN

INTRODUCTION

Models of spiking neurons are normally formulated as sets of differential equations for an analytical treatment or for numerical simulation. So-called “neuromorphic” hardware systems represent an alternative approach. In a physical, typically silicon, form they mimic the structure and emulate the function of biological neural networks. Neuromorphic hardware engineering has a tradition going back to the 1980s (Mead, 1989; Mead and Mahowald, 1988), and today an active community is developing analog or mixed-signal VLSI models of neural systems (Ehrlich et al., 2007; Häfliger, 2007; Merolla and Boahen, 2006; Renaud et al., 2007; Schemmel et al., 2007, 2008; Serrano-Gotarredona et al., 2006; Vogelstein et al., 2007).

The main advantage of the physical emulation of neural network models, compared to their numerical simulation, arises from the locally analog and massively parallel nature of the computations. This leads to neuromorphic network models being typically highly scalable and being able to emulate neural networks in real time or much faster, independent of the underlying network size. Often, the inter-chip event-communication bandwidth sets a practical limit on the scaling of network sizes by inter-connecting multiple neural network modules (Berge and Häfliger, 2007; Costas-Santos et al., 2007; Schemmel et al., 2008). Compared to numerical solvers of differential equations which require Von-Neumann-like computer environments, neuromorphic models have much more potential for being realized as miniature embedded systems with low power consumption.

A clear disadvantage is the limited flexibility of the implemented models. Typically, neuron and synapse parameters and the network connectivity can be programmed to a certain degree within limited ranges by controlling software. However, changes to the implemented model itself usually require a hardware re-design,

followed by production and testing phases. This process normally takes several months. Further fundamental differences between hardware and software models will be discussed in the Section “Neuromorphic Hardware”.

Except for the system utilized in this work, all cited neuromorphic hardware projects currently work with circuits operating in biological real-time. This allows interfacing real-world devices such as sensors (Serrano-Gotarredona et al., 2006) or motor controls for robotics, as well as setting up hybrid systems with *in vitro* neural networks (Bontorin et al., 2007). The neuromorphic hardware systems we consider in this article, as described in Schemmel et al. (2007, 2008), possess a crucial feature: they operate at a highly accelerated rate. The device which is currently in operation (Schemmel et al., 2007) (see “The Accelerated Hardware System” for a detailed description) exhibits a speedup factor of 10^5 compared to the emulated biological real time. This opens up new prospects and possibilities, which will be discussed in the Section “Neuromorphic Hardware”.

This computation speed, together with an implementation path towards architectures with low power consumption and very large scale networks (Fieres et al., 2008; Schemmel et al., 2008), makes neuromorphic hardware systems a potentially valuable research tool for the modeling community, where software simulators are more commonplace (Brette et al., 2006; Morrison et al., 2005, 2007). To establish neuromorphic hardware as a useful component of the neural network modelers’ toolbox requires a proof of the hardware system’s biological relevance and its operability by non-hardware-experts.

An approach which can help to fulfil both of these conditions is to interface the hardware system with the simulator-independent language PyNN (Davison et al., 2008) (see “PyNN and NeuroTools”). The PyNN meta-language allows for a unified description of neural

network experiments, which can then be run on all supported backends, e.g. various software simulators or the presented hardware system, without modifying the description itself. Experiment portability, data exchange and unified analysis environments are only some of PyNN's important implications. For neuromorphic devices, this provides the possibility to calibrate and verify the implemented models by comparing any emulated data with the corresponding results generated by established software simulators. Every scientist, who has already used such a simulator with scripting support or with an interpreter interface, will easily learn how to use PyNN. And every PyNN user can operate the presented hardware system without a deeper knowledge of technical device details.

In the Section "Simulator-like Setup, Operation and Analysis", the architecture of a Python (Rossum, 2000) interface to the hardware system, which is the basis for integration into PyNN, will be described in detail. The advantages and problems of the PyNN approach for the hardware system will also be discussed. In the Section "The Interface in Practice", an example of PyNN code for the direct comparison of an experiment run on both the hardware system and a software simulator, including the corresponding results, will be presented.

NEUROMORPHIC HARDWARE

Unlike most numerical simulations of neural network models, analog VLSI circuits operate in the continuous time regime. This avoids possible discretization artifacts, but also makes it impossible to interrupt an experiment at an arbitrary point in time and restart from an identical, frozen network state. Furthermore, it is not possible to perfectly reproduce an experiment because the device is subject to noise, to cross-talk from internal or external signals, and to temperature dependencies (Dally and Poulton, 1998). These phenomena often have a counterpart in the biological specimen, but it is highly desirable to control them as much as possible.

Another major difference between software and hardware models is the finiteness of any silicon substrate. This in principle also limits the software model size, as it utilizes standard computers with limited memory and processor resources, but for neuromorphic hardware the constraints are much more immediate: the number of available neurons and the number of synapses per neuron have strict upper limits; the number of manipulable parameters and the ranges of available values are fixed.

Still, neuromorphic network models are highly scalable at constant speed due to the intrinsic parallelism of their circuit operation. This scalability results in a relative speedup compared to software simulations, which gets more and more relevant the larger the simulated networks become, and provides new experimental possibilities. An experiment can be repeated many times within a short period, allowing the common problem of a lack of statistics, due to a lack of computational power, to be overcome. Large parameter spaces can be swept to find an optimal working point for a specific network architecture, possibly narrowing the space down to an interesting region which can then be investigated using a software simulator with higher precision. One might also think of longer experiments than have so far been attempted, especially long-term learning tasks which exploit synaptic plasticity mechanisms (Schemmel et al., 2007).

THE ACCELERATED HARDWARE SYSTEM

Within the FACETS research project (FACETS, 2009), an interdisciplinary consortium investigating novel computing paradigms by observing and modeling biological neural systems, an accelerated neuromorphic hardware system has been developed. It will be described in this section.

Neuron, Synapse and Connectivity model

The FACETS neuromorphic mixed-signal VLSI system has been described in detail in recent publications (Schemmel et al., 2006, 2007). Implemented is a leaky integrate-and-fire neuron model with conductance-based synapses, designed to exhibit a linear correspondence with existing conductance-based modeling approaches (Destexhe et al., 1998). The chip was built on a single 25 mm² die using a standard 180 nm CMOS process. It models networks of up to 384 neurons and the temporal evolution of the weights of 10⁵ synapses. The system can be operated with an acceleration factor of up to 10⁵ while recording the neural action potentials with a temporal resolution of approximately 0.3 ns, which corresponds to 30 μs in biological time.

The neuron circuits are designed such that the emulated membrane potential $V(t)$ is determined by the following differential equation for a conductance-based integrate-and-fire neuron:

$$-C_m \frac{dV}{dt} = g_m(V - E_l) + \sum_j p_j(t) g_j(t)(V - E_e) + \sum_k p_k(t) g_k(t)(V - E_i) \quad (1)$$

where C_m represents the total membrane capacitance. The first term on the right hand side, the so-called leak current, models the contribution of the different ion channels that determine the potential E_l the membrane will eventually reach if no other currents are present. The synapses use different reversal potentials, E_i and E_e , to model inhibitory and excitatory ion channels. The index j in the first sum runs over all excitatory synapses while the index k in the second sum covers the inhibitory ones. The activation of individual synapses is controlled by the synaptic opening probability $p_{j,k}(t)$ (Dayan and Abbott, 2001). The synaptic conductance $g_{j,k}$ is modeled as a product of the synaptic weight $\omega_{j,k}(t)$ and a maximum conductance $g_{j,k}^{\max}(t)$. The neuron emits a spike if a threshold voltage V_{th} is exceeded, after which the membrane potential is forced to a reset voltage V_{reset} and then released back into the influence of excitatory, inhibitory and leakage mechanisms. The weights are modified by a long-term plasticity algorithm (Schemmel et al., 2007) and thus can vary slowly with time. **Table 1** summarizes the most important hardware parameters, with their counterparts in the biological model, their available ranges and uncertainties.

Each chip is divided into two network blocks of 192 neurons each, and each block can receive 256 different input channels. Each input channel into a block can be configured to receive either a feedback signal from one specific neuron within the same block, a feedback signal from the opposite block, or an externally generated signal, for example from some controlling software. Every neuron within the block can be connected to every input channel via a configurable synapse. Synaptic time constants and the values for g^{\max} are shared for every input channel, while the connection weights

Table 1 | The most important hardware model parameters, the type of physical quantity used for their implementation, their configurability and an estimation of uncertainty. The first four columns show their typical biological interpretation and the resulting value ranges. The translation between both domains depends on the chosen speedup and the desired biological parameter value ranges. The given estimations (some being educated guesses) of configuration uncertainty reflect the current state of available methods to measure, to adjust or to calibrate the values, and may not necessarily reflect hardware limitations. The uncertainty of E_e is load-dependent, the relation is not yet sufficiently analyzed.

Biological Interpretation				Hardware parameter implementation		
Param	Unit	Min	Max	Physical quantity	Configurable	Estimation of uncertainty (%)
C_m	nF	0.2	0.2	Capacitance	No	10
G_i	nS	20	40	Current	Yes	10
E_i	mV	-80	-55	Voltage	Yes	2
E_i	mV	-80	-55	Voltage	Yes	2
E_e	mV	-80	20	Voltage	Yes	Unknown
V_{th}	mV	-80	-55	Voltage	Yes	5
V_{reset}	mV	-80	-55	Voltage	Yes	10
τ_{syn}	ms	30	50	Current	Yes	25
g^{max}	nS	1	100	Current	Yes	25

can be set between 0 nS and g^{max} with a four bit resolution for each individual connection.

Although the free parameter space is already large, the model flexibility is clearly limited, especially in terms of its inter-neuron connectivity. Based on the experience acquired with the prototype chip described above, a wafer-scale integration¹ system (Fieres et al., 2008; Schemmel et al., 2008) with up to 1.8×10^5 neurons and 4×10^7 synapses per wafer is currently under development. It will be operated with a speedup factor of up to 10^4 and will provide a much more flexible and powerful connectivity infrastructure.

Support framework

In order to give life to such a piece of manufactured neuromorphic silicon, an intricate framework of various pieces of custom-made support hardware and software layers has to be deployed, which has previously been reported on. The chip is mounted on a carrier board called Nathan (Fieres et al., 2004; Grübl, 2007, Chapter 3) which also holds, among other components, an FPGA for direct communication control and some RAM memory modules for storing input and output data. Up to 16 of these carrier boards can be placed on a so-called backplane (Philipp et al., 2007), which itself is connected to a host PC via a PCI-based FPGA card (Schürmann et al., 2002).

The connection from chip to computer via the PCI card allows the configuration of the hardware, the definition and application of spike stimuli and the recording of spiking activity from within the network. Analog sub-threshold data can only be acquired via an oscilloscope², which is connected to pins that can output selectable membrane potentials. Via a network connection, the information from this oscilloscope can be read and integrated into the software running on the host computer (see **Figure 1** for a setup schematic).

Both an FPGA on the backplane and those on the carrier boards are programmed and configured with dedicated code.

¹A silicon wafer which will not be cut into single chips as is usual, but left in one piece. Further post-processing steps will interconnect the disjoint reticles on the wafer, resulting in a highly configurable silicon neural network model of unique dimensions.

²Currently: LeCroy WaveRunner 44Xi.

Communication with the PCI board utilizes a specific device driver and a custom-made protocol (Philipp, 2008, Chapter 2.2.4). Multi-user access is realized via userspace daemon multiplexing connections to different chips while encapsulating control commands and data from multiple users in POSIX Message Queues (IEEE, 2004). Data transfer from and to the oscilloscope is based on TCP/IP sockets (Braden, 1989; LeCroy, 2005). Interconnecting multiple chips in order to set up larger networks will be possible soon (Philipp et al., 2007).

SIMULATOR-LIKE SETUP, OPERATION AND ANALYSIS

As proposed in the introduction, attracting neuroscience experts into the field of neuromorphic engineering is essential for the establishment of hardware devices as modeling tools. Neuroscience expertise has to be consulted not only during the design process, but also, and especially, after manufacturing, when it comes to verifying the device's biological relevance. This implies a whole set of requirements for the software which provides the user interface to the hardware.

If the system is to be operated by scientists from fields other than neuromorphic engineering, the software must hide as many hardware-specific details as possible. We propose that it should provide basic control mechanisms similar to typical interfaces of pure software simulators, i.e. an interpreter for interactive operation and scripting. Parameters and observables should be given in biological dimensions and follow a biological nomenclature. Moreover, drawing the attention of the neuroscience community to neuromorphic hardware can be strongly facilitated by the possibility of porting existing software simulation setups to the hardware with little effort.

Multiple projects and initiatives provide databases and techniques for sharing or unifying neuroscientific modeling code, see for example the NeuralEnsemble initiative (Neural Ensemble, 2009), the databases of Yale's SenseLab (Hines et al., 2004) or the software database of the International Neuroinformatics Coordination Facility (INCF Software Database, 2009). Creating a bridge from the hardware interface to these pools of modeling experience will provide the important possibility of formulating transparent tests,

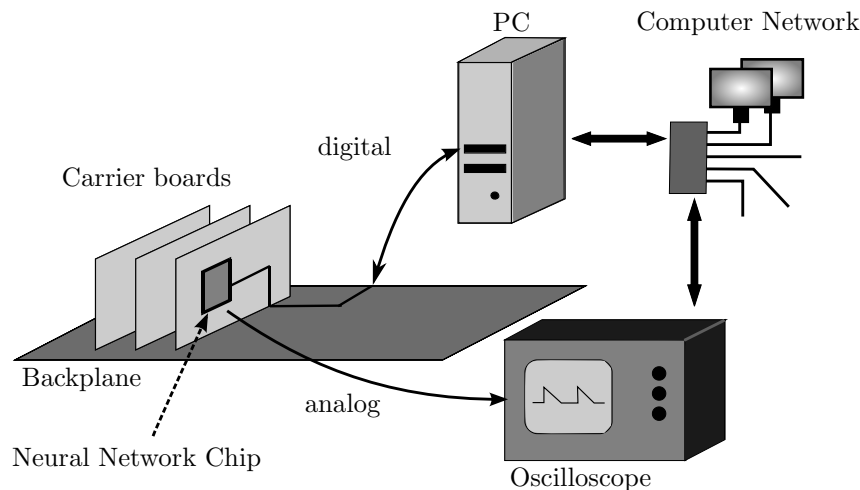


FIGURE 1 | Schematic of the accelerated FACETS hardware

system framework. Via a digital connection, software running on the host computer can control the parameters of any neural network chip mounted on a carrier board on the communication backplane. It can

stimulate the network with externally generated spikes and can record spikes generated on the chip. Analog sub-threshold information acquired with an oscilloscope can be integrated into the software via a network connection.

benchmarks and requests that will boost further hardware development and its establishment as a modeling tool.

Most software simulators for spiking neuron models come with an interpreter interface for programming, experiment setup and control. For example, NEURON (Hines and Carnevale, 2006; Hines et al., 2009) provides an interpreter called Hoc, NEST (Diesmann and Gewaltig, 2002; Eppler et al., 2008; Gewaltig and Diesmann, 2007) comes with a stack-based interface called SLI, and GENESIS (Bower and Beeman, 1998) has a different custom script language interpreter also called SLI. Both NEURON and NEST also provide Python (Rossum, 2000) interfaces, as do the PCSIM (PCSIM, 2009; Pecevski et al., 2009), Brian (Goodman and Brette, 2008) and MOOSE (Ray and Bhalla, 2008) simulators. Facilitating the usage of neuromorphic hardware for modelers means providing them with an interface similar to these existing ones. But there are further requirements arising from hardware specific issues.

TECHNICAL REQUIREMENTS

As shown in the Section “Support Framework”, operating the presented neuromorphic hardware system involves multiple devices and mechanisms, e.g. Message Queue communication with a user-space daemon accessing a PCI board, TCP/IP socket connection to an oscilloscope, software models that control the operation of the backplane, the carrier board and the VLSI chip itself, and high-level software layers for experiment definition. On the software side, this multi-module system utilizes C, C++ and Python, and multiple developers from different institutions are involved, applying various development styles such as object-oriented programming, reflective programming or sequential driver code. The software has to follow the ongoing system development, including changing and improving FPGA controller code and hardware revisions with new features.

This complexity and diversity argues strongly for a top-level software framework, which has to be capable of efficiently gluing all modules together, supporting object-oriented and reflective struc-

tures, and providing the possibility of rapid prototyping in order to quickly adapt to technical developments at lower levels.

One further requirement arises: the speedup of the hardware system can be exploited by an interactive, possibly intuition-guided work flow which allows the exploration of parameters with immediate feedback of the resulting changes. This implies the wish to have the option of a graphical interface on top of an arbitrary experiment description.

EXISTING INTERFACES

Descriptions in the literature of existing software interfaces to neuromorphic hardware are very rare. In Merolla and Boahen (2006), the existence and main features of a GUI for the interactive operation of a specific neuromorphic hardware device are mentioned.

Much more detailed software interface reports are found in Dante et al. (2005). They describe a framework which allows exchange of AER³ data between hardware and software while experiments are running. The framework includes a dedicated PCI board which is connected to the neuromorphic hardware module and which can be interfaced to Linux systems by means of a device driver. A C-library layered on top of this driver is available. Using this, a client-server architecture has been implemented which allows the on-line operation of the hardware from within the program MATLAB. The use of MATLAB implies interpreter-based usage, scripting support, the possible integration of C and C++ code, optional graphical front-end programming and strong numerical support for data analysis. Hence, most of the requirements listed so far are satisfied. Nevertheless, the framework is somewhat stand-alone and does not facilitate the transfer of existing software models to the hardware.

In Oster et al. (2005), an automatically generated graphical front-end for the manual tuning of hardware parameters is presented, including the convenient storing and loading of configurations.

³Address Event Representation.

Originally, a similar approach was developed for the hardware system utilized here, too (Brüderle et al., 2007). Manually defining parts of the enormous parameter space provided by such a chip via sliders and check-boxes can be useful for intuition-guided hardware exploration and circuit testing, but it turns out to be rather impractical for setting up large network experiments as usually performed by computational neuroscientists.

CHOOSING A PROGRAMMING LANGUAGE

Except for the convenient portability of existing experiment setups, an interface to the neuromorphic hardware system based on the programming language Python solves all of the requirements stated in the Sections “Importance of the Software Interface” and “Technical Requirements”, especially the hardware-specific ones. Python is an interpreter-based language with scripting support, thus it is able to provide a software-simulator-like interface. It can be efficiently connected to C and C++, for example via the package Boost.Python (Abrahams and Grosse-Kunstleve, 2003). Python supports sequential, object-oriented and reflective programming and it is widely praised for its rapid prototyping. Due to the possibility for modular code structure and embedded documentation, it has a high maintainability, which is essential in the context of a quickly evolving project with a high number of developers.

In addition to its strengths for controlling and interconnecting lower-level software layers, it can be used to write efficient post-processing tools for data analysis and visualization, since a wide range of available third-party packages offers a strong foundation for scientific computing (Jones et al., 2001; Langtangen, 2008; Oliphant, 2007), plotting (Hunter, 2007) and graphics (Lutz, 2001, Chapter 8; Summerfield, 2008). Hence, a Python interface to the hardware system would already greatly facilitate modeler adoption.

Still, the possibility of directly transferring existing experiments to the hardware is even more desirable; a unified meta-language usable for both software simulators and the hardware could achieve that. Thus, the existence of the Python-based, simulator-independent modeling language PyNN (see PyNN and NeuroTools) was the strongest argument for utilizing Python as a hardware interface, because the subsequent integration of this interface into PyNN depended on the possibility of accessing and controlling the hardware via Python.

Possible alternatives to Python as the top layer language for the hardware interface have been considered and dropped for different reasons. For example, C++ requires a good understanding of memory management, it has a complex syntax, and, compared to interpreted languages, has slower development cycles. Interpreter-based languages such as Perl or Ruby also provide plotting functionality, numerical packages (Berglihn, 2006; Glazebrook and Economou, 1997) and techniques to wrap C/C++ code, but eventually Python was chosen because it is considered to be easy to learn and to have a clean syntax.

PYNN AND NEUROTOOLS

The advantages of Python as an interface and programming language are not limited to hardware back-ends. For the software simulators NEURON, NEST, PCSIM, MOOSE and Brian, Python interfaces exist. This provides the possibility of creating a Python-

based, simulator-independent meta-language on top of all these back-ends. In the context of the FACETS project, the open-source Python module PyNN has been developed which implements such a unified front-end (see Davison, 2009; Davison et al., 2008).

PyNN offers the possibility of porting existing experiments between the supported software simulators and the FACETS hardware and thus to benchmark and verify the hardware model. Furthermore, on top of PyNN, a library of analysis tools called NeuroTools (2009) is under development, exploiting the possibility of a unified work flow within the scope of Python. Experiment description, execution, result storage, analysis and plotting can be all done from within the PyNN and NeuroTools framework. Independent of the used back-end, all these steps have to be written only once and can then be run on each platform without further modifications.

Especially since the operation of the accelerated hardware generates large amounts of data at high iteration rates, a sophisticated analysis tool chain is necessary. For the authors, as well as for every possible PyNN user, making use of the unified analysis libraries based on the PyNN standards (e.g. NeuroTools) avoids redundant development and debugging efforts. This benefit is further enhanced by other third-party Python modules, like numerical or visualization packages.

INTERFACE ARCHITECTURE

The complete software framework for interfacing the FACETS hardware is structured as follows: Various C++ classes encapsulate the functionality of the neural network chip itself, of its configuration parameter set, of the controller implemented on the carrier board FPGA, and of the communication protocol between the host software and this controller. There is a stand-alone daemon written in C++ which provides the transport of data via the PCI card. It utilizes a device-driver which is available for Linux systems. Furthermore, there is a C++ class which encapsulates the TCP/IP Socket communication with the oscilloscope.

The Boost.Python library (Boost.Python, 2003) is used to bind C++ classes and functions to Python. An instructive outline of the wrapping technique used can be found in Abrahams and Grosse-Kunstleve (2003).

On top of these Python bindings, a pure Python framework called PyHAL⁴ (Brüderle et al., 2007) provides classes for neurons, synapses and networks. All these classes have model parameters in biological terminology and dimensions, and their constructors impose no hardware specific constraints.

The main functionality of PyHAL is encapsulated by a hardware access class which implements the exchange layer between these higher-level objects and the low-level C++ classes exposed to Python via Boost. The hardware access layer performs the translation from biological parameters like reversal potentials, leakages, synaptic time constants and weights to the available set of hardware configuration parameters. This set consists of discrete integers, for example for the synaptic weights, and of analog values for currents and voltages. Some of these parameters do have a direct biological counterpart, some do not. For example, neuron voltage parameters like reversal potentials are mapped linearly to the available hardware membrane potential range of approximately 0.6–1.4 V,

⁴Python Hardware Abstraction Layer.

while membrane leakage conductances and synaptic time constants have to be translated into currents.

The translation layer also performs the transformation from biological to hardware time domain and back. Furthermore, all hardware-specific constraints, like the limited number of possible neurons or connections, the finite parameter ranges and the synaptic weight discretization, are incorporated in this hardware access class, generating instructive warnings or error messages in case of constraint violations.

Since the PyHAL framework is all Python code, it provides the desired interpreter-based interface to the hardware, corresponding to comparable Python interfaces to, for example, NEST or PCSIM. Also, as for these software simulators, a module for the integration of this interface into the meta-language PyNN has been implemented. **Figure 2** shows a schematic of the complete software framework with its most important components.

Thanks to this integration, all higher-level PyNN concepts like populations and inter-population projections plus the analysis and visualization tools developed on top of PyNN are now available for the hardware system.

Still, the integration of the hardware interface into PyNN also raises problems. Some of the PyNN API function arguments are specific to software simulators. In the hardware context, they have to be either ignored or be given a hardware-specific interpretation. For example, the PyNN function `setup` has an argument called `timestep`, which for pure software back-ends determines the numerical integration time step. In the PyNN module for the continuously operating hardware, this argument defines the temporal resolution of the oscilloscope for membrane potential recordings. Furthermore, the strict constraints regarding neuron number, connectivity and possible parameter values require an additional software effort, i.e. checking for violations and providing the messages mentioned above. PyNN does not yet sufficiently support fast and statistics-intensive parameter space searches with differential formulations of the changes from step to step, which

will be needed to optimize the exploitation of hardware specific advantages.

Without having access to the real hardware system, it is of course not possible to use the PyNN hardware module, hence it is not available for download. Still, it is planned to publicly provide a modified module on the PyNN website (Davison, 2009) which allows testing of PyNN scripts intended to be run on the hardware, i.e. to get back all warnings or error messages which might occur with the real system. With such a mapping test module, scripts can be prepared offline for a later, optimized hardware run.

THE INTERFACE IN PRACTICE

To demonstrate the usage and functionality of the PyNN interface, a simple example setup is given in the following. **Listing 1** shows the experiment described in PyNN, which is then executed both on the hardware system and using the software simulator NEST. A network consisting of 80 excitatory and 20 inhibitory neurons is created. The inhibitory sub-population is fed back into the network randomly with a probability of 0.5 for each possible inhibitory-to-excitatory connection. 160 excitatory and 40 inhibitory Poisson spike trains are randomly connected to the network with the same probability of 0.5 for each possible train-to-neuron connection.

Figure 3 shows a schematic of the implemented network architecture.

The maximum synaptic conductance g^{\max} is 0.5 nS for excitatory and 1.6 nS for inhibitory connections. The output spikes of eight neurons are recorded, and the average firing rate of these eight neurons over a period of 5 s of biological time is determined.

In line 1, the PyNN back-end NEST is chosen. In order to utilize the hardware system, the only necessary change within this script is to replace line 1 by `from pyNN.hardware.stage1 import *`, all the rest remains the same. From lines 4 to 9, the population sizes, the numbers of external stimuli, and the synaptic weights are set. In lines 11–17, the neuron parameters are defined. Lines 19

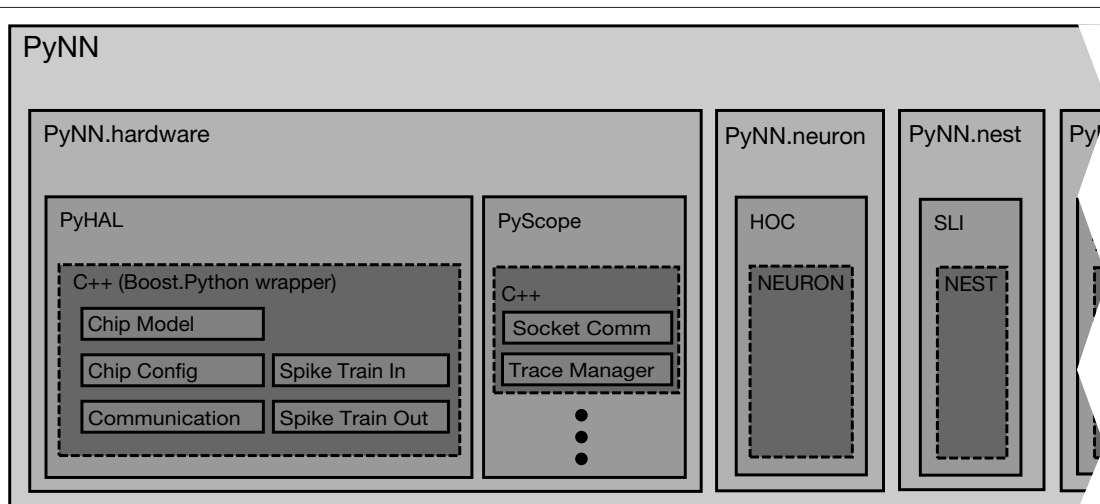


FIGURE 2 | Schematic of the software framework for the operation of the hardware system. It is integrated into the Python-based, simulator-independent language PyNN, which also supports back-ends like NEURON, NEST and more.

The module for the hardware back-end consists of Python-based sub-modules for the digital and analog access to the chip. Each of those wrap the functionality of lower-level C++ layers, which are described in more detail in the text.

```

1  from pyNN.nest2 import *
2  # OR: from pyNN.hardware.stage1 import *
3
4  numInhNeurons = 20
5  numExcNeurons = 80
6  numInhInputs  = 40
7  numExcInputs  = 160
8  w_exc = 0.0005 # uS
9  w_inh = 0.0016 # uS
10
11 neuronParams = {      'v_reset'      : -80.0,      # mV
12                       'e_rev_I'      : -75.0,      # mV
13                       'v_rest'       : -70.0,      # mV
14                       'v_thresh'     : -57.0,      # mV
15                       'g_leak'       : 20.0,       # nS
16                       'tau_syn_E'    : 30.0,       # ms
17                       'tau_syn_I'    : 30.0      } # ms
18
19 inputParameters = { 'rate'           : 5.0,         # Hz
20                    'duration'       : 5000        } # ms
21
22 setup(timestep=0.1)
23
24 n_inh = create(IF_facets_hardware1, neuronParams, n=numInhNeurons)
25 n_exc = create(IF_facets_hardware1, neuronParams, n=numExcNeurons)
26 net   = n_exc + n_inh
27
28 i_exc = create(SpikeSourcePoisson, inputParameters, n=numExcInputs)
29 i_inh = create(SpikeSourcePoisson, inputParameters, n=numInhInputs)
30
31 connect(i_exc, net, weight=w_exc, synapse_type='excitatory', p=0.5)
32 connect(i_inh, net, weight=w_inh, synapse_type='inhibitory', p=0.5)
33
34 connect(n_inh, net, weight=w_inh, synapse_type='inhibitory', p=0.5)
35
36 record(net[0:8], 'spikes.dat')
37 record_v(net[0], 'membrane.dat')
38
39 run(5000) # duration in ms
40 end()

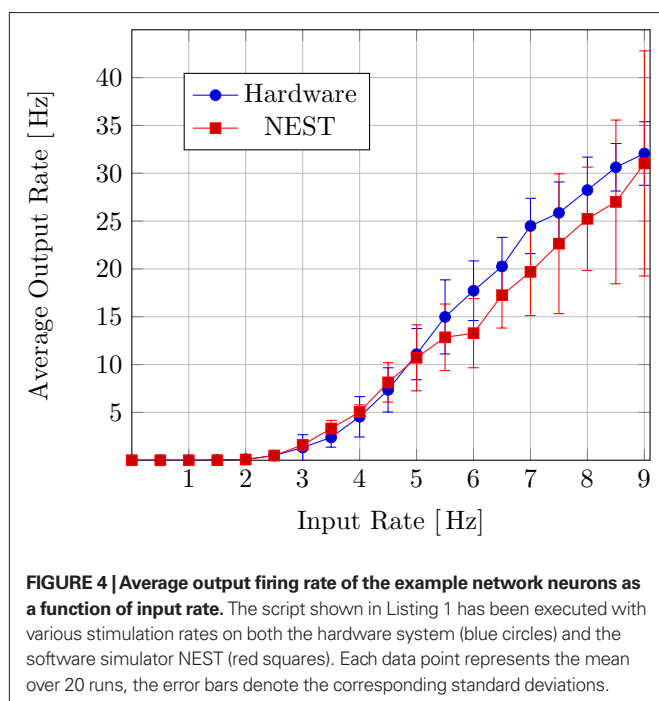
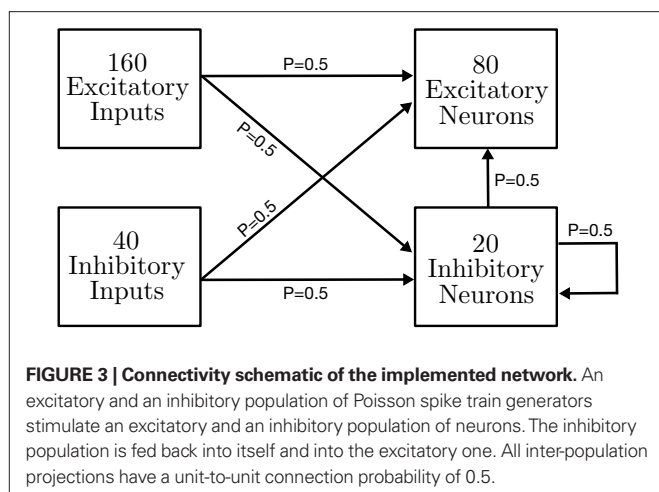
```

LISTING 1 | PyNN Example Script. For detailed explanation see text.

and 20 determine the rate and duration of the Poisson spike train stimuli. In line 22, PyNN is initialized, the numerical integration step size of 0.1 ms is passed. If the hardware back-end is chosen, no discrete step size is utilized due to the time continuous dynamics in its analog network core, and the function argument is used instead to determine the time resolution of the oscilloscope, if connected. In lines 24 and 25, the excitatory and inhibitory neurons are created, with the neuron parameters and the size of the populations as the second and the third arguments.

The first argument, `IF_facets_hardware1`, specifies the neuron type to be created. For the hardware system, no other neuron

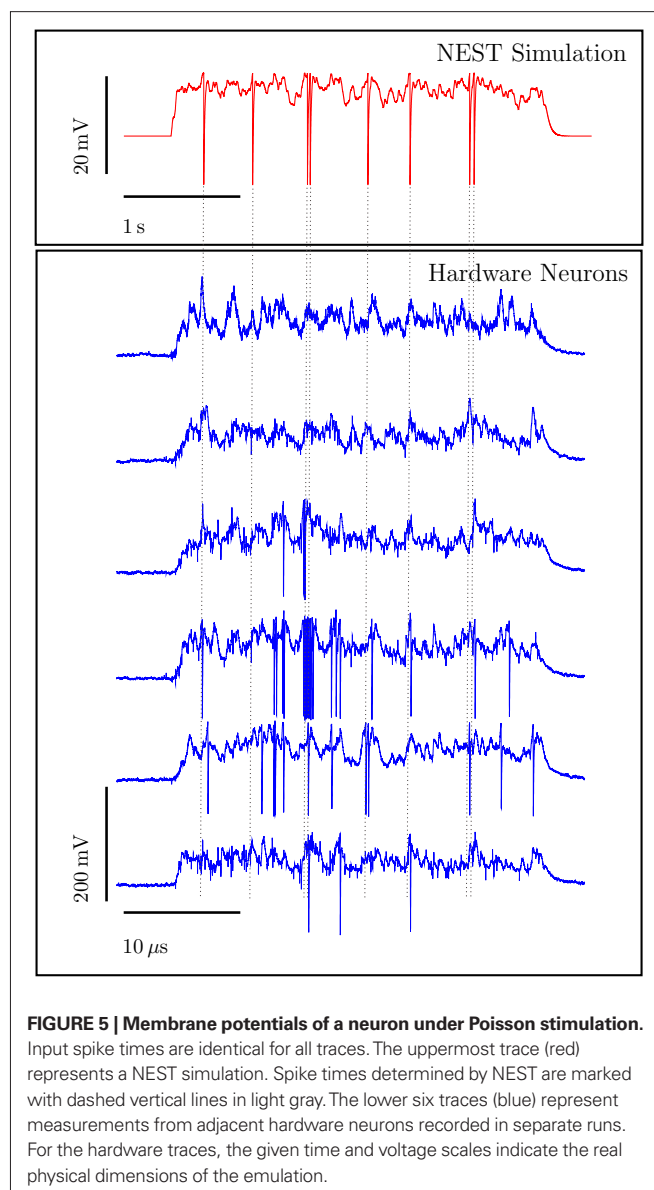
type is possible. For the NEST back-end, the neuron type determines parameter values for e.g. C_m , which are fixed to resemble the hardware. Line 26 concatenates the two populations. In lines 28 and 29, the Poisson spike sources are generated, passing the type of source, the previously defined parameters and the desired number. From lines 31 to 34, the neurons and spike generators are interconnected. The arguments of the connect command specify first a list of sources, then a list of targets, followed by the synaptic weights, the synapse types and finally by the probability with which each possible pairing of source and target objects is actually connected. The recording of the spikes of eight neurons and of one membrane



potential is prepared in lines 36 and 37 (not all neurons, due to a bug in the current hardware revision). In line 39, the experiment is executed for a duration of 5000 ms. Line 40 defines the end of the script, and deals with writing recorded values to file.

The experiment was run both on the FACETS hardware system and using the software simulator NEST. The firing rate of the stimulating Poisson spike trains was varied from 0 to 9 Hz in steps of 0.5 Hz, and for each rate the experiment was repeated 20 times with different random number generator seeds. **Figure 4** shows the resulting average output firing rates.

The firing rates measured on both back-ends exhibit a qualitative and, within the observed fluctuations, quantitative correspondence. For both NEST and the hardware system, the onset of firing activity occurs at the same level of synaptic stimulation. The small but seemingly systematic discrepancy for higher output rates indicates that for the NEST simulation the inhibitory feedback has a slightly



stronger impact on the network activity than on the hardware platform. The firing rate does not reflect dynamic properties like firing regularity or synchrony, which might be interesting for the estimation of possible differences in network dynamics due to the limited precision of hardware parameter determination or due to electronic noise. With PyNN, studies like these have now become possible, but go beyond the scope of this paper.

To give an impression of the inhomogeneities of a hardware substrate and of the noise a typical hardware membrane is exposed to, a second measurement is shown. A single neuron receives 80 excitatory and 20 inhibitory Poisson spike trains with 2.5 Hz each. It is connected to these stimuli with the same synaptic weights as in the setup described above, but gets no feedback from other neurons. The spike sources fire for 4 s, with a silent phase of 0.5 s before and after. Using a single PyNN description, the identical setup with identical spike times and identical connectivity can be deployed for both NEST and the hardware system. **Figure 5** shows

the resulting membrane potential trace simulated by NEST and the membrane potentials acquired from six adjacent neurons on the neuromorphic hardware. For the hardware traces, the unprocessed time and voltage scales are given as measured on the chip in order to illustrate the accelerated and physical nature of the neuromorphic model. The PyHAL framework automatically performs a translation of these dimensions into their biological equivalents.

The constant noise level in the hardware traces can be best observed during the phases with no external stimulation. This noise is a superposition of the noise actually occurring within the neuron circuits and the noise being added by the recording devices. The differences from hardware neuron to hardware neuron represent mainly device fluctuations on the transistor level, which strongly dominate time-dependent influences like temperature-dependent leakages or an unstable power supply. Counterbalancing these fixed-pattern effects with calibration methods is work in progress.

DISCUSSION

Today, the communities of computational neuroscientists and neuromorphic engineers work rather in parallel instead of benefitting from each other. We believe that closing this gap will boost the development, the usability and the number of application fields of neuromorphic systems, including the establishment of such devices as valuable modeling tools that will contribute to the understanding of neural information processing. Based on this motivation, we have described a set of requirements that a software interface for a neuromorphic system should fulfill.

REFERENCES

- Abrahams, D., and Grosse-Kunstleve, R. W. (2003). Building Hybrid Systems with Boost.Python. Available at: <http://www.boostpro.com/writing/bpl.pdf>.
- Berge, H. K. O., and Häfliger, P. (2007). High-speed serial AER on FPGA. In *ISCAS (IEEE)*, pp. 857–860.
- Berglin, O. T. (2006). RNUM Website. Available at: <http://rnum.rubyforge.org>.
- Bontorin, G., Renaud S., Garenne, A., Alvado, L., Le Masson, G., and Tomas, J. (2007). A real-time closed-loop setup for hybrid neural networks. In *Proceedings of the 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS2007)*.
- Boost.Python. (2003). Version 1.34.1 Website. Available at: http://www.boost.org/doc/libs/1_34_1/libs/python.
- Bower, J. M., and Beeman D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the General NEural Simulation System*, 2nd Edn. New York, Springer-Verlag. ISBN 0387949380.
- Braden, R. T. (1989). RFC 1122: Requirements for Internet Hosts—Communication Layers. Available at: <ftp://ftp.internic.net/rfc/rfc1122.txt>.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Jr., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., El Boustani, S., and Destexhe, A. (2006). Simulation of Networks of Spiking Neurons: A Review of Tools and Strategies. Available at: <http://arxiv.org/abs/q-bio.NC/0611089>.
- Brüderle, D., Grübl, A., Meier, K., Mueller, E., and Schemmel, J. (2007). A software framework for tuning the dynamics of neuromorphic silicon towards biology. In *Proceedings of the 2007 International Work-Conference on Artificial Neural Networks*, Vol. LNCS 4507 (Berlin, Springer Verlag), pp. 479–486.
- Costas-Santos, J., Serrano-Gotarredona, T., Serrano-Gotarredona, R., and Linares-Barranco, B. (2007). A spatial contrast retina with on-chip calibration for neuromorphic spike-based AER vision systems. *IEEE Trans. Circuits Syst.* 54, 1444–1458.
- Dally, W. J., and Poulton, J. W. (1998). *Digital Systems Engineering*. Cambridge, Cambridge University Press. ISBN 0-521-59292-5.
- Dante, V., Del Giudice, P., and Whatley, A. M. (2005). Hardware and software for interfacing to address-event based neuromorphic systems. *Neuromorphic Eng.* 2, 5–6.
- Davison, A. (2009). PyNN – A Python Package for Simulator-Independent Specification of Neuronal Network Models. Available at: <http://www.neuralensemble.org/PyNN>.
- 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. *Front. Neuroinform.* 2, 11. doi: 10.3389/neuro.11.011.2008.
- Dayan, P., and Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, The MIT Press. ISBN 0-262-04199-5.
- Destexhe, A., Contreras, D., and Steriade, M. (1998). Mechanisms underlying the synchronizing action of corticothalamic feedback through inhibition of thalamic relay cells. *J. Neurophysiol.* 79, 999–1016.
- Diesmann, M., and Gewaltig, M.-O. (2002). NEST: an environment for neural systems simulations. In *Forschung und wissenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, Vol. 58, GWDG-Bericht, Theo Plesser and Volker Macho, eds (Göttingen, Ges. für Wiss. Datenverarbeitung), pp. 43–70.
- Ehrlich, M., Mayr, C., Eisenreich, H., Henker, S., Srowig, A., Grübl, A., Schemmel, J., and Schüffny, R. (2007). Wafer-scale VLSI implementations of pulse coupled neural networks. In *Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems*.
- Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2, 12. doi: 10.3389/neuro.11.012.2008.
- FACETS (2009). Fast Analog Computing with Emergent Transient States, Project Homepage. Available at: <http://www.facets-project.org>.
- Fieries, J., Grübl, A., Philipp, S., Meier, K., Schemmel, J., and Schürmann, F. (2004). A platform for parallel operation of VLSI neural networks. In *Proceedings of the 2004 Brain Inspired Cognitive Systems Conference*, University of Stirling, Scotland.
- Fieries, J., Schemmel, J., and Meier, K. (2008). Realizing biological spiking network models in a configurable wafer-scale hardware system. In *Proceedings of the 2008 International Joint Conference on Neural Networks*.
- Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430.

- Glazebrook, K., and Economou, F. (1997). PDL: The Perl Data Language. Dr. Dobb's Journal. Available at: <http://www.ddj.com/184410442>.
- Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2, 5. doi: 10.3389/neuro.11.005.2008.
- Grübl, A. (2007). VLSI Implementation of a Spiking Neural Network. PhD Thesis, Heidelberg, Ruprecht-Karls-University. Available at: <http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1788>. Document No. HD-KIP 07-10.
- Häfliger, P. (2007). Adaptive WTA with an analog VLSI neuromorphic learning chip. *IEEE Trans. Neural Netw.* 18, 551–572.
- Hines, M. L., and Carnevale, N. T. (2006). The NEURON Book. Cambridge, Cambridge University Press. ISBN 978-0521843218.
- Hines, M. L., Davison, A. P., and Muller, E. (2009). NEURON and Python. *Front. Neuroinform.* 3, 1. doi: 10.3389/neuro.11.001.2009.
- Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). ModelDB: a database to support computational neuroscience. *J. Comput. Neurosci.* 17, 7–11.
- Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *IEEE Comput. Sci. Eng.* 9, 90–95.
- IEEE (2004). Standard for Information Technology – Portable Operating System Interface (POSIX). Shell and Utilities. Technical Report, IEEE. Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309816.
- INCF Software Database (2009). Website. Available at: <http://software.incf.net>.
- Jones, E., Oliphant, T., Peterson, P. et al. (2001). SciPy: Open Source Scientific Tools for Python. Available at: <http://www.scipy.org/>.
- Langtangen, H. P. (2008). Python Scripting for Computational Science, 3rd Edn. (Berlin, Springer). ISBN 978-3-540-73915-9.
- LeCroy (2005). X-Stream Oscilloscopes–Remote Control Manual. Technical Report Revision D, New York, LeCroy Corporation. Available at: <http://lecroygmbh.com>.
- Lutz, M. (2001). Programming Python: Object-Oriented Scripting. Sebastopol, O'Reilly & Associates, Inc. ISBN 0596000855.
- Mead, C. A. (1989). Analog VLSI and Neural Systems. Reading, Addison Wesley.
- Mead, C. A., and Mahowald, M. A. (1988). A silicon model of early visual processing. *Neural Netw.* 1, 91–97.
- Merolla, P. A., and Boahen, K. (2006). Dynamic computation in a recurrent network of heterogeneous silicon neurons. In Proceedings of the 2006 IEEE International Symposium on Circuits and Systems.
- Morrison, A., Aertsen, A., and Diesmann, M. (2007). Spike-timing-dependent plasticity in balanced random networks. *Neural Comput.* 19, 1437–1467.
- Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.
- Neural Ensemble (2009). Website. Available at: <http://neuralensemble.org>.
- NeuroTools (2009). Website. Available at: <http://neuralensemble.org/trac/NeuroTools>.
- Oliphant, T. E. (2007). Python for scientific computing. *IEEE Comput. Sci. Eng.* 9, 10–20.
- Oster, M., Whatley, A. M. Liu, S.-C., and Douglas, R. J. (2005). A hardware/software framework for real-time spiking systems. In Proceedings of the 2005 International Conference on Artificial Neural Networks.
- PCSIM (2009). Website. Available at: <http://www.lsm.tugraz.at/pcsims/>.
- Pecevski, D. A., Natschlager, T., and Schuch, K. N. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with python. *Front. Neuroinform.* 3, 11. doi: 10.3389/neuro.11.011.2009.
- Philipp, S. (2008). Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks. PhD Thesis, Heidelberg, Ruprecht-Karls Universität.
- Philipp, S., Grübl, A., Meier, K., and Schemmel, J. (2007). Interconnecting VLSI Spiking Neural Networks Using Isochronous Connections. In Proceedings of the 9th International Work-Conference on Artificial Neural Networks, Vol. LNCS 4507 (Berlin, Springer Verlag), pp. 471–478.
- Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2, 6. doi: 10.3389/neuro.11.006.2008.
- Renaud, S., Tomas, J., Bornat, Y., Daouzli, A., and Saighi, S. (2007). Neuromimetic ICs with analog cores: an alternative for simulating spiking neural networks. In Proceedings of the 2007 IEEE Symposium on Circuits and Systems.
- Rossum, G. V. (2000). Python Reference Manual: February 19, 1999, Release 1.5.2. iUniverse, Incorporated. ISBN 1583483748.
- Schemmel, J., Brüderle, D., Meier, K., and Ostendorf, B. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems, IEEE Press.
- Schemmel, J., Fieries, J., and Meier, K. (2008). Wafer-scale integration of analog neural networks. In Proceedings of the 2008 International Joint Conference on Neural Networks.
- Schemmel, J., Grübl, A., Meier, K., and Mueller, E. (2006). Implementing synaptic plasticity in a VLSI spiking neural network model. In Proceedings of the 2006 International Joint Conference on Neural Networks. IEEE Press.
- Schürmann, F., Hohmann, S., Schemmel, J., and Meier, K. (2002). Towards an artificial neural network framework. In Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware, A. Stoica, J. Lohn, R. Katz, D. Keymeulen, and R. S. Zebulum, eds (Los Alamitos, CA, IEEE Computer Society), pp. 266–273.
- Serrano-Gotarredona, R., Oster, M., Lichtsteiner, P., Linares-Barranco, A., Paz-Vicente, R., Gómez-Rodríguez, F., Riis, H. K., Delbrück, T., Liu, S. C., Zahnd, S., Whatley, A. M., Douglas, R. J., Häfliger, P., Jimenez-Moreno, G., Cività, A., Serrano-Gotarredona, T., Acosta-Jiménez, A., and Linares-Barranco, B. (2006). AER building blocks for multi-layer multi-chip neuromorphic vision systems. In Advances in Neural Information Processing Systems 18, Y. Weiss, B. Schölkopf, and J. Platt, eds (Cambridge, MIT Press), pp. 1217–1224.
- Summerfield, M. (2008). Rapid GUI Programming with Python and Qt. Prentice Hall, Upper Saddle River, NJ, ISBN 0132354187.
- Vogelstein, R. J., Mallik, U., Vogelstein, J. T., and Cauwenberghs, G. (2007). Dynamically reconfigurable silicon array of spiking neuron with conductance-based synapses. *IEEE Trans. Neural Netw.* 18, 253–265.

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: 14 September 2008; paper pending published: 23 December 2008; accepted: 09 May 2009; published online: 05 June 2009.

Citation: Brüderle D, Müller E, Davison A, Muller E, Schemmel J and Meier K (2009) Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. *Front. Neuroinform.* (2009) 3:17. doi:10.3389/neuro.11.017.2009
Copyright © 2009 Brüderle, Müller, Davison, Muller, Schemmel and Meier. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Near-infrared neuroimaging with NinPy

Gary E. Strangman^{1,2*}, Quan Zhang^{1,2} and Thomas Zeffiro²

¹ Department of Psychiatry, Harvard Medical School, Charlestown, MA, USA

² Neural Systems Group, Massachusetts General Hospital, Charlestown, MA, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Andrew D. Straw, California Institute of
Technology, USA

Matthew Brett, University of
Cambridge, UK

*Correspondence:

Gary E. Strangman, Neural Systems
Group, Massachusetts General
Hospital, 149 13th St – Psychiatry – Ste
2651, Charlestown, MA 02129, USA.
e-mail: strang@nmr.mgh.harvard.edu

There has been substantial recent growth in the use of non-invasive optical brain imaging in studies of human brain function in health and disease. Near-infrared neuroimaging (NIN) is one of the most promising of these techniques and, although NIN hardware continues to evolve at a rapid pace, software tools supporting optical data acquisition, image processing, statistical modeling, and visualization remain less refined. Python, a modular and computationally efficient development language, can support functional neuroimaging studies of diverse design and implementation. In particular, Python's easily readable syntax and modular architecture allow swift prototyping followed by efficient transition to stable production systems. As an introduction to our ongoing efforts to develop Python software tools for structural and functional neuroimaging, we discuss: (i) the role of non-invasive diffuse optical imaging in measuring brain function, (ii) the key computational requirements to support NIN experiments, (iii) our collection of software tools to support NIN, called NinPy, and (iv) future extensions of these tools that will allow integration of optical with other structural and functional neuroimaging data sources. Source code for the software discussed here will be made available at www.nmr.mgh.harvard.edu/Neural_SystemsGroup/software.html.

Keywords: near-infrared spectroscopy, python, NIRS, diffuse optical tomography, brain imaging

INTRODUCTION

The efficient conduct of neuroimaging experiments requires a diverse and complex assortment of computational resources. It follows naturally that constructing complete systems for data acquisition, analysis and display would be facilitated by the use of highly versatile, modular development environments. Functional neuroimaging data collection requires accurate timing of both stimulus displays and user responses, with near real-time graphics and device polling capabilities. The structural and functional neuroimaging datasets acquired over the course of a typical 1- to 2-h experimental session can exceed 10 gigabytes in size. These high data collection rates, along with the need to monitor the data flow for quality assurance purposes, require excellent system throughput and real-time data display capabilities to support experimental monitoring. Once acquired, neuroimaging datasets must undergo substantial preprocessing, data reduction and statistical processing to accurately model the many, often hierarchical, sources of variance in the raw data. These sources can include instrument noise, temporal autocorrelation, head motion, cardiovascular physiological effects, within-subject task effects, within-group effects, and between-group treatment effects. Finally, the statistical results must be displayed in an intuitive and easily comprehensible form using publication quality graphics.

While the construction of tools for each of these steps poses a substantial challenge, many current Python modules provide an excellent foundation on which to build data acquisition and processing pipelines. These advantages are already evident in magnetic resonance imaging (MRI) and electroencephalography (EEG) data processing applications, as demonstrated by other papers this issue. However, near-infrared neuroimaging (NIN) is

one domain for which no Python tools exist, and for which only two non-commercial software solutions are available (Huppert, 2006; Ye et al., 2009). We have therefore been developing a suite of Python modules to support the computational aspects of NIN data acquisition, analysis, and display. While our particular collection of tools is specialized for handling NIN data, the general design principles have broader application in experimental and theoretical neuroscience. We plan to release sub-modules under a BSD license, posting them at www.nmr.mgh.harvard.edu/Neural_SystemsGroup/software.html as they reach beta level stability.

We begin with an explanation of the physical and biological basis for NIN, followed by a brief comparative review of its chief uses. To provide context for our software development efforts, “Computational Requirements and Software” begins by describing the logistical and computational requirements associated with NIN experiments. The remainder of that section then describes the individual acquisition, analysis and visualization modules comprising the NinPy package, followed by a discussion of future software development directions in “Future Extensions”.

PRINCIPLES OF NEAR-INFRARED NEUROIMAGING

The physical principles underlying NIN are relatively simple, and similar to those encountered in pulse oximetry. The human scalp and skull are sufficiently transparent to the near-infrared (NIR) light wavelengths between 650 and 950 nm to enable non-invasive optical monitoring of physiological modulations associated with brain function (Jobsis, 1977). The NIR wavelengths are non-ionizing and therefore do not harm biological tissue at the low average power densities of 1–4 mW/cm² customarily utilized in brain imaging. For comparison, the ambient NIR light level on a

sunny summer day in mid-latitudes is approximately 20 mW/cm². By shining small spots of NIR light on the scalp and placing a detector a few centimeters away, the light intensity recorded by the detectors is modulated by the concentrations of all the absorbing chromophore molecules in the underlying tissues between the source and the detector. While sensitive to a range of chromophores and physiological phenomena (Villringer and Chance, 1997), NIN is particularly sensitive to the tissue oxygenation changes observed during changes in local neuronal activity (Huppert et al., 2006; Strangman et al., 2002b). A single source and detector pair can provide information about local changes in tissue optical properties. Spatiotemporal images of these physiological variables are generated by collecting multiple overlapping optical measurements and then applying tomographic image reconstruction techniques (Arridge, 1999; Franceschini et al., 2006; Pogue et al., 1999a). In addition to these spatial sampling capabilities, NIN is capable of temporal sampling in excess of 500 samples/s, a rate that compares quite favorably even with the most recent, ultra-fast MRI functional imaging methods (Lin et al., 2008a,b).

ADVANTAGES AND LIMITATIONS OF NEAR-INFRARED NEUROIMAGING

Near-infrared neuroimaging has several advantages when compared with other functional neuroimaging techniques, including: (i) comparatively low cost, (ii) sensitivity to multiple aspects of brain physiology, (iii) high temporal resolution, and (iv) suitability for portable or mobile applications. Together, these characteristics enable the use of non-invasive optical measurements in settings not normally compatible with brain imaging, including functional brain imaging in freely moving subjects. As with any technique, NIN also has limitations. Chief among these are a limited penetration depth of approximately 3–4 cm from the scalp surface, when using reflection geometry (Strangman et al., 2002a, 2003). In addition, non-invasive NIN allows only modest spatial resolution, estimated to be on the order of 0.5–1 cm in an adult human. Within these limits, however, NIN provides sensitive and reliable estimates of task-related neural activity originating in cortical structures comparable to results obtained using functional MRI (Huppert et al., 2006; Jasdzewski et al., 2003; Strangman et al., 2002b, 2006).

WHAT ASPECTS OF BRAIN FUNCTION CAN NEAR-INFRARED NEUROIMAGING MEASURE?

Although the basic NIN measurement involves recording the attenuation of light from a particular source as seen from the viewpoint of a particular detector, one can use raw light attenuation measurements at different wavelengths in the NIR range to obtain localized spectroscopic estimates of a wide range of physiological variables (Table 1). Some of these variables, like oxy- or deoxy-hemoglobin (O₂Hb and HHb) concentrations, are relatively straightforward conversions from measured attenuation values (see Section “Spectroscopic Conversion”). Others involve estimation of the physiological variables of interest from combinations of estimated chemical concentrations, as in the case of oxygen saturation or the cerebral rate of oxygen metabolism (CMRO₂). Finally, the temporal modulations of these variables can be used to compute indirect estimates of physiological phenomena like heart rate, respiration rate or modulation in baroreceptor activity (Mayer waves).

Table 1 | Physiological variables that can be estimated using NIN.

Chemical measurements	Physiological variables	Temporal variables
Oxy-hemoglobin concentration	Blood volume	Heart rate
Deoxy-hemoglobin concentration	Blood flow	Respiration rate
Total hemoglobin concentration	Oxygen saturation	Mayer waves
Water concentration	CMRO ₂	Low-frequency oscillations
Cytochrome oxidase concentration	Neural activity	
pH		

Near-infrared neuroimaging measurements of hemodynamic variables can be used to derive estimates of regional brain activity. This relationship between neural and hemodynamic activity is based on combined electrophysiological and fMRI results demonstrating that local changes in neural activity, reflecting both dendritic and axonal activity, are associated with focal variations in blood flow and volume (Logothetis, 2008). Because hemodynamic and neural activity changes often covary linearly, it is possible to use localized spatiotemporal recording of brain hemodynamics to make inferences about antecedent, and presumably causally related, neural activity patterns. For studying brain mechanisms underlying complex behavior, NIN hemodynamic imaging has particular advantages over other imaging modalities in the non-invasive detection of neural activity modulations. For example, as compared to EEG, NIN signals are more spatially localized (Strangman et al., 2003) and much less susceptible to the type of bioelectric interference generated by task-related scalp and face muscle activity. NIN signals also do not require tasks that produce the sorts of synchronous neural discharges that are needed to generate detectable event-related electrical potentials. In addition, when directly compared to invasive electrical measurements, hemodynamic responses are just as strongly related to induced patterns of neural activity as are the synchronous field potentials from which evoked potentials arise (Logothetis et al., 2001; Logothetis and Wandell, 2004).

In summary, the non-invasive character, and high sensitivity of NIN to a broad range of physiological phenomena reflecting many different aspects of brain function, makes it a promising method for use in a large number of clinical and experimental neuroscience contexts.

COMPUTATIONAL REQUIREMENTS AND SOFTWARE

Of its many potential applications, we have been particularly interested in using NIN to study the neural mechanisms underlying complex behavior. In particular, to facilitate the use of NIN in studies of the neural mechanisms of action and perception, we have developed a suite of programs, collectively called NinPy, that provide a wide range of integrated computational tools for use in optical functional neuroimaging experiments. A summary of the principal capabilities and components in NinPy appears in Table 2, along with the main Python modules and packages upon which each component is based. Each of these will be elaborated in the sections that follow.

There currently are two main software packages for handling NIN data: HomER (Huppert, 2006) and NIRS-SPM

(Ye et al., 2009). Both of these packages provide excellent data processing capabilities for many of the analysis and display aspects of NIN data processing. HomER provides a wealth of temporal processing capabilities and image reconstruction techniques, whereas NIRS-SPM provides broad statistical modeling and display capabilities by integrating with, and building upon, a well-established neuroimaging software package, SPM. However, neither package includes capabilities for acquisition, including experiment design, stimulus display, and data collection. NinPy seeks to provide an integrated platform combining all of these features, with a focus on features that complement those available in HomER and NIRS-SPM.

Table 2 | NinPy components and their core supporting Python modules.

Capability	NinPy component	Primary Python modules
ACQUISITION		
Stimulus display	NinSTIM	PsychoPy, Pyglet
User input	NinSTIM	PsychoPy, cgit, Pyglet
Synchronization	NinSTIM	pyparallel/pyserial
NIRS data collection	NinDAQ	Chaco, Traits
ANALYSIS		
Quality assurance	NinPROC	NumPy
Filtering	NinPROC	NumPy, SciPy
Image reconstruction	NinPROC	NumPy, SciPy
Parameter estimation	NinSTATS	SciPy, RPy
Statistical modeling	NinSTATS	RPy
DISPLAY		
Visualization	NinDISP	Matplotlib

CONDUCTING NEAR-INFRARED NEUROIMAGING EXPERIMENTS

Conducting a typical NIN experiment requires two distinct software tools: one for experimental control and the other for data acquisition. Although these tools operate independently, their efficient use together requires a high degree of functional integration at the design level. As described next, NinSTIM is a stimulus generation and display system for experimental control, and NinDAQ is a data acquisition and monitoring system for device control.

Stimulus generation and user input (NinSTIM)

Accurate and reliable control of stimulus presentation is a critical aspect of any functional neuroimaging experiment. NinSTIM is a high-level stimulus and experimental design toolkit, designed for non-programmers, that generates stimulus sequences for display by the Pyglet interface¹ to the PsychoPy package² (Peirce, 2008). NinSTIM directs PsychoPy to sequentially present an ordered collection of “trials”, where a trial is a very general entity consisting of one or more temporal phases, each composed of one or more visual or auditory stimuli. For example, a trial could be: (i) a simple instruction screen presented while the program waits indefinitely for a key press, (ii) a visual fixation of predetermined duration, (iii) a stimulus followed by a mask, or (iv) any other ordered series of stimuli. An example complex trial with five separate phases might be: (i) a side-by-side pair of photos, followed by (ii) a brief whole-screen mask image, followed by (iii) a variable duration blank screen delay period, followed by (iv) a go cue, and finally (v) an inter-trial rest period. Each unique trial type is defined in a ASCII trial definition (.DEF) file, with required Python-style indentation, for editing and interactive debugging (Figure 1, left).

¹www.pyglet.org

²www.psychopy.org

```
# trial definition .DEF file

backgroundColor (-1,-1,-1)
Ready
    -1 keyboard
        allowableKeys space
        Ready ...
            pos (0,0.2)
            height 0.15
Instructions_Left_3
    3 cumulative
        Instr_left_3.jpg
Fixation
    15 cumulative
        cross.jpg
Left.04
    1.5 exact
        L4.jpg
[etc.]
```

```
# trial order .ORD file

Ready
Instructions_Left_3
Fixation
Left0.04
Left0.03
Left0.05
Left0.01
Left0.02
Instructions_Right_1
Right1.04
Right1.01
Right 1.03
Right 1.02
Right 1.02
Fixation
Thanks
```

FIGURE 1 | Abridged examples of the trial definition (.DEF) file format and the trial order (.ORD) file format. Each trial named in the .ORD file must be defined in the .DEF file. For the first trial (“Ready”), “timing = -1 keyboard” means wait indefinitely for a keypress (the spacebar is the only allowable key) while displaying the text “Ready ...” at position (0,0.2) and

height 0.15. The “Fixation” trial involves displaying the image file cross.jpg in the center of the screen for 15 s, with extra frames inserted or removed there if cumulative timing errors have accumulated. The “Left.04” stimulus displays the image file L4.jpg in the center of the screen for exactly 1.5 s.

The breadth of experimental designs commonly employed in functional neuroimaging experiments requires sophisticated and flexible procedures for trial scheduling. Possibilities for the temporal ordering of trials include: (i) block designs, in which groups of evenly spaced trials alternate with periods of fixation, (ii) stochastic, or “event-related”, designs, in which the individual trial times are varied to allow efficient estimation of hemodynamic responses using deconvolution procedures (Dale, 1999), and (iii) mixed designs, combining aspects of both block and stochastic designs to achieve separation of state and task-related experimental effects. In the case of stochastic and mixed designs, the trial durations and orders that lead to maximum efficiency in the detection of task-related brain activity can be computed using programs such as *optseq*³, and then entered in a trial order (.ORD) file. As with the trial definition file, the trial order input file is a simple, ASCII file (**Figure 1**, right). From these two input files (.DEF and .ORD), NinSTIM builds and then runs a PsychoPy-compatible program.

PsychoPy and Pyglet, the engines driving stimulus presentation, also provide facilities for logging stimulus, keyboard and mouse events. Through the Pyglet event loop, one can continuously monitor these events and respond appropriately. For example, one can display different stimuli depending on user input, or compensate for certain timing vagaries inherent in soft real-time operating systems. In soft real-time operating systems like Microsoft Windows, interrupts and system processes can sometimes seriously disrupt the accuracy and precision of stimulus timing. This is a widely recognized problem that is addressed using differing mechanisms in the stimulus presentation packages most commonly used in experimental neuroimaging, including EPrime⁴, Presentation⁵, Psychtoolbox⁶, and Cogent⁷. To optimize timing in NinSTIM we: (i) increase the stimulus display process priority to “High” via Python’s *win32process.SetPriorityClass()*, (ii) disable Python garbage collection, (iii) enable drawing synchronized to the vsync pulse from the monitor, and (iv) pre-draw stimuli whenever possible to maximally engage the blocking mode of calls to OpenGL flip (Straw, 2008). Stimulus onset timestamps are collected using Python’s *time.clock()* call which is executed the line after the call to flip the OpenGL graphics buffer. The timing requested by the user in the trial definition and order files – which we call the nominal timing – is also simultaneously monitored. Using the “cumulative” timing type, users can identify the less critical stimulus or delay times, for which NinSTIM can add or subtract one or two frames, to preserve the experiment’s cumulative nominal timing. In a 12-h test using this approach, involving 15,600 trials and 31,000 stimuli, our *time.clock()* timestamps occurred a maximum of 26 ms early to 88 ms late compared to nominal, with a mean and SD timing error of 1.6 ± 6 ms. Individual stimulus durations ranged between ± 8 ms off nominal – or half a screen refresh on our 60-Hz monitor. Note that these latencies do not represent the total system delay, defined as the interval between the time a user event is captured and a new image is displayed. Moreover, these latencies

were measured by the internal computer clock, rather than an external source. Hence, the above numbers may underestimate the exact latency to stimulus presentation (Straw, 2008). However, the maintenance of nominal timing within a few tens of milliseconds over several hours is more than adequate for functional neuroimaging experiments based on hemodynamic responses, which includes the vast majority of NIN experiments.

Using the standard Python threading and ctypes modules it is also possible to collect continuous data streams from other user input devices during stimulus display. Access to almost any device driver is possible through ctypes. By setting up a separate timer thread, densely sampled data streams from auxiliary input devices can include time stamps from the same master clock that marks all stimulus, keyboard and mouse events. This arrangement dramatically reduces the timing uncertainty between stimulus presentation and recording devices and can provide a record of any mismatch between intended and actual experimental event times. This sort of continuous, simultaneous recording of auxiliary devices can be difficult or impossible to implement using many of the popular experimental control programs. In addition, the *pyserial* and *pyparallel* Python modules (Liechti, 2008) provide a separate means for acquiring event signals from, or exporting trigger signals to, the computer’s serial or parallel ports for synchronization with our NIN acquisition devices.

Because NinSTIM is based on Python, chaining multiple experiments is easily achieved with successive Python calls, or a separate Python script that runs each experiment in succession.

Data acquisition and real-time data display system (NinDAQ)

Optical imaging devices are constructed from multiple hardware subsystems that require dedicated device control software. Using Enthought’s Chaco/Traits modules (Enthought, 2007, 2008), along with NumPy (Oliphant, 2006) and SciPy (Jones et al., 2001) we have also developed NinDAQ, a device control program customized for two of our NIN instruments (**Figure 2**). This program provides complete, real-time control over the NIN device state variables, including laser state, amplifier gain, analog acquisition subsystem voltage range, and sampling rate. NinDAQ also controls the data acquisition process including start signals, stop signals, and data display modes. Important additional features include: real-time temporal display of relatively large amounts of data, pushbutton toggling to “zoom in and out” on the data stream as it is being collected, and automatic scaling of the signal range to the minimum and maximum values of each data line. Real-time control of the acquisition process is provided, including provisions for user-generated interrupts of data collection, variable temporal windows for strip-chart data views, and interactive laser control. The Chaco plotting package provides real-time plotting capabilities, while Enthought’s Traits supports rapid GUI development cycles. The standard Python ctypes module enables seamless access from Python to the commercial drivers for our analog-to-digital data acquisition boards.

SIGNAL PROCESSING (NINPROC)

Once complete, most neuroimaging experiments produce two file types: text files that log the stimulus and response events, and custom binary data files containing the neuroimaging data. Depending on the type of experiment and the specific neuroimaging device,

³<http://surfer.nmr.mgh.harvard.edu/optseq/>

⁴www.pstnet.com/products/e-prime

⁵www.neurobs.com

⁶<http://psychtoolbox.org/PTB-2/>

⁷www.vislab.ucl.ac.uk/cogent.php

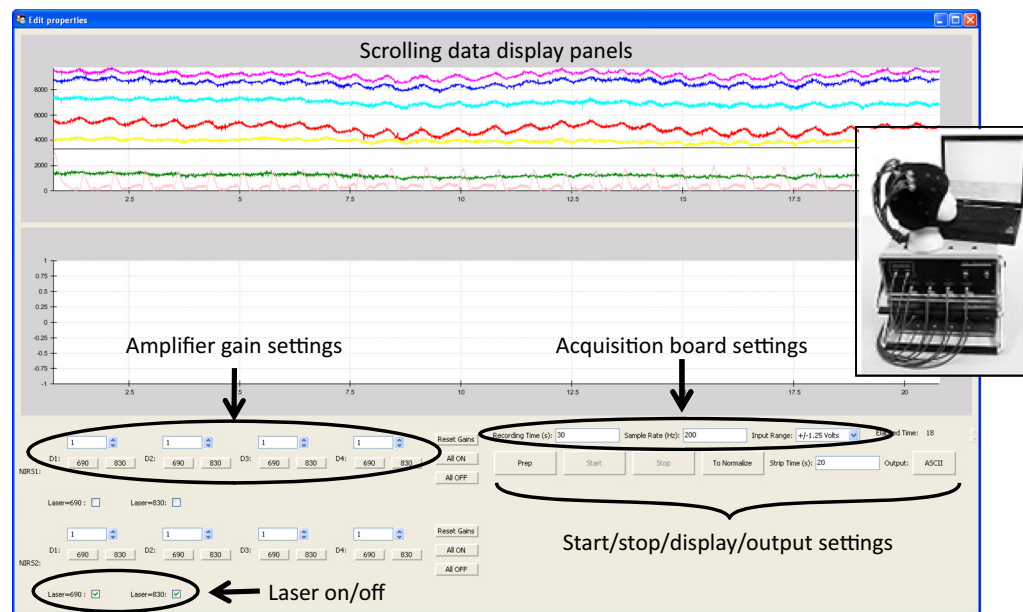


FIGURE 2 | Screenshot from the NinDAQ device control and data acquisition program. Inset: The NIN recording devices and head probe being controlled by this software.

raw data from a single participant in single experimental session can be many gigabytes in size. In experiments incorporating cardiac, respiratory, kinematic or other physiological data monitoring, a third file type containing records of such continuous data streams may also be produced. Each such data file has unique processing requirements that can be handled via Python, or using the NumPy and SciPy libraries.

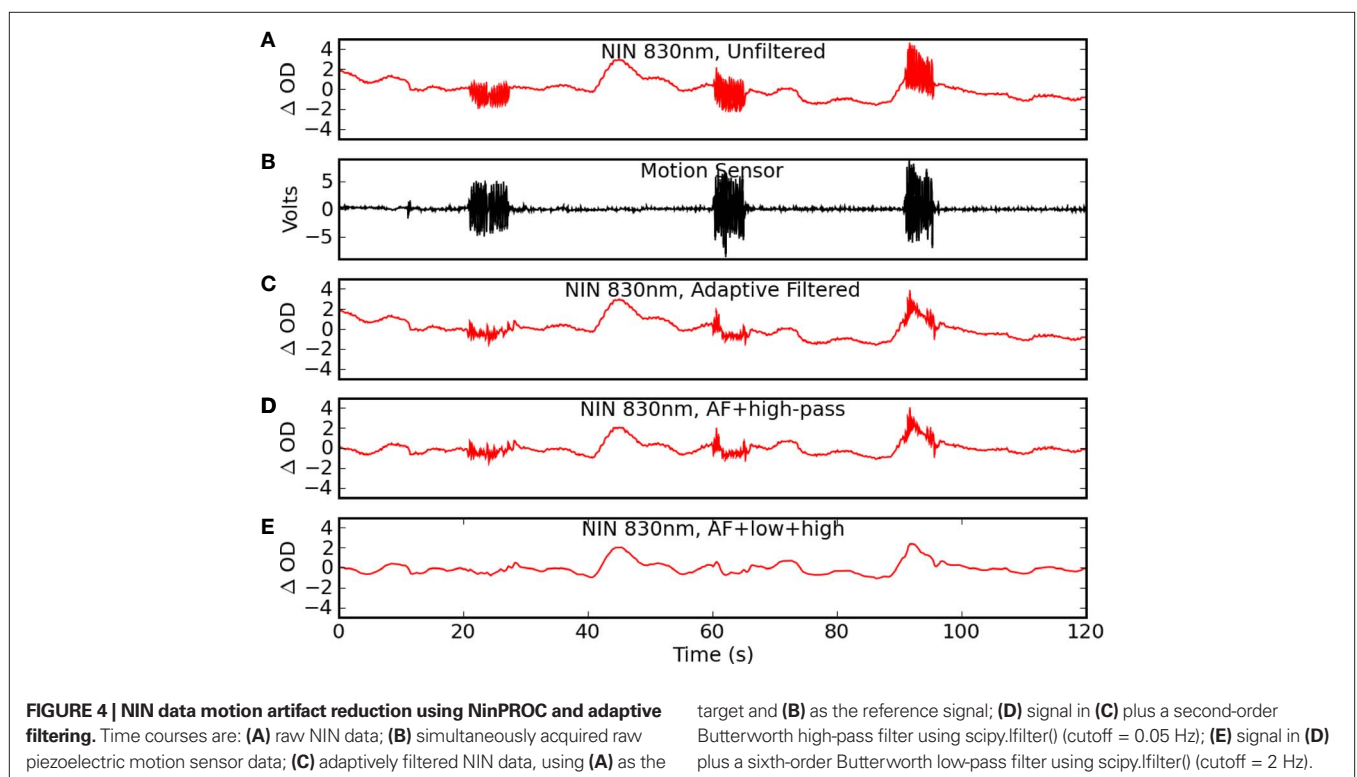
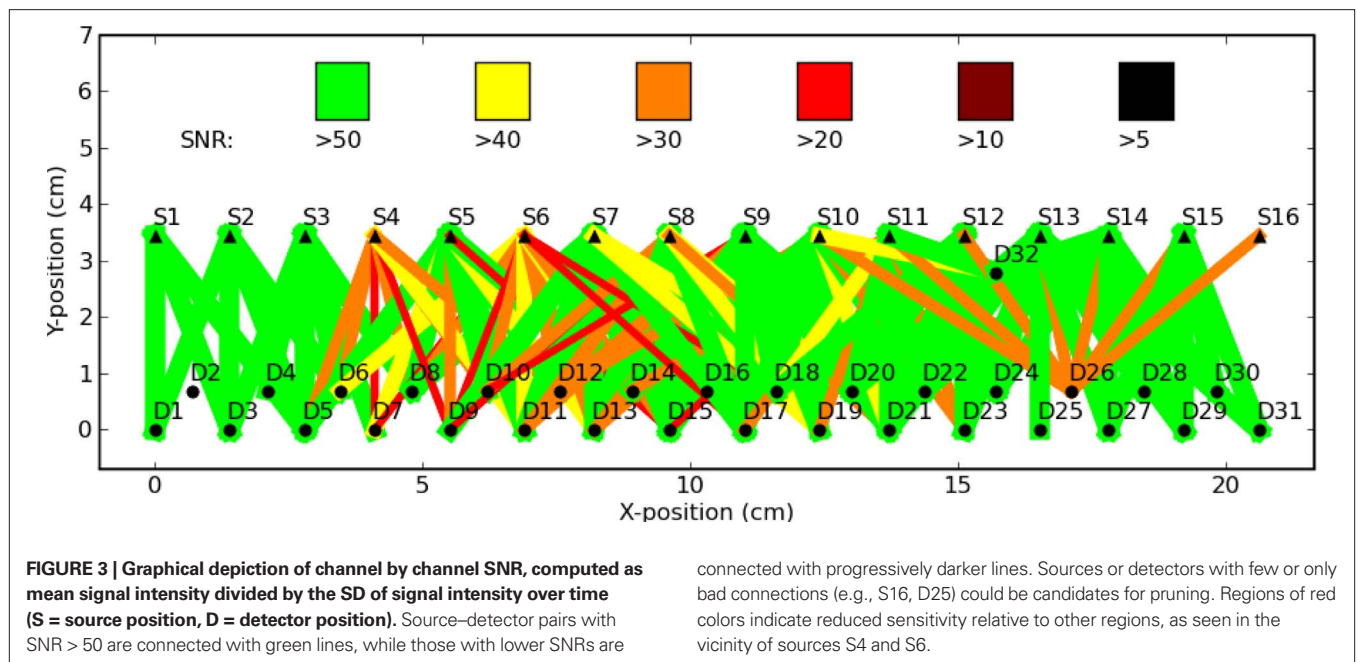
Quality assurance and filtering

Quality assurance procedures for stimulus and event log files involve validating event timing by examining deviations from nominal event times and durations, detection of skipped stimuli or skipped frames, detection of device failures, and identification of other experimental anomalies, including task performance deviations. Data quality checks can be easily implemented in Python by opening the log files generated by NinSTIM and NinDAQ, reading in each line with the recorded actual and nominal times, and computing various time differentials. NinPROC uses simple descriptive statistics to identify deviations from the expected experimental event timing, with relevant functions contained in NumPy (amin, amax, mean, std, or median) or scipy.stats (skew, kurtosis, or histogram). There is also an option to graphically display histograms to visually identify anomalous timing patterns during particular runs, using matplotlib.hist() and plot() functions. For physiological or NIN data time series, numpy.loadtxt() or numpy.fromfile() can be used to efficiently read in the data, which can be similarly scanned for timing irregularities, intermittent signal dropout or other deviations from the experimental protocol. In addition, multiple time series can be quickly and automatically plotted with nindisp.plot() for visual inspection.

To identify and remove the sorts of signal artifacts specific to NIN data, we have included algorithms in NinPROC for semi-automated

signal pruning. For a variety of reasons, not all source–detector pairs will provide useful information in all experiments. Data from some source–detector pairs not of primary interest may have been recorded during the experiment, some source–detector pairs may have been too far apart to provide reliable signals, or a detector may have lost contact with the head, thereby generating large signal artifacts. Within the preprocessing component NinPROC, the ninproc.prune() function is available to remove particular sources, detectors, or channels based on the known source–detector separations. In addition, low overall signal intensity can result in unreliable information, and high overall signal intensity can indicate light leakage from source to detector. Hence, facilities for displaying and pruning based on absolute signal intensity and signal-to-noise ratio (SNR) are also provided as options (Figure 3). In addition, the ninproc.lowpass(), ninproc.highpass(), and ninproc.notch() functions provide simple, zero-phase filtering to reduce 1/f physiological, instrument, or electrical interference noise components.

As with all neuroimaging data, NIN time series can contain physiological motion artifacts. When head motion occurs, the resulting signal modulations can be substantial and therefore must be identified and either excluded or otherwise mitigated. Exclusion of a motion contaminated time series segment is a less than ideal solution, so effective mitigation is an important tool. One approach, which is particularly well-suited to real-time applications, is adaptive filtering. In previous work, we have demonstrated the efficacy of adaptive filtering to identify and reduce global physiological interference in NIN signals, including signal modulations resulting from cardiac or respiratory oscillations (Zhang et al., 2007a,b). We have recently added a least mean squares-based adaptive filter for motion artifact reduction to NinPy called ninproc.lms() (Figure 4). Adaptive filtering has shown considerable promise in real-time reduction of



physiological motion artifacts without the bandwidth loss associated with using a low-pass filter with a low cutoff frequency. Other published approaches to dealing with NIN motion artifacts include the use of principle component analysis or independent component analysis to identify and separate signal from motion waveforms (Morren et al., 2004; Zhang et al., 2005), solutions that could be incorporated using the Python-based Modular toolkit

for Data Processing (Berkes et al., 2008) via `mdp.pca()` or `mdp.fastica()`.

Spectroscopic conversion

Table 1 lists multiple types of optical contrast detectable with NIN (Villringer and Chance, 1997). Many of these contrasts are computed via spectroscopic conversion using the modified Beer-Lambert law

(Delpy et al., 1988). These conversions are linear algebra transformations performed on each time point of raw attenuation data and the resulting time series reflect time-varying changes in chromophore concentrations. To compute chromophore concentrations, raw measurements recorded from two or more NIR wavelengths are first log transformed to changes in optical density, and then to changes in O₂Hb, HHb, and total hemoglobin (O₂Hb + HHb) concentrations:

$$\begin{aligned}\Delta OD(\lambda) &= -\log_{10}(I/I_o) = \Delta\mu_a(\lambda)L \cdot DPF(\lambda) \\ &= [\epsilon_{O_2Hb}(\lambda)\Delta[O_2Hb] + \epsilon_{HHb}(\lambda)\Delta[HHb]]L \cdot DPF(\lambda)\end{aligned}$$

where I is the raw measured intensity at a single point in time, I_o is the measured light intensity at a reference time point, ΔOD represents the change in optical density between I and I_o , the $\epsilon(\lambda)$ s are extinction coefficients for O₂Hb and HHb at a given wavelength (λ), L is the source–detector separation, and $DPF(\lambda)$ is the wavelength-dependent differential pathlength factor that converts L to the true (scattered) optical pathlength. Recording data from two wavelengths (λ_1 and λ_2) provides two such equations with two unknowns: the change in O₂Hb and HHb concentrations. The `ninproc.extinction_coef()` function uses interpolated lookup tables to obtain extinction coefficients of the various optical chromophores. With these coefficients, conversion to concentrations over

all time points can generally be accomplished compactly in Python using NumPy arrays, broadcasting, and its linear algebra capabilities, as shown in **Code Fragment 1**, where L is a 1D array of source–detector separations for each channel, `rawdata` is a 2D array of raw (or pruned and filtered) NIN data, `rawref` is a 1D array of raw NIN data from a reference period – e.g., `N.mean(rawdata[:100],0)`, A is a linear transform between optical density and concentration represented as a 2D matrix of extinction coefficients, `hhb` and `o2hb` are 1D arrays of HHb and O₂Hb concentrations (in units of moles/mm) over time. While A is normally invertible, sometimes it is not. For such cases, one can use `numpy.linalg.pinv()` in place of `numpy.linalg.inv()`. The results of these steps are shown in **Figure 5**.

IMAGING (NINDISP)

Near-infrared measurements of brain function can be made with a single source–detector pair, providing information localized to approximately 0.5–1 cm² of brain tissue (Strangman et al., 2003). A spatially-distributed collection of such measurements can be combined into an image for each relevant optical contrast. In functional neuroimaging, task-related images of O₂Hb, HHb and O₂Hb + HHb changes are of primary interest, as these parameters have been shown to reflect underlying changes in neural activity (Jasdzewski et al., 2003; Strangman et al., 2002b). Imaging procedures can consist of topology preserving sensor space representations, back

```
ODdata = -numpy.log10(rawdata/rawref) # compute optical density
A = ninproc.extinction_coef(wavelengths, 'Hemoglobin') # table lookup
hhb, o2hb = numpy.dot(numpy.linalg.inv(A), ODdata) / (L*DPF) # compute concentrations
```

CODE FRAGMENT 1 | Three lines to convert raw NIN data to oxygenation concentrations.

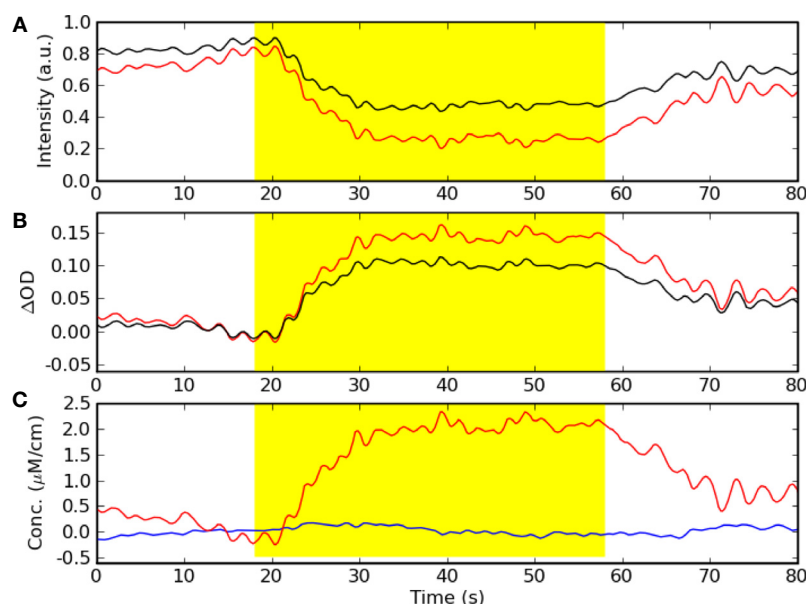


FIGURE 5 | Spectroscopic conversion steps of NIN data time series from one source–detector pair. **(A)** Raw recorded NIN light intensity data from two wavelengths, in arbitrary units. **(B)** Data from **(A)** after log transformation to optical density units. **(C)** Data from **(B)** after conversion to hemoglobin concentration units (red = oxy-Hb, blue = deoxy-Hb, yellow = period of task activity).

propagation – also called topographic imaging – or tomographic reconstructions (Arridge, 1999), as discussed below.

Sensor space representations

Perhaps the simplest approach to imaging, commonly utilized in EEG and MEG data displays, involves plotting multiple sensor time series or time averages, with each sensor positioned in the display according to the scalp location of the measurement. An example of this approach from NinDISP, using the powerful matplotlib plotting package (Hunter, 2007), appears in **Figure 7B**. The surface array visualization technique preserves the temporal information at each sampling point, and is particularly effective if the sensors are widely separated.

Topographic imaging

In topographic imaging, measurements obtained from different locations in space are linearly interpolated to a regular grid to generate 2D images of either the underlying optical signal changes or derived parameters. The matplotlib.mlab.griddata() function can be used to compute such tomographic images. For example, if data is an $N \times 3$ array of $[x, y, \text{val}]$ triples irregularly spaced over a 10 cm by 6 cm region, a 2D topographic projection of the val parameter with 1 mm pixels could be computed as follows (see **Figure 7C**):

```
# xi is the interpolated, regular grid x-dim
xi = numpy.linspace(0., 10., 100)
# yi is the interpolated regular grid y-dim
yi = numpy.linspace(0., 6., 60)
zi = matplotlib.mlab.griddata(data[:,0],
data[:,1], data[:,2], xi, yi)
```

This is a simple and compact data visualization technique, but it also embodies many important assumptions. In particular, interpolation assumes that the time varying optical properties of brain tissue between measurement locations can be accurately

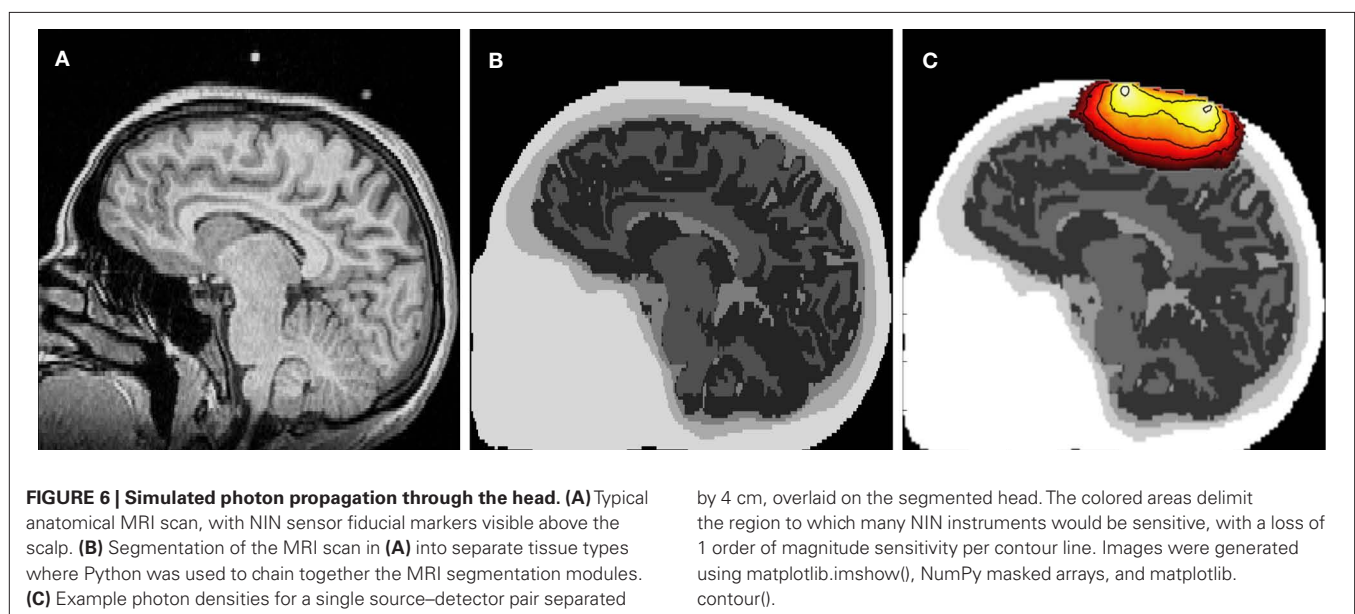
estimated by averaging the signals derived from the neighboring actual measurements. This may or not be true depending on the spatial scales of the signal and the source–detector geometry. In the above example, it also assumes accurate prior knowledge of the (x, y) coordinates of the val parameter, which may be difficult to obtain or estimate. For simple geometries, however, this computationally efficient method is suitable for real-time display and can be quite useful for visualizing the spatiotemporal structure of signal modulations.

Tomographic imaging

Tomographic imaging, in contrast to topographic imaging, is more appropriate when multiple, spatially overlapping NIN measurements are collected. In this case, tomographic image reconstruction generates a solution that best satisfies all measurements simultaneously. The reconstruction is computed in two stages. First, one must estimate the diffusion paths of photons and calculate the sensitivity profile throughout the brain. In image reconstruction, this step is termed the “forward problem.” For simple, semi-infinite, homogeneous media, the distribution of photons injected into tissue can be approximated by the diffusion equation (Farrell et al., 1992), and solved analytically. However, for more complicated geometries, analytical solutions are not possible and hence numerical solutions are often employed, including finite difference, finite element and Monte Carlo approaches (Jacques and Wang, 1995). Here we discuss the Monte Carlo approach. For the particularly complex tissue geometry of the head, one can start with a standard high resolution, T1-weighted MRI scan (**Figure 6A**). This structural scan can then be segmented into gray matter, white matter, cerebrospinal fluid, skull, and scalp tissue types using Python to call any of the MRI tissue segmentation tools contained in analysis packages such as SPM8⁸ or FSL⁹. Next,

⁸www.fil.ion.ucl.ac.uk/spm

⁹www.fmrib.ox.ac.uk/fsl



each tissue type in the segmented volume is assumed to be homogeneous and assigned optical properties based on literature values (Choi et al., 2004; Kohri et al., 2002; Leung et al., 2005; Okada and Delpy, 2003; Strangman et al., 2003).

To perform the Monte Carlo simulation process, approximately 100 million photons are injected, one at a time, into the segmented model (**Figure 6B**) at the location of a source or detector. The propagation of each photon through the tissue is determined probabilistically given the physics of light and the optical properties assigned to each tissue type. This process is repeated for each source and detector location and the result is a participant-specific solution to the forward problem. Multiplying together the photon densities for a given source–detector pair, point by point throughout the brain volume, provides an estimated sensitivity profile for that source–detector measurement pair (**Figure 6C**). As with the MRI segmentation routines, Monte Carlo techniques can be implemented with Python calls to existing toolboxes in Matlab (Boas, 2004) via `mlabwrap` (Schmolck, 2007), or by calls to binaries such as `tMCimg` (Boas, 2008) using Python's `os.popen()` function. For NinPROC, and the steps in **Figure 6**, we utilized the lattermost approach, which allows us to gradually transition complex code bases to Python, as time and resources permit.

Given a stable solution to the forward problem (**Figure 6C**), the second imaging step is to generate an image of the optical contrast parameter. This step is called “inverse modeling,” and it can be accomplished using linear or non-linear methods. The linear approach is typically formulated as $y = Ax$, where y is a length- M vector containing the value of the parameter of interest for each NIN source–detector pair, x is a length- N vector of all voxels in the image reconstruction, and A is the sensitivity matrix (Jacobian), which is an $M \times N$ matrix based on the Monte Carlo simulation that maps the sensitivity of each point in x to each measurement in y (**Figure 6C**). To solve for x , the equation of interest becomes: $x = A^{-1}y$, where A^{-1} computed using `numpy.linalg.inv(A)` or, more often, the pseudoinverse of A via `numpy.linalg.pinv(A)`. Because this problem is usually ill-posed and underdetermined ($N \gg M$), regularization is typically applied, often via singular value tapering as is used in Tikhonov regularization (Pogue et al., 1999b). NIN image reconstruction then essentially reduces to two python function calls: matrix multiplication via `numpy.dot()` and regularization with `numpy.linalg.svd()`.

STATISTICAL MODELING AND VISUALIZATION

The final stage of an NIR functional imaging experiment, after completing the data collection and the signal and image processing steps, involves parameter estimation, statistical modeling, and visualization of the results.

Statistical modeling (NinSTAT)

Statistical modeling involves modeling experimental variance to derive parameter estimates pertaining to the experimental effects of interest. SciPy includes a number of basic statistical functions that are suitable for modeling experimental effects in individual subjects. However, data from many neuroimaging experiments, particularly those involving comparisons of different participant groups, have a complex and hierarchical variance structure that cannot be effectively modeled with SciPy routines. In particular, within-subject

designs, incorporating repeated measurements collected from each participant under a range of experimental conditions are quite common. These designs are popular because they have relatively high sensitivity, and they avoid the time and expense of recruiting and fully characterizing large groups of research participants. Within-subject variability in functional neuroimaging data, while substantial, tends to be smaller than between-subject variability. Prominent sources of between-subject variation include: (i) brain size and shape differences, (ii) neurovascular coupling differences, (iii) task performance differences in accuracy or response time, and (iv) variation in the specific strategy used to perform the task. To accurately model both within- and between-subject effects, therefore, requires mixed-effects modeling techniques (combining fixed and random effects), which are not available in HomER or SciPy. In addition, given the great diversity in experimental designs employed in functional neuroimaging experiments, specifically coding each statistical model in Python would be associated with substantial effort. These reasons motivate integration with an external statistics package.

R is a widely-used, open-source, statistics package that contains a very comprehensive and sophisticated collection of statistical analysis methods (R Development Core Team, 2005), including tools that are able to model NIN data with complex hierarchical structure. One common example is with mixed effects models that contain variables measured at different levels in a hierarchy, as in the case of summary statistic models in which separate regression analyses are computed for each participant, with the resulting first-level regression coefficients being treated as random variables at the second level (Pinheiro and Bates, 2000). Rather than rewriting the requisite statistical procedures in Python, the RPy module (Moriera and Warnes, 2004) provides a lightweight yet powerful interface between Python and R for statistical analysis, with results automatically returned to Python for storage, subsequent processing or display.

A particular advantage of using R is that an extremely broad range of models can be applied to the data, since all input variables are treated equally. In particular, the neuroimaging data can be used either as an outcome variable, a predictor, or a covariate. This assignment flexibility is in contrast to that found in the most commonly used neuroimaging software packages, including SPM, FSL, AFNI, FSLFast. These packages require the neuroimaging variable to be the outcome variable, which significantly restricts the types of scientific questions that can be addressed. For example, one question that is receiving growing interest concerns identification of brain regions that might provide predictive information about treatment response. This determination requires the neuroimaging data to act as a predictor and the therapeutic response measure to serve as a dependent or outcome variable. Implementing these models using existing neuroimaging packages requires extracting the data from each potential brain region of interest, exporting the data series, and then performing the statistical analysis using an external program (Strangman et al., 2008). By directly interfacing with R, one can fit predictive models as easily as those utilizing the image data as the dependent variable. **Code Fragment 2** provides an example of a NinSTATS implementation of predictive modeling. Importantly, R includes a large, and continually growing, collection of heavily tested and more sophisticated models, including robust

```

import rpy2.robjects as ro

nin = nifti.NiftiImage('allsubj_contrast1.nii') # parameter file with subject by X by Y by Z dimensions

tags = numpy.loadtxt('allsubj_tags.txt') # columns: subjnum, age, pretest score, outcome score
header = ['subj', 'age', 'pretest', 'outcome', 'nin']
shape2D = (nin.data.shape[0], numpy.multiply.reduce(nin.data.shape[1:]))

nindata = numpy.reshape(nin.data, shape2D) # flatten X, Y and Z dimensions (subj by voxel)

# Prepare to save 3 results (coef/sterr/T-score) for 4 terms (intercept, age, gender, nin) at each voxel
results = numpy.zeros((4,3,nindata.shape[1:]),numpy.Float)
for i in xrange(len(nindata.shape[1])): # loop over all voxels
    # COLLECT NIN DATA FOR THIS VOXEL AND CREATE AN R DATA FRAME
    thisdata = make_RVector_list(tags) # NINstats helper function for building data frames
    thisdata = thisdata + [ro.RVector(array.array('f',nindata[:,i]))]
    header = header + ['nin']
    # CREATE THE DATA FRAME, WITH NAMES (using a dict alone segfaults rpy1.0rc1)
    tl = rlc.TaggedList(thisdata,tuple(header))
    df = ro.RDataFrame(tl)

    # FIT A MULTIPLE LINEAR REGRESSOIN MODEL WITH NIN AS A PREDICTOR
    formula = ro.r.formula('outcome ~ age + pretest + nin') # use NIN data as predictor of outcome
    fittedmodel = ro.r.lm(formula,df) # fit a linear multiple regression model

    # EXTRACT AND STORE RESULTS FORM THE MODEL FIT FOR THIS VOXEL
    summ = ro.r.summary(fittedmodel)
    ttable = summ[3] # retrieve estimated coefficients and t-table results
    for j in range(len(ttable)):
        results[j,0,i] = ttable[j][0] # coefficient
        results[j,1,i] = ttable[j][1] # sterr
        results[j,2,i] = ttable[j][2] # T-value

```

CODE FRAGMENT 2 | NinSTATS code fragment to perform statistical analysis with functional NIN data as a predictor of outcome.

covariance and generalized linear models, as well as a wealth of post-hoc testing capabilities.

Visualization (NinDISP)

Once a neuroimaging statistical analysis is complete, visualization enhances both interpretation and communication of the results. Sensor space visualization, an approach discussed earlier, is shown in **Figures 7A,B**. However, it is common in neuroimaging experiments to have even larger collections of spatially coherent univariate statistical results. For example, the code in **Code Fragment 2** might produce 1,000 or more distinct model fits. In this case, sensor space visualization may be either impossible, because of too many measurements, or misleading, because overlapping measurements may be sensitive to different depths. Imaging provides certain advantages in these situations, as shown in the topographic image in **Figure 7C**, generated from task-related regression parameters from the O₂Hb traces in **Figure 7B**. Applying a statistical threshold to topographic images helps identify regions that are significantly modulated by the task, as shown in **Figure 7D**.

In addition to statistical parametric maps (**Figure 7D**), and time series plots (**Figure 7B**) it is often useful to generate and examine scatter or bar plots from regions-of-interest, or to produce summary plots of activity levels in various brain regions, including histograms and box plots. The matplotlib module provides all these options as well as many additional plot types. Critically, matplotlib includes complete customization capabilities for the creation of publication-quality figures (Hunter, 2007). Math or Greek symbols

can be easily added to the plot or axis labels, options that are particularly important for representing physical or derived units in NIN data (cf. **Figure 5**).

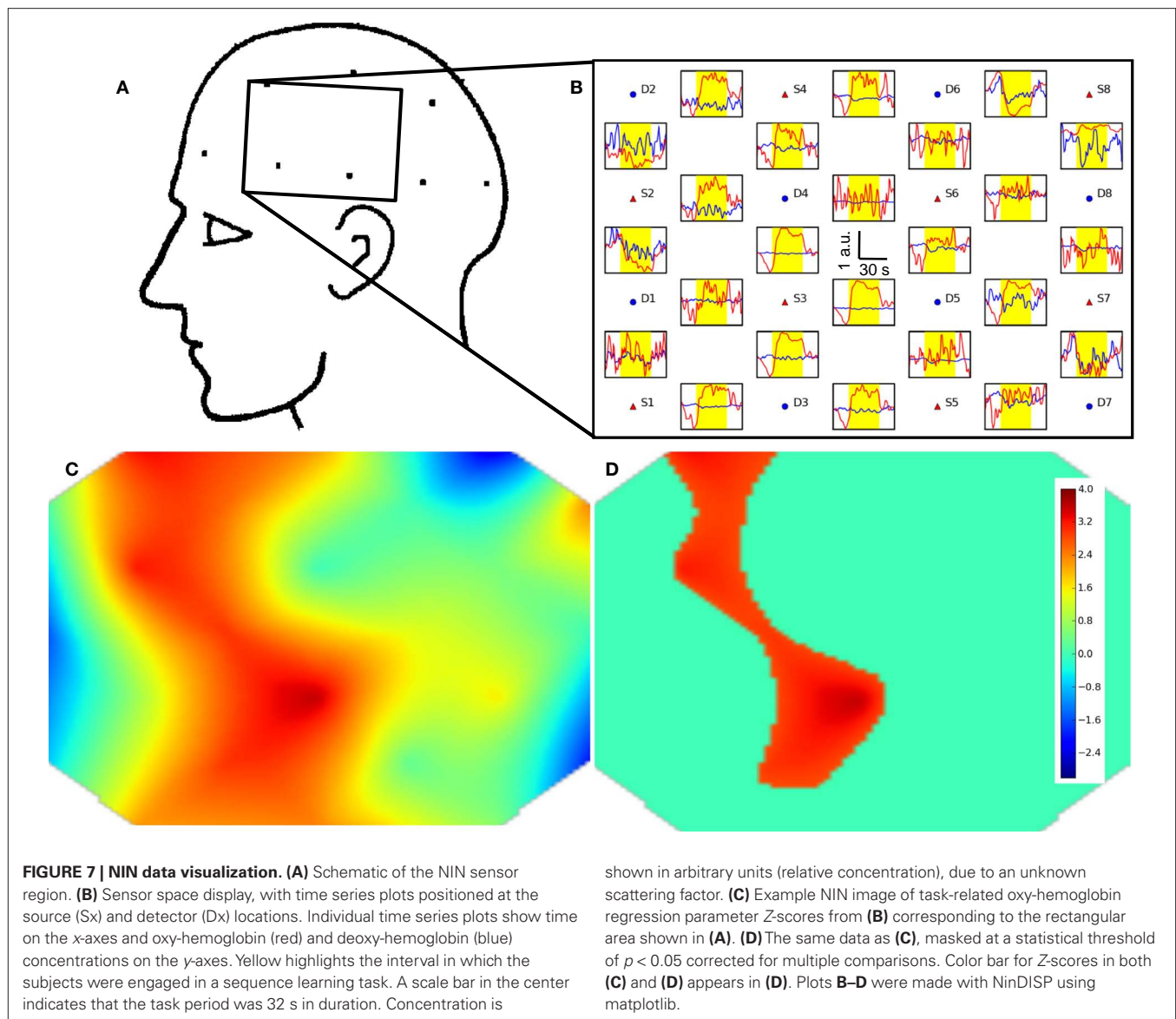
FILE FORMAT INTERFACES TO EXISTING OPTICAL IMAGING TOOLS

Due to the large volume of spatial and temporal data generated by neuroimaging experiments, neuroimaging data have always required custom file formats, and in the 1990s image file formats proliferated. Fortunately, the NIfTI standard (Cox et al., 2004) has made major inroads as a standard file format for MRI data. An example of its use in NinPy is seen in **Code Fragment 2**. Other formats still dominate in EEG, MEG, PET, as well as NIN, and a number of legacy formats still persist with some frequency in MRI applications. Our goal has been to integrate NinPy programs with three key data formats: NIfTI, the Matlab-based format used by HomER (Huppert, 2006), and the broad standard HDF5. These formats enable broad interoperability of the NinPy suite with existing tools for neuroimaging data analysis. NIfTI files are created, read and written through the use of the PyNifti package (Hanke, 2008), whereas the HomER file format can be read and written as a Matlab.mat file or HDF5 file (also readable by Matlab) containing multiple arrays with specific variable names. Reading and writing Matlab files is supported through scipy.io.loadmat() and scipy.io.savemat(), and thus HomER files can be saved from appropriate variables in Python as follows:

```

scipy.io.savemat('outname.mat',{ 'd':nindata, 't':
timebase, 'ml':meas_list, 'aux10':auxiliary}).

```



FUTURE EXTENSIONS

MULTIMODAL INTEGRATION

While we have only briefly discussed MRI integration with regard to Monte Carlo simulation, there are additional advantages associated with integrating NIN with MRI and other neuroimaging modalities. For example, the segmented MRI images (**Figure 6B**) could be used to constrain the NIN image reconstruction process by restricting reconstructed brain activity modulations to gray matter, thereby not allowing the estimated signal changes to occur in scalp, skull, cerebrospinal, or white matter tissue compartments. As another example, automatic identification of optical sources and detectors within the MRI space (the white fiducial markers above the head in **Figure 6A**) could be used as inputs to the Monte Carlo simulations or to provide more accurate co-registration of NIN statistical parametric maps with underlying brain anatomy.

While integration with EEG, MEG, and other neuroimaging technologies is occurring at the experimental level, integration at the data analysis and interpretation levels is a relatively underdeveloped area. One interesting possibility for integration involves the optical “fast signal”. NIN studies from several labs have shown changes in non-invasive optical signals on timescales much faster than typical hemodynamic changes, less than 100 ms as compared to 2–3 s or more (Franceschini and Boas, 2004; Gratton et al., 1997; Morren et al., 2004). Since the nature of this fast NIN signal is an area of active investigation, close integration of NIN measurements with more direct EEG and MEG measurements of neuronal activity could lead to a fuller understanding of the nature of this optical fast signal. Integration with new, high-speed, MRI acquisition techniques (Lin et al., 2008a,b) may also help shed light on the nature of this optical fast signal and whether or not there might be analogous fast hemodynamic signal modulations detectable using MRI.

ADVANCED VOLUME VISUALIZATION

Combining structural and functional neuroimaging results requires advanced volume visualization tools. Thus far, we have sought to capitalize on the popularity of the NIfTI file format, as it allows convenient utilization of a range of existing MRI 3D visualization packages. However, with the development of Python neuroimaging tools such as NiPy (NiPy Development Team, 2006), as well as the impressive capabilities afforded by Python bindings to both the Visualization Toolkit (via vtk's own Python bindings, or via Enthought's tvtk) and OpenGL (via PyOpenGL), adding native Python 3D visualization for neuroimaging is expected in the near future. Incorporating 3D display capabilities in NinPy would facilitate the sorts of flexible and customized visualization often absent in existing packages. Visualization in three dimensions is often critical to developing better insights into the structure of high-dimensional datasets. The ease with which customization can be made with Python scripting, coupled to a high-level visualization package, is expected to be widely adopted in a broad array of neuroimaging data visualization applications.

CONCLUSION

The relatively short time needed to construct the NinPy suite of tools was made possible given the substantial prior efforts reflected in the packages listed in **Table 3**. Thanks to these developments, we can foresee completion of an end-to-end, Python solution for

Table 3 | Versions utilized and website information for major modules and tools used in the NinPy tool suite.

Module	Version	Website
cgkit	2.0.0a7	http://cgkit.sourceforge.net
chaco/traits	2.5.2001	http://www.enthought.com/products/epd.php
matplotlib	0.98.3	http://matplotlib.sourceforge.net/
mlabwrap	1.0	http://mlabwrap.sourceforge.net/
numpy	1.0.4	http://www.numpy.org/
psychopy	0.97.0	http://www.psychopy.org
pynifti	0.20090303.1	http://niftilib.sourceforge.net/pynifti/
pyparallel	0.2	http://pyserial.wiki.sourceforge.net/pySerial
pyserial	2.2	http://pyserial.wiki.sourceforge.net/pySerial
R	2.8.0	http://www.r-project.org/
rpy	2.0.1	http://rpy.sourceforge.net/
scipy	0.6.0	http://www.scipy.org/

REFERENCES

- Arridge, S. R. (1999). Optical tomography in medical imaging. *Inverse Probl.* 15, R41–R93.
- Berkes, P., Wilbert, N., and Zito, T. (2008). Modular toolkit for data processing (version 2.3). Available at: <http://mdp-toolkit.sourceforge.net> (Retrieved September 2, 2008).
- Boas, D. A. (2004). Photon migration imaging toolbox. Available at: <http://www.nmr.mgh.harvard.edu/PMI/resources/tmcimg/index.htm> (Retrieved August 25, 2008).
- Boas, D. A. (2008). Monte Carlo photon transport. Available at: <http://www.nmr.mgh.harvard.edu/PMI/resources/tmcimg/index.htm> (Retrieved August 25, 2008).
- Choi, J., Wolf, M., Toronov, V., Wolf, U., Polzonetti, C., Hueber, D., Safonova, L. P., Gupta, R., Michalos, A., Mantulin, W., and Gratton, E. (2004). Noninvasive determination of the optical properties of adult brain: near-infrared spectroscopy approach. *J. Biomed. Opt.* 9, 221–229.
- Cox, R. W., Ashburner, J., Breman, H., Fissell, K., Haselgrove, C., Holmes, C. J., Lancaster, J. L., Rex, D. E., Smith, S. M., Woodward, J. B., and Strother, S. C. (2004). A (sort of) new image data format standard: NIfTI-1. 10th Annual Meeting of the Organization for Human Brain Mapping, Budapest, Hungary.
- Dale, A. M. (1999). Optimal experimental design for event-related fMRI. *Hum. Brain Mapp.* 8, 109–114.
- Delpy, D. T., Cope, M., van der Zee, P., Arridge, S., Wray, S., and Wyatt, J. (1988). Estimation of optical path-length through tissue from direct time of flight measurement. *Phys. Med. Biol.* 33, 1433–1442.
- Enthought (2007). Chaco. Available at: <http://code.enthought.com/chaco/> (Retrieved August 25, 2008).
- Enthought (2008). Traits. Available at: <http://code.enthought.com/projects/traits/> (Retrieved August 25, 2008).
- Farrell, T. J., Patterson, M. S., and Wilson, B. (1992). A diffusion theory model of spatially resolved, steady-state diffuse reflectance for the

developing, conducting, analyzing and displaying the results of NIN experiments. Key enabling technologies that have appeared over the past few years include the stabilization of numeric arrays and processing (NumPy), the advancement and continuing stabilization of a broad base of scientific algorithms (SciPy), the development of a robust interface to the R statistical modeling package (RPy), and substantial advances in the mechanisms for stimulus, array and volume visualization (e.g., PsychoPy, Matplotlib and Chaco). We have found that the use of Python as the core programming language for our NIN programs provides significantly better control over most aspects of an NIN experiment than is possible with existing packages. Importantly, our development efforts have not required any time-consuming coding or debugging in C, nor do users need to learn multiple programming or scripting languages to complete a functional neuroimaging experiment. We have found that, particularly for complex problems including optical image reconstruction, hierarchical statistical analysis, or volume visualization, Python can serve as a convenient, powerful, and maintainable scripting “glue”. This architecture allows us to rapidly deploy an operational end-to-end Python solution, allowing later conversion of non-Python algorithms as resources and motivation permit. Reducing our dependence on multiple separate software tools or programming languages for stimulus presentation, data acquisition, data analysis, image reconstruction, statistical modeling, and graphical display greatly simplifies the experimental working environment, and has substantially increased scientific productivity. In addition, the single-language solution facilitates the development and distribution of easy-to-use, self-contained packages for conducting NIN experiments in mobile or remote settings where a dedicated experimenter may not be available. As more open-source tools are ported to Python, further improvements in productivity are envisioned.

We are releasing the source code for all of the NinPy modules for unrestricted use as each sub-module reaches beta level software quality. Completed modules will be available under BSD licensing¹⁰, or by contacting the authors.

ACKNOWLEDGMENTS

We would like to acknowledge support from the National Space Biomedical Research Institute through NASA Cooperative Agreement NCC 9-58.

¹⁰www.nmr.mgh.harvard.edu/Neural_Systems_Group/software.html

- noninvasive determination of tissue optical properties in vivo. *Med. Phys.* 19, 879–888.
- Franceschini, M. A., and Boas, D. A. (2004). Noninvasive measurement of neuronal activity with near-infrared optical imaging. *Neuroimage* 21, 372–386.
- Gratton, G., Joseph, D. K., Huppert, T. J., Diamond, S. G., and Boas, D. A. (2006). Diffuse optical imaging of the whole head. *J. Biomed. Opt.* 11, 054007.
- Gratton, G., Fabiani, M., Corballis, P. M., Hood, D. C., Goodman-Wood, M. R., Hirsch, J., Kim, K., Friedman, D., and Gratton, E. (1997). Fast and localized event-related optical signals (EROS) in the human occipital cortex: comparisons with the visual evoked potential and fMRI. *Neuroimage* 6, 168–180.
- Hanke, M. (2008). PyNifti – Python-style access to Nifti and ANALYZE files. Available at: <http://niftilib.sourceforge.net/pynifti/> (Retrieved August 25, 2008).
- Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *Comput. Sci. Eng.* 9, 90–95.
- Huppert, T. J. (2006). HomER. Available at: <http://www.nmr.mgh.harvard.edu/DOT/resources/homer/home.htm> (Retrieved September 2, 2008).
- Huppert, T. J., Hoge, R. D., Diamond, S. G., Franceschini, M. A., and Boas, D. A. (2006). A temporal comparison of BOLD, ASL, and NIRS hemodynamic responses to motor stimuli in adult humans. *Neuroimage* 29, 368–382.
- Jacques, S. L., and Wang, L. (1995). Monte Carlo modeling of light transport in tissues. In *Optical-Response of Laser-Irradiated Tissue*, A. J. Welch and J. C. van Gemert, eds (New York, NY, Plenum), pp. 73–100.
- Jasdzewski, G., Strangman, G., Wagner, J., Kwong, K. K., Poldrack, R. A., and Boas, D. A. (2003). Differences in the hemodynamic response to event-related motor and visual paradigms as measured by near-infrared spectroscopy. *Neuroimage* 20, 479–488.
- Jobsis, F. F. (1977). Non-invasive, infra-red monitoring of cerebral O₂ sufficiency, blood volume, HbO₂-Hb shifts and blood flow. *Acta Neurol. Scand. Suppl.* 64, 452–453.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: open source scientific tools for python. Available at: <http://www.scipy.org> (Retrieved August 25, 2008).
- Kohri, S., Hoshi, Y., Tamura, M., Kato, C., Kuge, Y., and Tamaki, N. (2002). Quantitative evaluation of the relative contribution ratio of cerebral tissue to near-infrared signals in the adult human head: a preliminary study. *Physiol. Meas.* 23, 301–312.
- Leung, T. S., Elwell, C. E., and Delpy, D. T. (2005). Estimation of cerebral oxy- and deoxy-haemoglobin concentration changes in a layered adult head model using near-infrared spectroscopy and multivariate statistical analysis. *Phys. Med. Biol.* 50, 5783–5798.
- Liechti, C. (2008). pySerial/pyParallel. Available at: <http://pyserial.wiki.sourceforge.net/pySerial> (Retrieved August 25, 2008).
- Lin, F., Witzel, T., Mandeville, J., Polimeni, J., Zeffiro, T., Greve, D., Wiggins, G., Wald, L., and Belliveau, J. (2008a). Event-related single-shot volumetric functional magnetic resonance inverse imaging of visual processing. *Neuroimage* 42, 230–247.
- Lin, F. H., Witzel, T., Zeffiro, T. A., and Belliveau, J. W. (2008b). Linear constraint minimum variance beam-former functional magnetic resonance inverse imaging. *Neuroimage* 43, 297–311.
- Logothetis, N. K. (2008). What we can do and what we cannot do with fMRI. *Nature* 453, 869–878.
- Logothetis, N. K., Pauls, J., Augath, M., Trinath, T., and Oeltermann, A. (2001). Neurophysiological investigation of the basis of the fMRI signal. *Nature* 412, 150–157.
- Logothetis, N. K., and Wandell, B. A. (2004). Interpreting the BOLD signal. *Annu. Rev. Physiol.* 66, 735–769.
- Moriera, W., and Warnes, G. R. (2004). Rpy, a robust Python interface to the R Programming Language. Available at: <http://rpy.sourceforge.net/> (Retrieved September 2, 2008).
- Morren, G., Wolf, U., Lemmerling, P., Wolf, M., Choi, J. H., Gratton, E., De Lathauwer, L., and Van Huffel, S. (2004). Detection of fast neuronal signals in the motor cortex from functional near infrared spectroscopy measurements using independent component analysis. *Med. Biol. Eng. Comput.* 42, 92–99.
- NiPy Development Team (2006). NiPy: neuroimaging tools for python. Available at: <http://neuroimaging.scipy.org> (Retrieved September 2, 2008).
- Okada, E., and Delpy, D. T. (2003). Near-infrared light propagation in an adult head model. I. Modeling of low-level scattering in the cerebrospinal fluid layer. *Appl. Opt.* 42, 2906–2914.
- Oliphant, T. E. (2006). Guide to NumPy. Spanish Fork, UT, Trelgol Publishing.
- Peirce, J. W. (2008). Generating stimuli for neuroscience using PsychoPy. *Front. Neuroinformatics* 2, 10.
- Pinheiro, J. C., and Bates, D. M. (2000). Mixed-effects models in S and S-Plus. New York, NY, Springer.
- Pogue, B. W., McBride, T. O., Osterberg, U. L., and Paulsen, K. D. (1999a). Comparison of imaging geometries for diffuse optical tomography of tissue. *Opt. Express* 4, 270–286.
- Pogue, B. W., McBride, T. O., Prewitt, J., Osterberg, U. L., and Paulsen, K. D. (1999b). Spatially variant regularization improves diffuse optical tomography. *App. Opt.* 38, 2950–2961.
- R Development Core Team (2005). R: a language and environment for statistical computing. Available at: <http://www.R-project.org> (Retrieved August 25, 2008).
- Schmolck, A. (2007). Mlabwrap v1.0. Available at: <http://mlabwrap.sourceforge.net/> (Retrieved August 25, 2008).
- Strangman, G., Boas, D. A., and Sutton, J. P. (2002a). Noninvasive brain imaging using near infrared light. *Biol. Psychiatry* 52, 679–693.
- Strangman, G., Culver, J. P., Thompson, J. H., and Boas, D. A. (2002b). A quantitative comparison of simultaneous BOLD fMRI and NIRS recordings during functional brain activation. *Neuroimage* 17, 719–731.
- Strangman, G., Franceschini, M. A., and Boas, D. A. (2003). Factors affecting the accuracy of near-infrared spectroscopy concentration calculations for focal changes in oxygenation parameters. *Neuroimage* 18, 865–879.
- Strangman, G., Goldstein, R., Rauch, S. L., and Stein, J. (2006). Near-infrared spectroscopy and imaging for investigating stroke rehabilitation: test-retest reliability and review of the literature. *Arch. Phys. Med. Rehabil.* 87, 12–19.
- Strangman, G. E., O'Neil-Pirozzi, T. M., Goldstein, R., Kelkar, K., Katz, D. I., Burke, D., Rauch, S. L., Savage, C. R., and Glenn, M. B. (2008). Prediction of memory rehabilitation outcomes in traumatic brain injury by using functional magnetic resonance imaging. *Arch. Phys. Med. Rehabil.* 89, 974–981.
- Straw, A. D. (2008). Vision egg: an open-source library for realtime visual stimulus generation. *Front. Neuroinformatics* 2, 4.
- Villringer, A., and Chance, B. (1997). Non-invasive optical spectroscopy and imaging of human brain function. *Trends Neurosci.* 20, 435–442.
- Ye, J. C., Tak, S., Jang, K. E., Jung, J., and Jang, J. (2009). NIRS-SPM: statistical parametric mapping for near-infrared spectroscopy. *Neuroimage* 44, 428–447.
- Zhang, Q., Brown, E. N., and Strangman, G. E. (2007a). Adaptive filtering for global interference cancellation and real-time recovery of evoked brain activity: a Monte Carlo simulation study. *J. Biomed. Opt.* 12, 044014.
- Zhang, Q., Brown, E. N., and Strangman, G. E. (2007b). Adaptive filtering to reduce global interference in evoked brain activity detection: a human subject case study. *J. Biomed. Opt.* 12, 064009.
- Zhang, Y., Brooks, D. H., Franceschini, M. A., and Boas, D. A. (2005). Eigenvector-based spatial filtering for reduction of physiological interference in diffuse optical imaging. *J. Biomed. Opt.* 10, 11014.

Conflict of Interest Statement: Quan Zhang and Gary E. Strangman have a patent pending on technologies related to mobile neuroimaging.

Received: 11 September 2008; paper pending published: 11 February 2009; accepted: 30 April 2009; published online: 29 May 2009.

Citation: Strangman GE, Zhang Q and Zeffiro T (2009) Near-infrared neuroimaging with NinPy. *Front. Neuroinform.* (2009) 3:12. doi: 10.3389/neuro.11.012.2009

Copyright © 2009 Strangman, Zhang and Zeffiro. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Network features and pathway analyses of a signal transduction cascade

Ryoji Yanashima^{1,2}, Noriyuki Kitagawa^{1,2}, Yoshiya Matsubara^{1,2}, Robert Weatheritt³, Kotaro Oka⁴, Shinichi Kikuchi^{1,5*}, Masaru Tomita^{1,5} and Shun Ishizaki⁵

¹ Institute for Advanced Biosciences, Keio University, Japan

² Graduate School of Media and Governance, Keio University, Japan

³ Department of Biology, Chemistry and Computer Science, University of York, UK

⁴ Department of Biosciences and Informatics, Faculty of Science and Technology, Keio University, Japan

⁵ Faculty of Environment and Information Studies, Keio University, Japan

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Marcus Kaiser, Newcastle University,
UK

Bruce Southey, University of Illinois,
USA

*Correspondence:

Shinichi Kikuchi, Institute for Advanced
Biosciences and Faculty of
Environment and Information Studies,
Keio University, Endo 5322, Fujisawa
252-8520, Japan.
e-mail: kikuchi@sfc.keio.ac.jp

The scale-free and small-world network models reflect the functional units of networks. However, when we investigated the network properties of a signaling pathway using these models, no significant differences were found between the original undirected graphs and the graphs in which inactive proteins were eliminated from the gene expression data. We analyzed signaling networks by focusing on those pathways that best reflected cellular function. Therefore, our analysis of pathways started from the ligands and progressed to transcription factors and cytoskeletal proteins. We employed the Python module to assess the target network. This involved comparing the original and restricted signaling cascades as a directed graph using microarray gene expression profiles of late onset Alzheimer's disease. The most commonly used method of shortest-path analysis neglects to consider the influences of alternative pathways that can affect the activation of transcription factors or cytoskeletal proteins. We therefore introduced included *k*-shortest paths and *k*-cycles in our network analysis using the Python modules, which allowed us to attain a reasonable computational time and identify *k*-shortest paths. This technique reflected results found *in vivo* and identified pathways not found when shortest path or degree analysis was applied. Our module enabled us to comprehensively analyse the characteristics of biomolecular networks and also enabled analysis of the effects of diseases considering the feedback loop and feedforward loop control structures as an alternative path.

Keywords: signal transduction, Alzheimer's disease, network analysis, *k*-shortest path analysis, python, network robustness, graph theory, hippocampal CA1

INTRODUCTION

Network analysis has lead to the discovery of new components of the metabolic pathways in metabolic pathways and in signal transduction cascades. Examples of network analysis models include the small-world network model (Jeong et al., 2000), in which the average path length is shortened, and the scale-free network model (Wuchty, 2001), which has a degree distribution that follows a power law. Multilayer structural and motif analyses (Milo et al., 2002; Shen-Orr et al., 2002) have shown that metabolic pathways and protein interactions have more notable cluster structures (Ravasz et al., 2002) than random networks, and that metabolic and signaling pathways behave like complex regulatory networks. In recent research on diseases, network analyses, like degree analysis of cancer-related genes using gene regulatory networks to identify the genes (Futreal et al., 2004) and various other analyses of disease genes, revealed structural effects of disease on biomolecular networks (Ideker and Sharan, 2008). Taken together, these findings suggest that cellular functions can be modelled as network structures and that investigation of disease phenomena through network analysis has the potential to reveal novel properties and pathways in biomolecular pathways associated with disease states?

The studies mentioned above assume that proteins do not change in the absence of external stimulation. Proteins in networks

are known to be regulated by gene expression patterns, as well as adapting to the external environment (Luscombe et al., 2004). To characterize the dynamic nature of protein networks, investigations into the effects of diseases on gene expression have been initiated for Alzheimer disease by means of diffusion kernels and microarray data (Ma et al., 2007) and for cancer by means of gene expression data and network information (Chuang et al., 2007). However, because networks function as multiple-complex regulatory structures, it is insufficient to study disease dynamics in protein networks through analysis of a single factor affecting the network or through analysis of structural properties.

In the present study, we investigated the protein networks associated with Alzheimer's disease through feature analysis of regulated signal molecules, as well as by structural analysis of network component. Intraneuronal amyloid β ($A\beta$) is reported to be a major important factor for Alzheimer's disease. $A\beta$, which is the product of the protein catabolic enzyme, is normally transported out of cells (Iwata et al., 2000). In Alzheimer's disease the aggregation and deposition of insoluble $A\beta$ leads to nerve cell damage and is thought to be the pathogenic mechanism of Alzheimer's disease (Hardy and Selkoe, 2002). Studies of $A\beta$ and protein catabolic enzymes, like β -secretase, have focused on changes in certain proteins. Although a few studies have focused on the entire network, the mechanism

underlying the accumulation of A β has not been discovered. Thus, it is still unclear if the accumulation of A β is the direct cause of Alzheimer's disease (Heneka and O'Banion, 2007). Here, we aimed to use a network model to discover the characteristics of structures that most affect the hippocampal signal transduction pathway, and the regulatory mechanisms controlling gene expression in Alzheimer's disease. We generated a network model for the Alzheimer's disease patient signal transduction cascade, referred to as the Alzheimer's disease network ("ADN"), from the signal transduction pathway in the hippocampal CA1 region (Ma'ayan et al., 2005) and from gene expression data derived from patients with late onset Alzheimer's disease (Liang et al., 2007).

In order to understand the network form, we conducted feature analysis of signal molecules in the signal transduction cascade by measuring k -core, degree, closeness, betweenness, the change in the average shortest path length, and the change in the articulation points, following the removal of the Alzheimer's-related signal molecules from the network. In our structural analysis of the network, we considered the network density, average clustering index, and average shortest path length. Regulatory structures, like the feedback loop and the feedforward loop, are more frequent in hippocampal signaling pathways than in the randomly generated networks (Ma'ayan et al., 2005). Therefore, we analysed feedback loops and feedforward loops in the model network using the k -cycle structure (Nochomovitz and Li, 2006). The k -cycle structure is defined as a network structure in which duplicating nodes are removed from the network when one node to the in-neighbours can be reached by the k -step. For analysis of pathway characteristics, the extracellular ligand was set as the input and cytoskeletal proteins and transcription factors were set as the output. Since there are many alternative signal transduction pathways (Coulson, 2006), we used the k -shortest pathway (Rahman and Schomburg, 2006)

instead of the shortest path or path length for pathway analysis. With our model we were able to reproduce the Alzheimer's disease shift in gene expression in the hippocampal signal transduction pathway and the shift in signal transduction in Alzheimer's disease revealed in earlier studies.

MATERIALS AND METHODS

ANALYSIS PACKAGE FOR BIOMOLECULAR NETWORKS

In our study, we developed the network analysis module "Analysis Package for Biomolecular Networks (BioNetpy)" using the Python software program. Python is suitable as an open resource because it excels in readability over other program languages and has superior system execution by utilizing the just-in-time compiler, psyco¹. The BioNetpy module was constructed using the Python network analysis module NetworkX-0.3.6² and igraph-0.4.5³. We also used the numerical package Numpy-1.0.4, which is a Python numerical module⁴. The BioNetpy module performs the three analysis methods outlined in **Figure 1**.

BioNetpy and Supplementary Material can be downloaded from the following website: <http://medcd.iab.keio.ac.jp/bionetpy/>; <http://www.frontiersin.org/neuroinformatics/paper/10.3389/neuro.11/013.2009>.

ANALYSIS OF GENE EXPRESSION DATA FOR MODEL ASSEMBLY

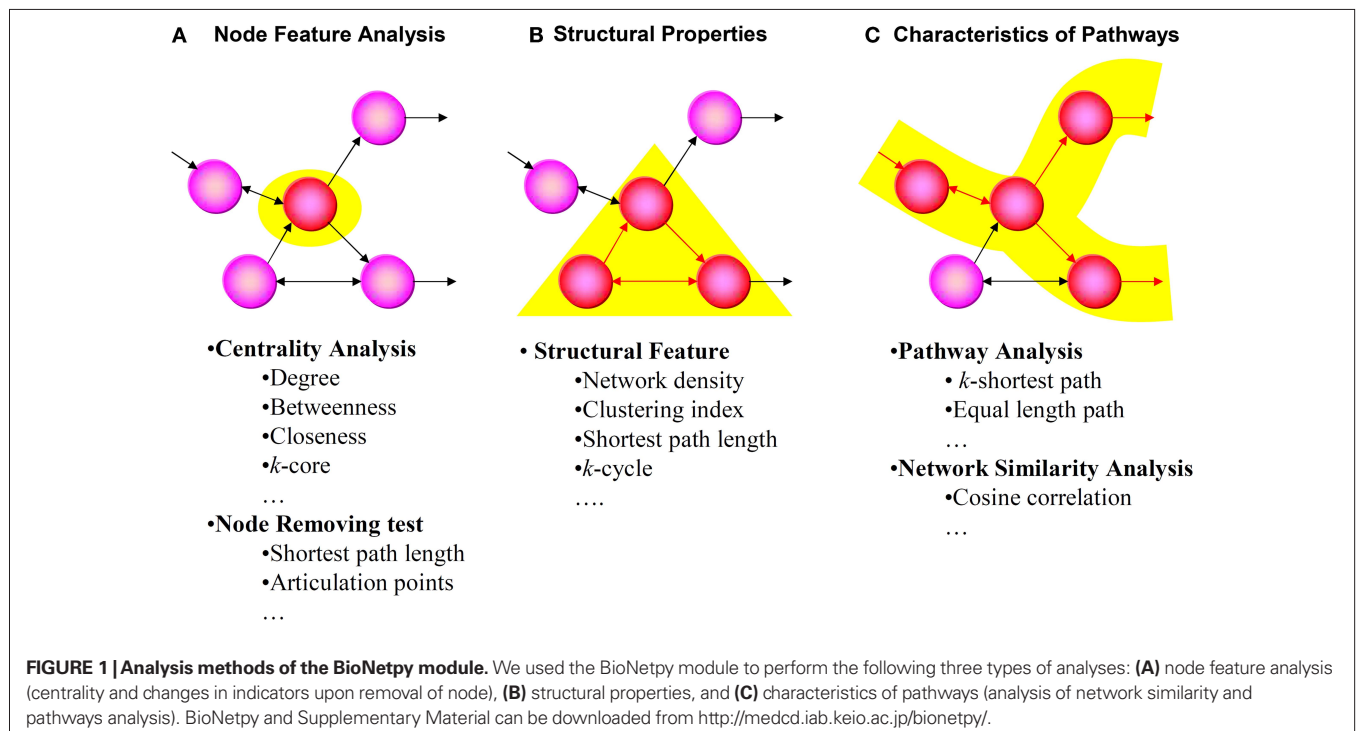
We used a network expressed by a directed graph of the signal transduction pathway of the hippocampal CA1 region in humans (Ma'ayan et al., 2005). This network contains 570 nodes (signal molecules)

¹<http://psyco.sourceforge.net/>

²<https://networkx.lanl.gov/>

³<http://igraph.sf.net/>

⁴<http://numpy.scipy.org/>



and 1,333 edges (reactions). The edges can be categorized into three types of information defined as active, inactive, and bidirected (bidirectional activation or inactivation) information. We extracted gene expression data derived from GeneChip (Affymetrix) analysis of human hippocampal CA1 region. We applied the Bioconductor 2.2 program to analyse gene expression data (Reimers and Carey, 2006). Bioconductor can be applied to the Python module by using the Rpy program⁵. We used the Human Genome U133 Plus 2.0 Array from the Bioconductor *affy* package (Gautier et al., 2004). We extracted Alzheimer's disease-related genes by analyzing GSE5281, which is a set of gene expression data derived from patients with late-onset Alzheimer's ($n = 10$) and controls ($n = 13$) (Liang et al., 2007) that has been recorded on the GEO database. We normalized the data by the distribution-free summarization method, which has been tested with the Spike-ins benchmark test on the Human Genome U133 Plus 2.0 Array and is known for its high-resolution summarization of microarray data (Chen et al., 2007). After data normalization, we used the Bioconductor *limma* package (Smyth, 2004) to define genes as Alzheimer's disease-related genes within the $P < 0.005$ threshold by employing the empirical Bayes t -statistic test (Jeffery et al., 2006). We matched genes and the corresponding signal molecules by correlating information from the NCBI Gene ID (Maglott et al., 2007) and Swiss-Prot ID (Bairoch et al., 2004) and defined signal molecules coded by Alzheimer's disease-related genes as Alzheimer's disease-related signal molecules. We conducted feature analyses by measuring k -core, betweenness centrality, closeness centrality, and degree centrality. We also analysed changes in the shortest path length, which is an indicator of a small-world network (Mason and Verwoerd, 2007), and changes in articulation points, which is an indicator of network connectivity, after removing nodes from the Alzheimer's disease-related signal molecule network. The k -core of a graph is the maximal subgraph in which each node's degree is at least k . Betweenness centrality measures the importance of a node within a network. Nodes that occur on many short paths between other nodes have higher betweenness centrality than those nodes that do not. Closeness centrality is defined as the number of nodes minus one divided by the sum of the lengths of all shortest path lengths from and to the given node. Degree centrality is the number of nodes that a given node is connected to. We were able to analyse the characteristics of signal molecules in the network on multiple dimensions using these indicators.

STRUCTURAL PROPERTIES OF HIPPOCAMPAL PATHWAYS OF PATIENTS WITH ALZHEIMER'S DISEASE

We conducted a structural index analysis by generating an ADN after removing Alzheimer's disease-related signal molecules from the control network ("CN"). We used a k -cycle structure for the analysis of feedback loop in the networks. The k -cycle structure is defined as a network structure from which duplicating nodes are removed when one node can be reached from the in-neighbors. An earlier study (Ma'ayan et al., 2005) and our pilot study shows that 90% of all nodes can be reached within 9 steps for input ($n = 30$). Thus, we defined pathways within 9 steps of each other to be important for intercellular signal transduction. Because network structure depends on the number of nodes, we generated

a randomly removed network ("RRN") by removing nodes from the CN to equal the number of nodes of the ADN. We then limited the network density, average clustering index, and average shortest pathway length change of this new CN to 5% and compared the results. The k -cycle data can be analysed according to Eq. 1:

$$C = \sum_{n=1}^k \frac{\text{cycle}_n(\text{Node}_i)}{n} \quad (1)$$

where C_k represents the number of k -cycle structures in the network. The function cycle_n represents the number of cycle structures can be reached from the in-neighbors.

CHARACTERISTICS OF HIPPOCAMPAL SIGNAL PATHWAYS IN PATIENTS WITH ALZHEIMER'S DISEASE

Cellular processes are controlled by many alternate signal transduction pathways (Coulson, 2006). For this reason, we analysed the k -shortest pathway instead of analyzing pathway length or shortest pathways. We also generated an RRN and compared the k -cycle of the RRN with that of the ADN. Through exploration of the k -shortest path length, the number of pathways was carried out by calculating the shortest pathway length between nodes and by using Depth-First Iterative-Deepening (Korf, 1987). We used the k -shortest pathway with extracellular ligands ($n = 30$) as input and cytoskeletal proteins ($n = 24$) and transcription factors ($n = 35$) as output to define 1,770 pathways for analysis. We defined the input and output of two important functions of the neural cell, neuronal plasticity and neurite outgrowth, to analyse the effects of Alzheimer's disease on neural functions. Neuronal plasticity is controlled by depolarization of the postsynaptic cell by binding of glutamate to its receptors. Consistent with the network analysis described above, activation of these receptors activates the cAMP response element-binding protein (CREB), thus increasing the level of amino-3-hydroxy-5-methyl-4-isoxazolepropionic acid (AMPA) receptor (Hayashi et al., 2000). For these reasons, we set glutamate as the starting point of the pathway and CREB as the endpoint for the neuronal plasticity pathway. The direction of neurite outgrowth is determined by guidance factors (Dickson, 2002). Therefore, we set the guidance factors acetylcholine (ACh), insulin-like growth factor I (IGF1), nerve growth factor (NGF), and Ephrin at the start of the pathway, and tubulin, a microtubule protein, at the endpoint. An evaluation of robustness, defined in Eq. 2, was conducted by comparing the robustness values of all inputs and outputs of the ADN with that of the CN and RRN.

We also conducted a k -shortest pathway analysis of the pathways involved in neural cell death, the pathways that link directly to the amyloid β protein precursor (APP), and the pathways that link extracellular ligands to transcription factors or cytoskeletal proteins. Neuronal cells are known to enter apoptosis readily upon receiving signals of extracellular death ligands or DNA damage (Jellinger, 2006). We defined the starting points of the neural death pathway as fas ligand (FasL) and tumor necrosis factor- α (TNF α), which induce apoptosis, and the endpoint as the DNA fragmentation factor (ICAD), an inhibitor of caspase-activated DNase, which fragments DNA. In addition, we defined the pathways between all ligands and included the APP-binding family A member 1 (MINT-1) (Yoon et al., 2007) and caspase 3 (Su et al., 2002) in the APP-related pathway. These pathways are shorter than that of neuronal plasticity and neurite outgrowth and can traverse from the

⁵<http://rpy.sourceforge.net/>

input to the output through a shorter path. Therefore, we compared the number of pathways having the same input and output set in the total number of pathways and the number of pathways in the RRN in total number of pathways. The number of steps, k , used in the k -shortest pathway analysis in the k -cycle structure, was defined as 9 steps, using the following equation:

$$R_{ij} = \frac{N_{ij} - \text{mean}_x}{\text{SD}_x} \quad (2)$$

where R is the robustness value (R -value) of the pathway. In the pathways from glutamate to CREB and ACh, NGF, IGF1 and from Ephrin to tubulin, R is the difference between the numbers of k -shortest paths obtained by all inputs to outputs in all k -shortest path sets, which is defined as X . In the pathways from FasL and TNF α to ICAD, including all inputs to MINT-1 and caspase 3, R is the difference in the number of k -shortest paths between node i and node j obtained in the RRN sets, which is defined as X in this case. N_{ij} is the k -shortest path number from node i to node j in the network of interest. Mean_x is the mean of all k -shortest path sets or nodes in the RRN sets. SD_x is the standard deviation of all k -shortest path sets or nodes in the RRN sets.

Equation 3 below shows the interpretation of network similarity using a single value (Barrett et al., 2006) for the vector space of inputs and outputs in a network using a matrix expression for equal-length shortest path (Borgwardt and Kriegel, 2005), which indicates pathways with equal steps. Our study analyzes the change in the entire pathway at step e .

$$S = \arccos \left(\frac{\vec{c^e} \cdot \vec{o^e}}{|\vec{c^e}| \cdot |\vec{o^e}|} \right) \quad (3)$$

where S represents network similarity between the first mode of singular value c (equal-length shortest-path matrix of CN) and o (equal-length shortest-path matrix of ADN or RRN); e represents the specific step value of the equal-length shortest-path matrix.

RESULTS

FEATURE ANALYSIS OF SIGNAL MOLECULES

Through empirical Bayes t -statistics, we extracted 76 Alzheimer's disease-related genes known to downregulate actin (Harigaya et al., 1996) and beta-catenin (Li et al., 2007), resulting in a decrease in the level of calcium/calmodulin-dependent protein kinase type II (CaMKII) (Allison et al., 2000). Please refer to the Supplemental Material for a list of genes aforementioned. By observing the pathway functions of the signal molecules encoded by these 76 genes, we found the largest changes in the actual numbers of molecules with Kinase and Adapter functions, and the largest percentage change for nodes in the Receptor and Bcl2Family functional groups, which decreased at rates greater than the rate of change for the network overall (13%; Table 1). We conducted a feature analysis of Alzheimer's disease-related signal molecules and other molecules by measuring k -core, betweenness, closeness, degree, the change in average shortest path length, and the change in articulation points. There were no significant differences in these measurements between Alzheimer's disease-related signal molecules and other molecules ($P < 0.05$, Mann-Whitney U -test; Table 2).

Table 1 | Number of constituent signal molecules on CN and ADN.

"Other" denotes small molecules or histones. The actual connection graph of the 570 nodes and 1,333 edges of CN and the 494 nodes and 974 edges of ADN is shown. We extracted 76 Alzheimer's disease-related signal molecules known to decrease actin, beta-catenin, and CaMKII. This group of genes represents 13% of the CN. By observing the pathway functions of these 76 Alzheimer's disease-related signal molecules, we discovered that nodes in the Bcl2Family and Receptor groups decreased at a rate greater than the network as a whole.

Function	Number of signal molecules in networks		
	ADN	CN	CN-ADN (%)
Adapter	89	103	14 (14)
Kinase	71	86	15 (17)
Receptor	39	51	12 (24)
Transcriptional factor	28	35	7 (20)
Ligand	30	30	0 (0)
Cytoskeletal protein	21	24	3 (13)
Vesicle	17	21	4 (19)
Ion channel	17	20	3 (15)
GEF	19	20	1 (5)
Inhibitor	17	18	1 (6)
GAP	13	13	0 (0)
GTPase	11	13	2 (15)
PDE	9	11	2 (18)
G protein	9	10	1 (10)
Ribosome	10	10	0 (0)
Activator	8	8	0 (0)
Bcl2Family	6	8	2 (25)
Protease	8	8	0 (0)
Phosphatase	15	16	1 (6)
Other	57	65	8 (12)
	494	570	76 (13)

When we removed these Alzheimer's disease-related signal molecules, the ADN contained 494 nodes and 974 edges. In total, 91% of the input-output sets were connected in the CN (average path length = 5.94), and 50% of those sets were connected in the ADN (average path length = 6.68).

k -CYCLE ANALYSIS OF ADN

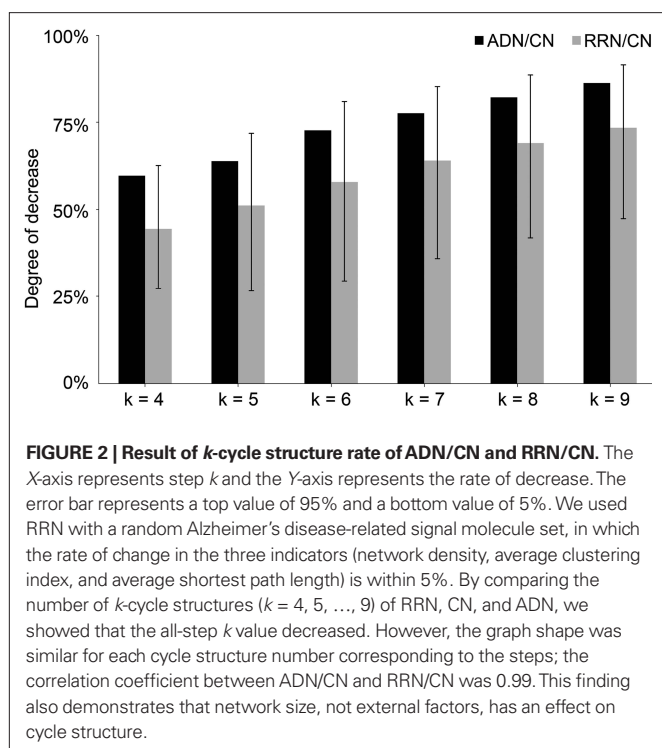
By comparing the number of k -cycle structures ($k = 4, 5, \dots, 9$) of RRN, CN, and ADN, we showed that the all-step k value decreased (Figure 2). However, the graph shape was similar for each RRN and for each cycle structure number corresponding to the steps in the random sampling network; the correlation coefficient between ADN/CN and RRN/CN was 0.99. This finding also demonstrates that network size, not external factors, has an effect on cycle structure.

k -SHORTEST ANALYSIS OF ADN

The k -shortest pathway analysis ($k = 9$) of CN, ADN, and RRN showed no notable difference in distribution shape between all inputs and outputs. There were also no differences in the average network pathway between ADN (67 ± 216) and RRN (144 ± 342)

Table 2 | Network feature analysis of signal molecules. Network feature analysis of Alzheimer's disease-related signal molecules and other signal molecules in the network ("Others") performed by measuring k -core, betweenness, closeness, degree, change in average shortest path length, and change in articulation points (mean \pm SD). There were no significant differences in these measurements between Alzheimer's disease-related signal molecules and other signal molecules in the network ($P < 0.05$, Mann-Whitney U -test). This network feature is the same as that of disease-related molecules defined in earlier studies. IN means the incoming paths OUT means the outgoing paths, and ALL means both incoming and outgoing paths.

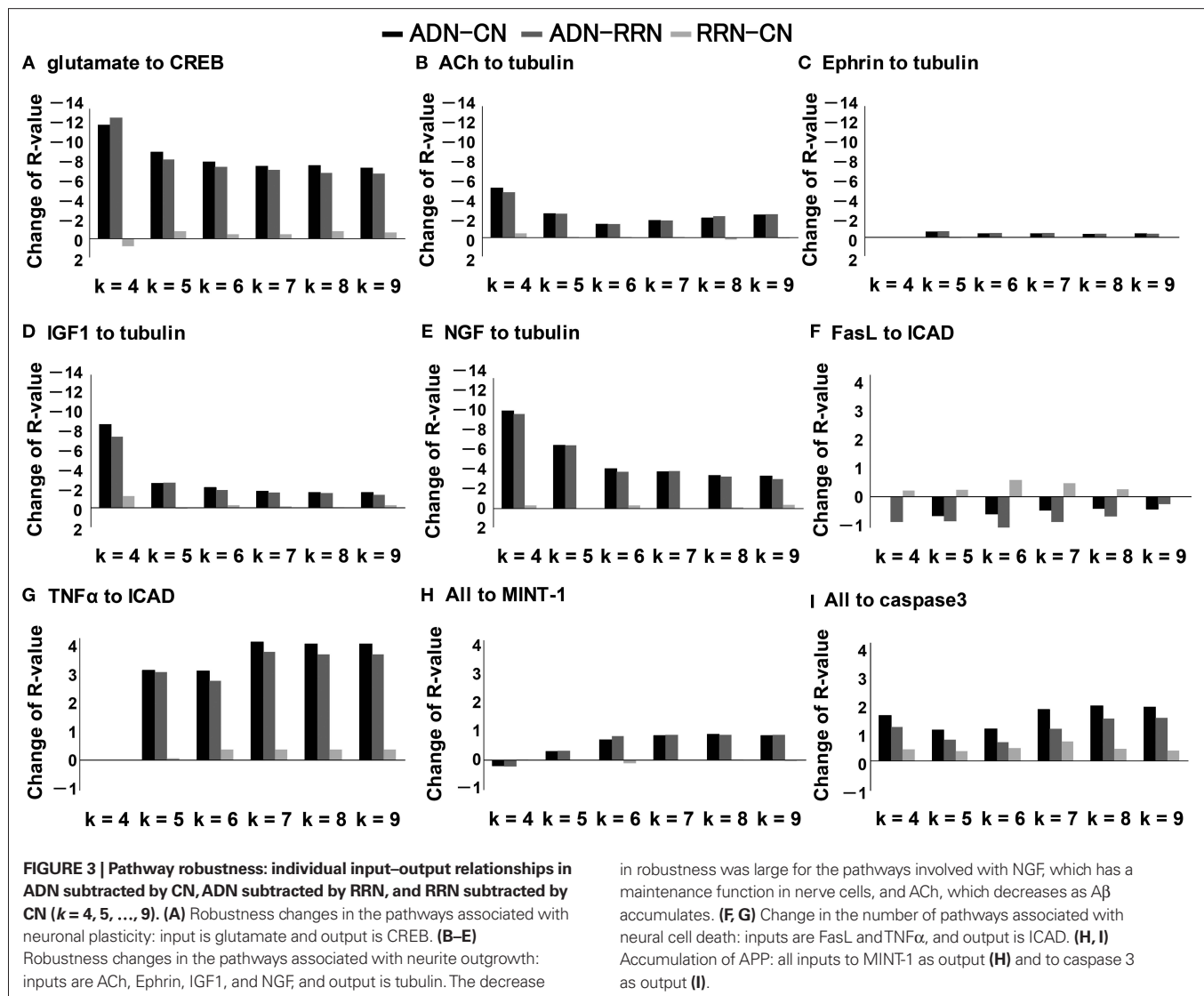
	Centrality analysis				Node removal analysis	
	k -core	Betweenness	Closeness	Degree	Average path length	Articulation point
AD						
ALL	0.61 \pm 1.24	0.006 \pm 0.013	0.21 \pm 0.18	0.012 \pm 0.015	5.453 \pm 0.024	107.78 \pm 0.75
OUT	0.66 \pm 1.05	0.006 \pm 0.013	0.27 \pm 0.30	0.012 \pm 0.015		
IN	2.62 \pm 1.33	0.007 \pm 0.016	0.24 \pm 0.04	0.009 \pm 0.013		
OTHERS						
ALL	0.70 \pm 1.10	0.005 \pm 0.011	0.21 \pm 0.17	0.010 \pm 0.011	5.452 \pm 0.022	107.83 \pm 0.64
OUT	0.76 \pm 1.42	0.005 \pm 0.011	0.21 \pm 0.23	0.010 \pm 0.011		
IN	2.59 \pm 1.25	0.006 \pm 0.013	0.24 \pm 0.04	0.008 \pm 0.009		



at $k = 9$. Thus, there was no difference in the effect of Alzheimer's disease-related signal molecules and random signal molecule on any of the inputs or outputs. Next, we conducted an analysis of change in robustness ($k = 4, 5, \dots, 9$) for pathways associated with neuronal plasticity and neurite outgrowth, and for pathways associated with neuronal death and APP (Figure 3). The change in robustness was the greatest for the pathways associated with neuronal plasticity (Walsh et al., 2002) for ADN subtracted by CN and ADN subtracted by RRN, for each k value. Likewise, for the pathways associated with neurite outgrowth, there was a decrease in robustness for those involving NGF (Tuszynski et al., 2005),

which has a maintenance function in nerve cells, and ACh (Hoshi et al., 1997), which decreases as A β accumulates. For the pathways associated with neuronal plasticity, the decrease in robustness for NGF and ACh was within the top 10% of all combinations. The set that showed the largest change in robustness was the pathway between glutamate and actin signal transduction (R -value was -14.9 , -13.6 and -1.29 for ADN subtracted by CN, ADN subtracted by RRN and RRN subtracted by CN). The same change in robustness for the glutamate to actin signal transduction pathway was observed between ADN and RRN and between ADN and CN. This finding suggests that these changes in robustness do not depend on signal molecule number, network density, the average clustering index, or the average shortest path length. In the analysis of the pathways associated with neural cell death, there were no changes in robustness observed for the FasL to ICAD pathway; however, CN and RRN showed increases in each step of the TNF α to ICAD and caspase 3 pathways. TNF α and caspase 3 correlate positively with the accumulation of A β (Cacquevel et al., 2004; McCusker et al., 2001). Furthermore, these results show that Alzheimer's disease-related signal molecules have more selective effects on neural plasticity and neurite outgrowth than random signal molecules.

Analysis of certain inputs to all outputs showed a large decrease in signal molecules associated with neuregulin (NRG), which is a substrate of BASE1 (Willem et al., 2006); with NGF, which is the drug target in Alzheimer's disease; with reelin, which is thought to be related to Alzheimer's disease (Botella-Lopez et al., 2006); and with dopamine, which is a neurotransmitter (Figure 4). By comparison, epidermal growth factor (EGF) and the neurotrophin family, which includes brain-derived neurotrophic factor (BDNF) and neurotrophin 4 (NT4), showed an increase in associated signal molecules. The level of BDNF is increased in patients with Alzheimer's disease and in the hippocampus of a transgenic mouse model of Alzheimer's disease (Laske et al., 2006; Tang et al., 2000). However, our finding that the R -value of inputs was between 0.8 and -1.2 suggests that the effect of BDNF on robustness in Alzheimer's disease is small. Analysis of all inputs to certain outputs revealed that the largest



decrease in associated signal molecules was for key factors in neural activity, including actin and tubulin, which are cytoskeletal proteins regulating neural plasticity and neurite outgrowth, and CREB, which is a transcription factor (Figure 4). By comparison, transcription factors, such as the nuclear factor of activated T cells (NFAT), and actin-binding proteins, such as α -actinin and profilin, showed an increase in associated signal molecules. Because the R -value range was between 1.2 and -4.3 , the result for the comparison of input to total output implies that Alzheimer's disease affects the expression of output molecules more than input molecules.

The analysis of the change in similarity between the input and output sets of CN, ADN, and RRN, shown as a matrix, indicate that ADN is lower than RRN when $e = 5$ and 9, but higher than RRN when $e = 6, 7$ and 8 (Figure 5).

DISCUSSION

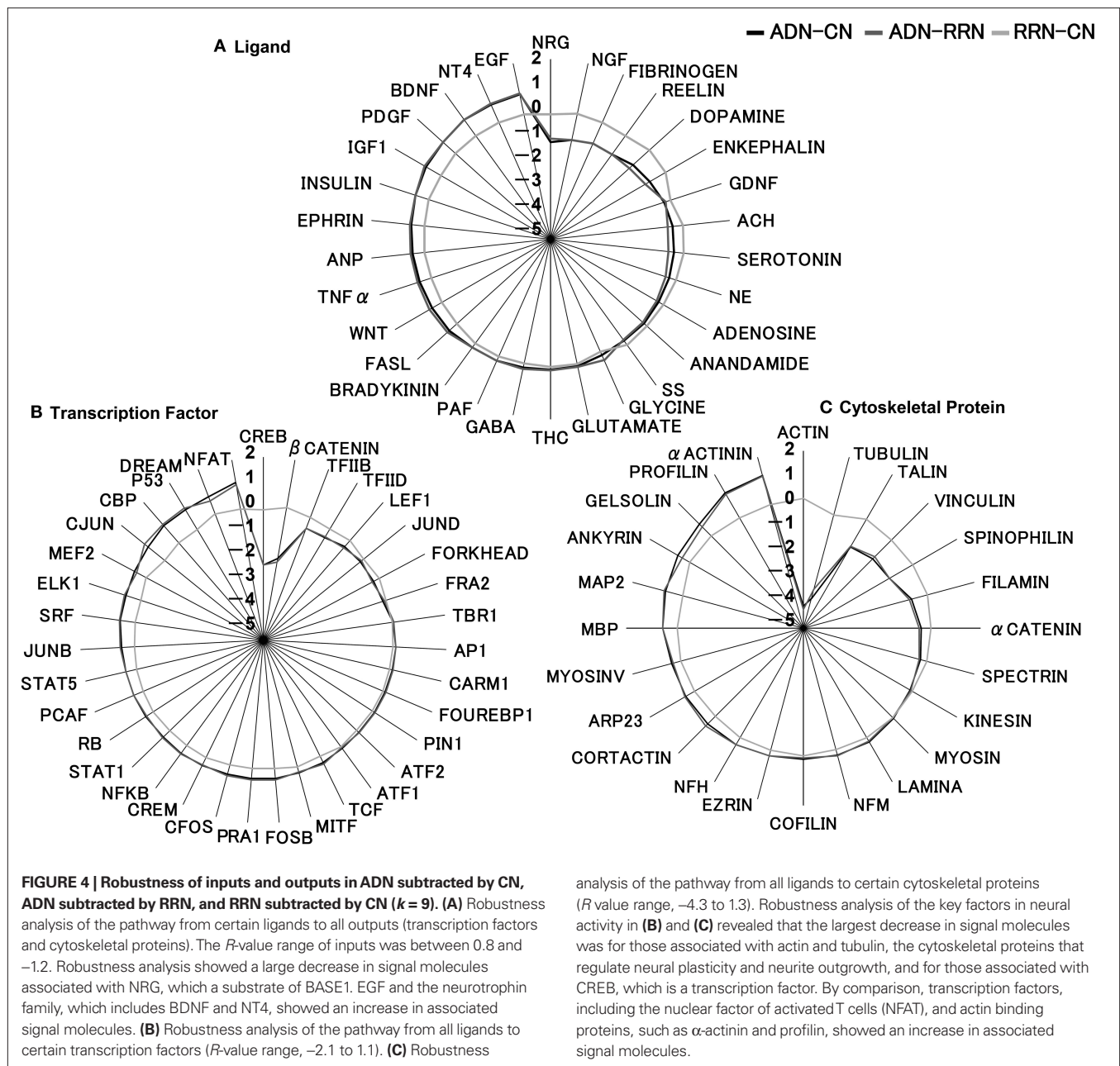
MICROARRAY AND CENTRALITY ANALYSIS OF SIGNAL MOLECULES

In our study, we conducted a feature analysis of Alzheimer's disease-related signal molecules in a network. We conducted the analysis on

in robustness was large for the pathways involved with NGF, which has a maintenance function in nerve cells, and ACh, which decreases as $A\beta$ accumulates. (F, G) Change in the number of pathways associated with neural cell death: inputs are FasL and TNF α , and output is ICAD. (H, I) Accumulation of APP: all inputs to MINT-1 as output (H) and to caspase 3 as output (I).

genes from a large sample of patients in the early stage of late-onset Alzheimer's disease. It is thought that new information on a disease pathogenesis can be gained by observing changes in a signaling pathway produced by the changes in the stages of Alzheimer's disease. Data similar to that used in the present study, namely the registered expression data derived from the hippocampal CA1 region of Alzheimer's patients at different stages (Blalock et al., 2004), may be used for a similar analysis in the future. The data from the aforementioned study covers the four categories of Alzheimer's disease status termed control, incipient, moderate, and severe. Therefore, we believe that we will be able to conduct time-series network analyses of these symptoms. The present study focuses only on gene expression data, yet Alzheimer's disease characteristics not regulated by gene expression may also be considered by using alternative experimental methods, for example, the large-scale databases from other *in vivo* experiments (Bertram et al., 2007) or positron-emission tomography (PET) studies (Tuszynski et al., 2005).

In the feature analysis, we found no significant difference in signal molecules for all indicators. By comparison the average



of indicators including degree and betweenness increased, for Alzheimer's disease-related signal molecules compared to the other signal molecules. This trend is the same as that for characteristic disease-related genes defined in earlier studies (Ideker and Sharan, 2008). Changing the threshold for defining Alzheimer's disease-related genes has an effect on the results of gene expression data analysis. In addition, it is difficult to analyze indicators like degree and betweenness, due to the method of calculating substances at the ends of networks. For this reason, substances like ACh and NGF, which are located at the ends of networks and are targets of drug development, require a combination of signal molecule analysis and pathway analysis that controls the input and output data. Therefore, additional findings on the

pathogenesis of Alzheimer's disease may be discovered through additional feature analysis of networks for data other than gene expression.

NETWORK STRUCTURE

In the analysis of k -cycle structure, we discovered that k -cycle numbers decreased in all steps in the ADN/CN compared with that in RRN/CN and that the rate of decrease increased according to the step number. We also discovered that RRN had more k -cycle structure than ADN. However, since the decreasing rate at each step was the same in ADN and CN, the change in k -cycle number in this study has a larger effect on the network scale than the Alzheimer's disease-related signal molecules. Moreover, the

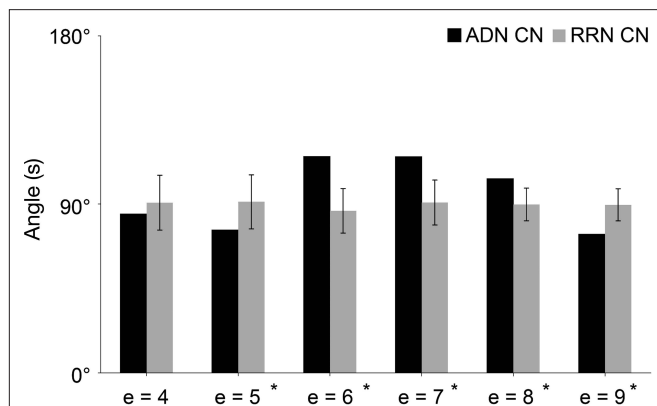


FIGURE 5 | Network similarity analysis of CN, ADN, and RRN. The X-axis represents step e and the Y-axis represents the angle value (S). Error bars represent the SD. The results of the network similarity analysis for the input and output set are converted into a matrix and indicate that ADN is lower than RRN when $e = 5$ and 9 , but is higher than RRN when $e = 6, 7$ and 8 .

reason for the greater change in cycle structure by step number is believed to result from the effect of an increase in the number of nodes, which were randomly moved into the cycle. In future studies, it may be necessary to normalize changes in the network scale to conduct analyses on k -cycle structure. It must be noted that in this study, we focused on feedforward and feedback loops in the results of loop structure.

PATHWAY CHARACTERISTICS

Our k -shortest pathway analysis of pathway characteristics revealed no changes between the R -values of all inputs and outputs and the pathway average. This result suggests that the effect of Alzheimer's disease on the hippocampal signal transduction pathway does not correspond to the number of pathways or to the distribution of k -shortest pathways. We also discovered that the ADN-CN and ADN-RRN sets of Alzheimer's disease-related signal molecules affect specific pathways more selectively than the random sets. In addition, the glutamate-actin pathway plays an important role in the formation of mature spines in the rat brain (Serge et al., 2003), which showed the most significant decrease in R -value, also showed the most significant decrease in the RRN-CN pathway.

In the analysis of inputs, the decrease in robustness of NGF agreed with the decrease in robustness of Alzheimer's disease. Increase in robustness was seen in both NT4 and BDNF. Earlier studies suggested that BDNF tends to increase in early-onset Alzheimer's disease and decrease in late-onset Alzheimer's disease. Also, insulin and IGF1 decreased in our study, but an increase in insulin and IGF1 was thought to occur as the result of an increase in $A\beta$ in prior studies (Cole and Frautschy, 2007). An increase in the level of the EGF receptor and $A\beta$ is reported to be correlated (Zhang et al., 2007), yet we found no evidence of this relationship in the present study. With respect to output factors, there were significant decreases of R -value in cytoskeletal proteins, like actin and tubulin, or in CREB thus suggesting that

Alzheimer's disease selectively affects the neural plasticity and neurite outgrowth. Moreover, increase was seen in actin binding proteins such as α -actinin and profilin. The reason for the decrease in actin might be explained by the tendency of actin-binding proteins to bind other proteins, such as cortactin, cofilin, and β -catenin; thus, actin may perform other functions that are specific to Alzheimer's disease. There was an increase in NFAT in ADN, which is expressed at the same time as BDNF (Groth and Mermelstein, 2003), and thus we believed that changes in NFAT synchronized with the changes in BDNF. In addition, the angle value in CN showed more change by step compared with RRN. This is because the effects of Alzheimer's disease-related signal molecules are different at each step, and further interpretation of each step in the k -shortest pathway will be required in future studies. In our study, we succeeded in indicating changes caused by Alzheimer's disease in signal transduction pathways through analysis of the features of signal molecules and of the properties of pathways in network structures.

CONCLUSION

We conducted a feature analysis on networks of signal molecules regulated by Alzheimer's disease and analysed the properties of the network structure. In our analysis of signal molecules, we found no significant difference in all indicators. Network structure analysis revealed that Alzheimer's disease-related signal molecule sets have a specific effect on the average shortest path length, with effects on motif structures, like feedforward and feedback loops, controlling the functions of neuronal cells. Also, our analyses of pathway characteristics extracted pathways related to neuronal plasticity, neurite outgrowth (including ACh and NGF), and neural death (including the TNF α pathway and caspase 3). In addition, similar changes in R -value in our study were observed for other Alzheimer's disease signal transduction pathways. Similarity and k -shortest analysis of pathways showed that the effect of Alzheimer's disease-related genes on networks depends on steps. This finding indicates that a k -shortest pathway analysis is more useful than a shortest pathway analysis. In summary, the Python module use in the present study enabled us to comprehensively analyse the characteristics of biomolecular networks and to assess the effects of Alzheimer's disease using feedforward and feedback loop control structures as alternative paths.

ACKNOWLEDGEMENTS

We thank Y. Imanishi for initial help with the paper and N. Yachie and H. Nakamura for help on the website. This research was partially supported by Grant-in-Aid for Scientific Research from the Ministry of Education, Science, Culture, and Sport. This research was also supported by JGC-S Scholarship Foundation.

SUPPLEMENTARY MATERIAL

The Supplementary Material for the network edgelist, the table for mapping the network nodes and Swiss-Prot ID and Python module can be found online at <http://medcd.iab.keio.ac.jp/bionetpy/> and <http://www.frontiersin.org/neuroinformatics/paper/10.3389/neuro.11/013.2009>.

REFERENCES

- Allison, D. W., Chervin, A. S., Gelfand, V. I., and Craig, A. M. (2000). Postsynaptic scaffolds of excitatory and inhibitory synapses in hippocampal neurons: maintenance of core components independent of actin filaments and microtubules. *J. Neurosci.* 20, 4545–4554.
- Bairoch, A., Boeckmann, B., Ferro, S., and Gasteiger, E. (2004). Swiss-Prot: juggling between evolution and stability. *Brief. Bioinform.* 5, 39–55.
- Barrett, C. L., Price, N. D., and Palsson, B. O. (2006). Network-level analysis of metabolic regulation in the human red blood cell using random sampling and singular value decomposition. *BMC Bioinformatics* 7, 132.
- Bertram, L., McQueen, M. B., Mullin, K., Blacker, D., and Tanzi, R. E. (2007). Systematic meta-analyses of Alzheimer disease genetic association studies: the AlzGene database. *Nat. Genet.* 39, 17–23.
- Blalock, E. M., Geddes, J. W., Chen, K. C., Porter, N. M., Markesbery, W. R., and Landfield, P. W. (2004). Incipient Alzheimer's disease: microarray correlation analyses reveal major transcriptional and tumor suppressor responses. *Proc. Natl. Acad. Sci. U.S.A.* 101, 2173–2178.
- Borgwardt, K., and Kriegel, H. (2005). Shortest-Path Kernels on Graphs. Data mining. *IEEE Int. Conf. Data Mining* 50, 74–81.
- Botella-Lopez, A., Burgaya, F., Gavin, R., Garcia-Ayllon, M. S., Gomez-Tortosa, E., Pena-Casanova, J., Urena, J. M., Del Rio, J. A., Blesa, R., Soriano, E., and Saez-Valero, J. (2006). Reelin expression and glycosylation patterns are altered in Alzheimer's disease. *Proc. Natl. Acad. Sci. U.S.A.* 103, 5573–5578.
- Cacquevel, M., Lebeurrier, N., Cheenne, S., and Vivien, D. (2004). Cytokines in neuroinflammation and Alzheimer's disease. *Curr. Drug Targets.* 5, 529–534.
- Chen, Z., McGee, M., Liu, Q., and Scheuermann, R. H. (2007). A distribution free summarization method for Affymetrix GeneChip arrays. *Bioinformatics* 23, 321–327.
- Chuang, H. Y., Lee, E., Liu, Y. T., Lee, D., and Ideker, T. (2007). Network-based classification of breast cancer metastasis. *Mol. Syst. Biol.* 3, 140.
- Cole, G. M., and Frautschy, S. A. (2007). The role of insulin and neurotrophic factor signaling in brain aging and Alzheimer's Disease. *Exp. Gerontol.* 42, 10–21.
- Coulson, E. J. (2006). Does the p75 neurotrophin receptor mediate Abeta-induced toxicity in Alzheimer's disease? *J. Neurochem.* 98, 654–660.
- Dickson, B. J. (2002). Molecular mechanisms of axon guidance. *Science* 298, 1959–1964.
- Futreal, P. A., Coin, L., Marshall, M., Down, T., Hubbard, T., Wooster, R., Rahman, N., and Stratton, M. R. (2004). A census of human cancer genes. *Nat. Rev. Cancer* 4, 177–183.
- Gautier, L., Cope, L., Bolstad, B. M., Irizarry, R. A. (2004). affy—Analysis of Affymetrix GeneChip data at the probe level. *Bioinformatics* 20, 307–315.
- Groth, R. D., and Mermelstein, P. G. (2003). Brain-derived neurotrophic factor activation of NFAT (nuclear factor of activated T-cells)-dependent transcription: a role for the transcription factor NFATc4 in neurotrophin-mediated gene expression. *J. Neurosci.* 23, 8125–8134.
- Hardy, J., and Selkoe, D. J. (2002). The amyloid hypothesis of Alzheimer's disease: progress and problems on the road to therapeutics. *Science* 297, 353–356.
- Harigaya, Y., Shoji, M., Shirao, T., and Hirai, S. (1996). Disappearance of actin-binding protein, drebrin, from hippocampal synapses in Alzheimer's disease. *J. Neurosci. Res.* 43, 87–92.
- Hayashi, Y., Shi, S. H., Esteban, J. A., Piccini, A., Ponce, J. C., and Malinow, R. (2000). Driving AMPA receptors into synapses by LTP and CaMKII: requirement for GluR1 and PDZ domain interaction. *Science* 287, 2262–2267.
- Heneka, M. T., and O'Banion, M. K. (2007). Inflammatory processes in Alzheimer's disease. *J. Neuroimmunol.* 184, 69–91.
- Hoshi, M., Takashima, A., Murayama, M., Yasutake, K., Yoshida, N., Ishiguro, K., Hoshino, and T., Imahori, K. (1997). Nontoxic amyloid beta peptide 1–42 suppresses acetylcholine synthesis. Possible role in cholinergic dysfunction in Alzheimer's disease. *J. Biol. Chem.* 272, 2038–2041.
- Ideker, T., and Sharan, R. (2008). Protein networks in disease. *Genome Res.* 18, 644–652.
- Iwata, N., Tsubuki, S., Takaki, Y., Watanabe, K., Sekiguchi, M., Hosoki, E., Kawashima-Morishima, M., Lee, H. J., Hama, E., Sekine-Aizawa, Y., Saido, T. C. (2000). Identification of the major Abeta1-42-degrading catabolic pathway in brain parenchyma: suppression leads to biochemical and pathological deposition. *Nat. Med.* 6, 143–150.
- Jeffery, I. B., Higgins, D. G., and Culhane, A. C. (2006). Comparison and evaluation of methods for generating differentially expressed gene lists from microarray data. *BMC Bioinformatics* 7, 359.
- Jellinger, K. A. (2006). Challenges in neuronal apoptosis. *Curr. Alzheimer Res.* 3, 377–391.
- Jeong, H., Tombor, B., Albert, R., Oltvai, Z. N., and Barabasi, A. L. (2000). The large-scale organization of metabolic networks. *Nature* 407, 651–654.
- Korf, R. E. (1987). Depth-first iterative-deepening. *Artif. Intell.* 27, 97–109.
- Laske, C., Stransky, E., Leyhe, T., Eschweiler, G. W., Wittorf, A., Richartz, E., Bartels, M., Buchkremer, G., and Schott, K. (2006). Stage-dependent BDNF serum concentrations in Alzheimer's disease. *J. Neural Transm.* 113, 1217–1224.
- Li, H. L., Wang, H. H., Liu, S. J., Deng, Y. Q., Zhang, Y. J., Tian, Q., Wang, X. C., Chen, X. Q., Yang, Y., Zhang, J. Y., Wang, Q., Xu, H., Liao, F. F., and Wang, J. Z. (2007). Phosphorylation of tau antagonizes apoptosis by stabilizing beta-catenin, a mechanism involved in Alzheimer's neurodegeneration. *Proc. Natl. Acad. Sci. U.S.A.* 104, 3591–3596.
- Liang, W. S., Dunkley, T., Beach, T. G., Grover, A., Mastroeni, D., Walker, D. G., Caselli, R. J., Kukull, W. A., McKeel, D., Morris, J. C., Hulette, C., Schmechel, D., Alexander, G. E., Reiman, E. M., Rogers, J., and Stephan, D. A. (2007). Gene expression profiles in anatomically and functionally distinct regions of the normal aged human brain. *Physiol. Genomics* 28, 311–322.
- Luscombe, N. M., Babu, M. M., Yu, H., Snyder, M., Teichmann, S. A., Gerstein, M. (2004). Genomic analysis of regulatory network dynamics reveals large topological changes. *Nature* 431, 308–312.
- Ma, X., Lee, H., Wang, L., and Sun, F. (2007). CGI: a new approach for prioritizing genes by combining gene expression and protein-protein interaction data. *Bioinformatics* 23, 215–221.
- Ma'ayan, A., Jenkins, S. L., Neves, S., Hasseldine, A., Grace, E., Dubin-Thaler, B., Eungdamrong, N. J., Weng, G., Ram, P. T., Rice, J. J., Kershenbaum, A., Stolovitzky, G. A., Blitzer, R. D., and Iyengar, R. (2005). Formation of regulatory patterns during signal propagation in a mammalian cellular network. *Science* 309, 1078–1083.
- Maglott, D., Ostell, J., Pruitt, K. D., and Tatusova, T. (2007). Entrez Gene: gene-centered information at NCBI. *Nucleic Acids Res.* 35, D26–D31.
- Mason, O., Verwoerd, M. (2007). Graph theory and networks in Biology. *IET. Syst. Biol.* 1, 89–119.
- McCusker, S. M., Curran, M. D., Dynan, K. B., McCullagh, C. D., Urquhart, D. D., Middleton, D., Patterson, C. C., McIlroy, S. P., and Passmore, A. P. (2001). Association between polymorphism in regulatory region of gene encoding tumour necrosis factor alpha and risk of Alzheimer's disease and vascular dementia: a case-control study. *Lancet* 357, 436–439.
- Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). Network motifs: simple building blocks of complex networks. *Science* 298, 824–827.
- Nochomovitz, Y. D., and Li, H. (2006). Highly designable phenotypes and mutational buffers emerge from a systematic mapping between network topology and dynamic output. *Proc. Natl. Acad. Sci. U.S.A.* 103, 4180–4185.
- Rahman, S. A., and Schomburg, D. (2006). Observing local and global properties of metabolic pathways: 'load points' and 'choke points' in the metabolic networks. *Bioinformatics* 22, 1767–1774.
- Ravasz, E., Somera, A. L., Mongru, D. A., Oltvai, Z. N., and Barabasi, A. L. (2002). Hierarchical organization of modularity in metabolic networks. *Science* 297, 1551–1555.
- Reimers, M., and Carey, V. J. (2006). Bioconductor: an open source framework for bioinformatics and computational biology. *Methods Enzymol.* 411, 119–134.
- Serge, A., Fourgeaud, L., Hemar, A., and Choquet, D. (2003). Active surface transport of metabotropic glutamate receptors through binding to microtubules and actin flow. *J. Cell Sci.* 116, 5015–5022.
- Shen-Orr, S. S., Milo, R., Mangan, S., and Alon, U. (2002). Network motifs in the transcriptional regulation network of *Escherichia coli*. *Nat. Genet.* 31, 64–68.
- Smyth, G. K. (2004). Linear models and empirical bayes methods for assessing differential expression in microarray experiments. *Stat. Appl. Genet. Mol. Biol.* 3, 1027–1053.
- Su, J. H., Kessler, J. P., Head, E., and Cotman, C. W. (2002). Caspase-cleaved amyloid precursor protein and activated caspase-3 are co-localized in the granules of granulovacuolar degeneration in Alzheimer's disease and Down's syndrome brain. *Acta Neuropathol.* 104, 1–6.
- Tang, Y., Yamada, K., Kanou, Y., Miyazaki, T., Xiong, X., Kambe, F.,

- Murata, Y., Seo, H., and Nabeshima, T. (2000). Spatiotemporal expression of BDNF in the hippocampus induced by the continuous intracerebroventricular infusion of beta-amyloid in rats. *Brain Res. Mol. Brain Res.* 80, 188–197.
- Tuszynski, M. H., Thal, L., Pay, M., Salmon, D. P., U, H. S., Bakay, R., Patel, P., Blesch, A., Vahlsing, H. L., Ho, G., Tong, G., Potkin, S. G., Fallon, J., Hansen, L., Mufson, E. J., Kordower, J. H., Gall, C., and Conner, J. (2005). A phase I clinical trial of nerve growth factor gene therapy for Alzheimer disease. *Nat. Med.* 11, 551–555.
- Walsh, D. M., Klyubin, I., Fadeeva, J. V., Cullen, W. K., Anwyl, R., Wolfe, M. S., Rowan, M. J., and Selkoe, D. J. (2002). Naturally secreted oligomers of amyloid beta protein potently inhibit hippocampal long-term potentiation *in vivo*. *Nature* 416, 535–539.
- Willem, M., Garratt, A. N., Novak, B., Citron, M., Kaufmann, S., Rittger, A., DeStrooper, B., Saftig, P., Birchmeier, C., and Haass, C. (2006). Control of peripheral nerve myelination by the beta-secretase BACE1. *Science* 314, 664–666.
- Wuchty, S. (2001). Scale-free behavior in protein domain networks. *Mol. Biol. Evol.* 18, 1694–1702.
- Yoon, S., Choi, J., Haam, J., Choe, H., and Kim, D. (2007). Reduction of mint-1, mint-2, and APP overexpression in okadaic acid-treated neurons. *Neuroreport* 18, 1879–1883.
- Zhang, Y. W., Wang, R., Liu, Q., Zhang, H., Liao, F. F., and Xu, H. (2007). Presenilin/gamma-secretase-dependent processing of beta-amyloid precursor protein regulates EGF receptor expression. *Proc. Natl. Acad. Sci. U.S.A.* 104, 10613–10618.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 14 September 2008; paper pending published: 30 September 2008; accepted: 30 April 2009; published online: 29 May 2009.
- Citation: Yanashima R, Kitagawa N, Matsubara Y, Weatheritt R, Oka K, Kikuchi S, Tomita M and Ishizaki S (2009) Network features and pathway analyses of a signal transduction cascade. *Front. Neuroinform.* (2009) 3:13. doi:10.3389/neuro.11.013.2009
- Copyright © 2009 Yanashima, Kitagawa, Matsubara, Weatheritt, Oka, Kikuchi, Tomita and Ishizaki. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Brainlab: a Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the NeoCortical Simulator

Rich Drewes^{1,2*}, Quan Zou¹ and Philip H. Goodman^{1,3}

¹ Brain Computation Laboratory, University of Nevada, Reno, USA

² Program in Biomedical Engineering, University of Nevada, Reno, USA

³ Department of Medicine and Program in Biomedical Engineering, University of Nevada, Reno, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Marc-Oliver Gewaltig, Honda Research
Institute Europe GmbH, Germany
Tim Masquelier, Centre de Recherche
Cerveau & Cognition, France

*Correspondence:

Rich Drewes, Program in Biomedical
Engineering, University of Nevada,
Reno, USA.

e-mail: drewes@interstices.com

Neuroscience modeling experiments often involve multiple complex neural network and cell model variants, complex input stimuli and input protocols, followed by complex data analysis. Coordinating all this complexity becomes a central difficulty for the experimenter. The Python programming language, along with its extensive library packages, has emerged as a leading “glue” tool for managing all sorts of complex programmatic tasks. This paper describes a toolkit called Brainlab, written in Python, that leverages Python’s strengths for the task of managing the general complexity of neuroscience modeling experiments. Brainlab was also designed to overcome the major difficulties of working with the NCS (NeoCortical Simulator) environment in particular. Brainlab is an integrated model-building, experimentation, and data analysis environment for the powerful parallel spiking neural network simulator system NCS.

Keywords: python, toolkit, neuron, spiking neural network, simulator

INTRODUCTION

Spiking neural network simulator software systems continue to grow in speed and capacity (see Brette et al., 2007 for a recent survey). The complexity and size of the models simulated on these systems also continue to grow, threatening to overwhelm the ability of the experimenter to build the models, conduct parameterized experiments, and analyze the huge amounts of resulting data. The simulators themselves are generally extremely efficient but minimalist tools written in low-level programming languages that are difficult to understand and modify by any but a few dedicated experts. Tools beyond the simulators themselves are needed to help the experimenter cope with the complexity of the experiments.

In our work with one such powerful spiking neural network simulator called NCS¹ (the NeoCortical Simulator, described briefly in the Section “NCS”) we encountered these general complexity barriers. Our work was also hampered by problems specific to working with NCS, most notably the necessity of preparing network models for simulation using NCS’s restrictive neural modeling interface, the .in file format. We confronted all these problems together by creating a unified Python toolkit called Brainlab², which has greatly eased the burden of organizing and conducting our experiments in general, and working with NCS in particular.

The fundamental proposition of Brainlab is this: For the tasks of complex neuroscience model-building, experimentation, and analysis, nothing short of a full-fledged programming language will suffice. No neural model file format or restricted special purpose programming language for modeling will ultimately suffice for day to day work. And as long as a real programming language will be needed to hold the whole experimental enterprise together,

it might as well be a modern mature programming language with a large scientific user community, rather than a custom-built, special purpose language. In Brainlab we selected the Python language for this purpose, and the rationale for our decision is given in the Section “Why Python?”

Brainlab has been in use since 2003, with publications in 2005 (Drewes, 2005a,b). In the intervening time, validation for the decisions we made in the design of Brainlab seems to have come from several areas. Scientific support for Python, in the form of libraries and the user community, has continued to grow and mature. Other projects have independently started that also use Python as a front-end modeling and back-end analysis tool for various other neural simulators. The NEST simulator³ system now offers a Python interface called PyNEST⁴. The NEURON⁵ simulator has added Python as an alternative interpreter to Hoc. PyGENESIS is now available for the GENESIS⁶ simulator. The PyNN⁷ system, part of the broader Neuralensemble initiative⁸, goes a step further and offers a common Python interface to NEURON, NEST, and PCSIM⁹ (but not NCS).

The Brian¹⁰ project differs from the systems mentioned so far, and also NCS, in that Brian is a self-contained Python neural simulation solution, rather than a front-end to a simulation engine written in a different programming environment. Brian still achieves

¹<http://brain.cse.unr.edu/>

²<http://brainlab.sourceforge.net/>

³http://www.nest-initiative.org/index.php/Main_Page

⁴<http://www.nest-initiative.org/index.php/PyNEST>

⁵<http://www.neuron.yale.edu/neuron/>

⁶<http://www.genesis-sim.org/GENESIS/>

⁷<http://neuralensemble.org/trac/PyNN/>

⁸<http://www.neuralensemble.org/>

⁹<http://www.lsm.tugraz.at/pcsim/>

¹⁰<http://brian.di.ens.fr/>

good single-processor simulation performance through the use of vectorized processing provided by the NumPy library, and it can also manage multiple jobs in parallel on a cluster computer system, but splitting a single large simulation onto multiple compute nodes is not supported. The Topographica¹¹ project provides standalone Python tools intended for exploring higher-level neural abstractions like sheets and projections from neural area to area. Though not primarily intended for investigations that require detailed simulation of individual neurons, Topographica can be interfaced to lower-level simulators like NEURON and GENESIS. Topographica is one of the older Python neuroscience tool packages, with an initial public release in late 2005.

Perhaps because NCS has a fraction of the number of users of some other simulators (e.g. NEURON and GENESIS), Brainlab has attracted comparatively little attention. Brainlab merited brief mention in a recent survey of major spiking neural net simulator packages (Brette et al., 2007). Brainlab was unnoticed by another recent survey of interoperability of neuroscience software (Cannon et al., 2007) though Python interfaces to other spiking neural network simulators (e.g. NEURON's and NEST's) were described there in some detail.

BRAINLAB MOTIVATION, DESIGN, AND IMPLEMENTATION

In this section, we will first describe enough about NCS so that a reader will understand the problems we faced designing a system to interface to and control it. Next we will describe the broad features we wanted to include in our toolkit, and how we wanted the finished system to appear to the user for modeling, simulation, and analysis. Then we will describe in detail how we actually confronted the problems interfacing to NCS, to implement the Brainlab system.

NCS

The development history of NCS is recounted elsewhere (Drewes, 2005b). In its current evolution, NCS is a parallel (MPI-based) spiking neural network simulator written in C/C++ that can perform very large discrete-time simulations with a reasonably high degree of biological realism. Simulations with a million neurons and a billion synapses have been accomplished. NCS allows for neuron models that include detailed and customizable ion channel and cell membrane voltage dynamics, but for efficiency the stereotypical action potential voltage and postsynaptic conductivity waveforms are templated rather than generated dynamically. NCS supports multi-compartment cells but often large scale simulations are done using single compartment models. A good recent comparison of NCS with other spiking neural network simulators, including some discussion of maximum simulation sizes, is Brette et al. (2007).

THE NCS INPUT FILE (THE .in FILE)

NCS reads a description of a neural network model and other simulation parameters from a plain text file whose filename is supplied to NCS as a command line argument. For our purposes here it is not necessary to go into great detail about the format of this file, but we do wish to describe it generally in order to explain some of the shortcomings of working with it.

This input file, hereafter called a .in file after the convention of using .in as a filename extension for such files, contains a variable number of subsections. Each subsection starts with a line that contains the name of the subsection (which must be one of a limited number of keywords permitted by the system) and ends with a line that contains END_ with the section name appended. The first subsection in a .in file is the BRAIN section. In the BRAIN section of the file are defined global features that affect the entire simulation. For example, a line beginning with JOB defines a job name for the simulation. Some subsections can be repeated (for example, a COLUMN or LAYER), and then each is assigned a unique text identifier within the file. The file format allows other portions of the file to reference these named objects, to create additional instances of them, but no structural or other significant variation in a defined object is permitted. The .in format definition permits no looping constructs or macro substitutions. Other sections of the .in file define connections between these objects, with references to the text names of the objects being connected. Because of these restrictions, NCS .in files tend to be quite long even for fairly simple networks, and they tend to be prone to syntactical error or internal referential inconsistency when edited manually.

Other neural simulator systems acquired programming languages (e.g. Hoc for NEURON) to avoid the limitations of a flat input file format like NCS's. NCS never went this far, though there were several attempts to elaborate the .in file with macros, loops and other features. None of these efforts for NCS were widely used or reached the generality of a true programming language. Many NCS users eventually created custom text processing programs in other programming languages (like MATLAB) that would emit .in files. But writing special-purpose macro processors to create .in files is time consuming work that generally cannot be reused on later projects, and MATLAB is not a particularly good text processing tool. The experimentation process was either not automated or automated with external custom scripts, making the whole process cumbersome and systematic model parameter search difficult. Data file management was typically done manually using ftp type tools.

One other unusual aspect of NCS deserves mention: it imposes a notion of the cortical column and the cortical layer as structural elements, and this requirement is reflected in the structure of the NCS input file. Even if an NCS user wishes to simply simulate two connected cells, or a homogeneous collection of cells for a study of, say, synfire chains, he must define those cells within an NCS LAYER text block, and that in turn within an NCS COLUMN text block. This introduces additional complication for the simplest simulations.

NCS USAGE

NCS is optimized for large cluster computer systems (Beowulf clusters). A common usage pattern is as follows: A user typically first prepares an input file in the .in file format in a text editor, specifying the neuron, synapse, channel, and network model. This file is copied across a network to the cluster computer and NCS is invoked there with the file as a command line argument. Reports are written to the cluster computer's disks during the simulation run, which can last from a few seconds to days. Data analysis is then performed on the cluster computer if the data set is very large, or the data is copied back to the user's workstation for data analysis.

¹¹<http://topographica.org/Home/index.html>

if that is feasible. The experimenter then makes some adjustments to the model and tries again.

BRAINLAB MOTIVATION AND DESIGN GOALS

Faced with the powerful but difficult to use NCS simulator, we set about to design a toolkit that would offer the following:

1. An interactive shell for simple experimentation with NCS, making NCS a more suitable educational tool for learning the behavior of spiking neural networks and also a more convenient platform for experienced users to explore the behavior of new cell or network elements.
2. A convenient platform for parameterized control of sets of NCS experiments.
3. A convenient platform for scripted regression testing of NCS itself, with flexible output validation.
4. Scripted, algorithmic generation of neural network models rather than NCS's native static file specification of networks.
5. Convenient, integrated, graphical on-line reporting and plotting of spiking, current, and voltage activity of cells, synapses, and channels.
6. Convenient, integrated, on-line three-dimensional plotting of neural network architecture for expository and diagnostic purposes.
7. Experimental support for higher-level abstractions than those provided natively in NCS (for example support for *areas*, composed of arrays of columns, and a variety of distinct area-to-area synaptic connection patterns), and a flexible environment to add new ones.
8. Support for lower-level abstractions too unwieldy to reasonably manage in native NCS (for example, columns where all cells are enumerated and independently, rather than just statistically, addressable).
9. A container for a standard and extensible library of NCS network building blocks (for example channels, cell types, columns, spike templates), where all components are guaranteed to interoperate, utilize consistent naming conventions, and may be manipulated programmatically as variable objects rather than text chunks.
10. A more convenient, higher-level, object-oriented representation of neural networks that hides many complexities and inconveniences inherent in NCS's native `.in` file format.
11. A convenient environment in which to convert a neural network description into a chromosomal representation suitable for use with a genetic algorithm.
12. A convenient environment in which to access NCS's realtime stimulus input capabilities, especially for robotic interface applications (see Goodman et al., 2008 for more information on using NCS in robotics).
13. The ability to conveniently extend many of these capabilities without recourse to coding in NCS's native compiled programming environment (the C/C++ language).

WHY PYTHON?

When we selected Python as the language for Brainlab, Python was not yet in wide use in neuroscience, and it was also in the midst of a seemingly endless reorganization of its vector processing math

support libraries. Nevertheless, there were hopeful signs of building momentum for Python as a scientific platform, and the base language was so appealing in several respects that we selected Python as the language for our project.

Python is an open source, cross platform programming language. The base Python language is constantly being extended and made more powerful by hundreds of developers working together across the world. In addition to the base language, there are dozens of external packages in various states of development, from polished to prototype. These packages gradually move into the base distribution as they mature and if they are of sufficiently wide interest.

Python is ordinarily compiled into bytecode automatically and the bytecode is then interpreted in a runtime virtual machine. This is essentially the same approach used by Java, though the compilation generally requires an explicit step with Java. Compilation to bytecode results in code execution that is generally faster than ordinary interpreted code. Python is dynamically typed, making programming extremely convenient. Built in datastructures like lists, dictionaries (hashes), and arrays help make Python programs very concise. The clean syntax makes programs easy to understand. Python has a well deserved reputation as an extremely clean and easy to read and understand language.

At the time we selected it, Python already had a growing set of support library packages for scientific computation. These have since matured. Some of these packages are used in Brainlab, including:

- Matplotlib¹², a MATLAB-like plotting package
- PyOpenGL¹³, OpenGL bindings for Python
- NumPy¹⁴, MATLAB-style array processing
- SciPy¹⁵, a set of scientific tools for Python, including pseudo random number generators and transforms

BRAINLAB TO NCS INTERFACE FOR NETWORK MODELING

When we were designing the Python to NCS interface for the first version of Brainlab, there were already a number of ways to interface Python to a C/C++ application. Of these, one approach we considered seriously was to create a Python module out of the NCS C/C++ program with fairly simple and standardized wrapper code using standard techniques¹⁶. The wrapped C code could then be included into a Python program with the `import` command. With this approach the Python program would be in charge from the beginning, and it could selectively make normal looking Python function calls into the wrapped C code to actually perform the NCS simulation and other functions. How would the network, cell, synapse, and other neural network parameters be communicated to NCS? A reasonable approach would be to define a new abstract network modeling interface using high-level Python facilities, perhaps a Python Object class for a Cell, a Synapse, and so on, that allows these objects to be created and

¹²<http://matplotlib.sourceforge.net/>

¹³<http://pyopengl.sourceforge.net/>

¹⁴<http://numpy.scipy.org/>

¹⁵<http://www.scipy.org/>

¹⁶<http://docs.python.org/extending/>

interconnected. This Python-based model could then be converted directly to the internal in-memory representation of network models of NCS, called the `GCList`, through a new function provided by the imported NCS python module. This function, being in the C/C++ side of things, would have full access to the memory structures, memory allocation, and cluster-distribution routines that NCS itself uses to convert the `.in` file representation into the `GCList` representation for simulation, merely bypassing the file parsing NCS normally uses to build its internal network representation.

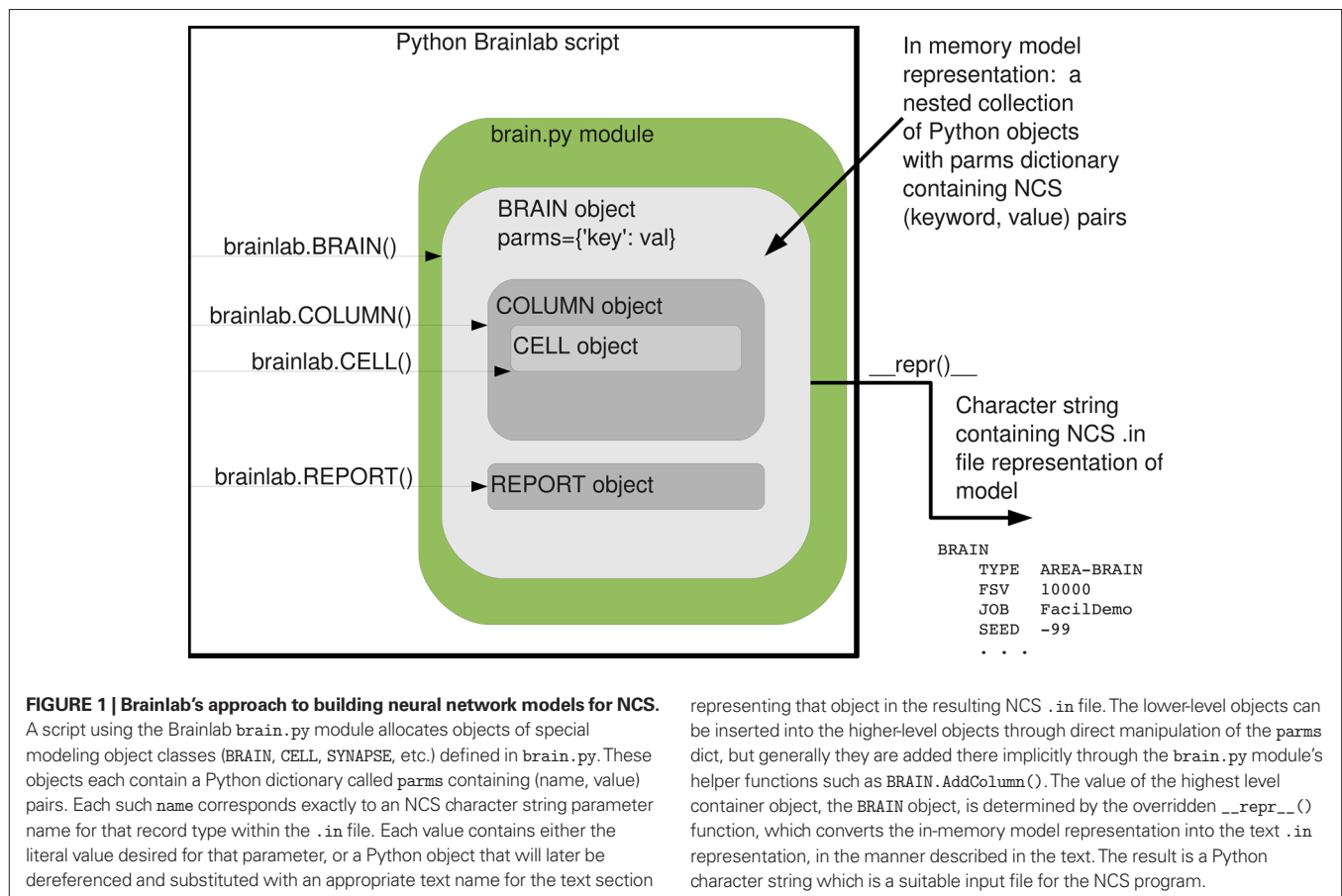
However this tightly integrated approach would have a number of disadvantages. Such a Brainlab system would have to be at least recompiled with every new release of NCS. But there would be more complications than just that. While the NCS `.in` file representation is part of the NCS documentation and is fairly stable, the internal `GCList` representation does not have a publicly documented interface. The `GCList` interface changes over time, and when it changes, corresponding detailed C/C++ changes would then have to be made in the NCS/Python module for import. A possibly larger documentation burden also would be placed on Brainlab to describe the new model-building interface.

We opted instead to try to achieve our design goals with a much looser Brainlab-NCS interface for modeling and simulation. We left NCS as a completely separate programming project and did not even try to integrate more tightly with it than its existing published modeling (`.in` file) and invocation (command line) interface. So

Brainlab would have to provide a convenient and powerful Pythonic network modeling interface to the user, since that was a primary design goal, but it would also have to emit a properly formatted `.in` file for use by NCS on the back-end. The approach we took to model-building in Brainlab is depicted in **Figure 1**.

The `BRAIN`, `CELL`, `LAYER`, and other sections of the NCS `.in` file are each implemented in Brainlab as a Python object class. The `__repr__()` method for each object is overridden so that printing an object results in text for that object in a format suitable for inclusion in the NCS `.in` file. In the case of a lower-level object, this method just prints out the object itself, but does not print any other objects that are referenced by the object being printed. The `BRAIN` object's `__repr__()` method, however, first recursively traverses the entire tree of objects referenced from the `BRAIN` object and a list is composed for each type of referenced object. Once all referenced objects have been collected together, the entire NCS `.in` file is printed, starting with the `BRAIN` section, and proceeding to all of the other sections of the `.in` file in the conventional order.

The lower-level classes are implemented as nested classes within the `BRAIN` class. Note that they are not derived subclasses, but rather nested classes. Derived subclasses are appropriate where the subclass has most of the aspects of the superclass but some additional features. In Brainlab the nested classes are not logically subclasses of the `BRAIN` since they do not share the same characteristics as the super-object but are merely contained by it.



However, the lower-level classes do need access to the component type libraries that are stored with the BRAIN class. If the lower-level objects were entirely separate classes, they would not have convenient access to the component type libraries. By making the lower-level classes nested within the BRAIN class, they do have that access.

We chose to map many of the modeling details of Brainlab directly onto the underlying NCS implementation, rather than providing a completely new modeling interface. This primarily means that we preserve NCS's text character string names for various neural parameters of the cells, synapses, channels, and so on. This eases the documentation burden on Brainlab since we can refer directly to the NCS's documentation on many points. Furthermore, it makes keeping Brainlab up to date with respect to NCS very easy. Whenever NCS adds support for a new parameter within an existing modeling object, it is usually a simple matter to add it to the permitted parameter list of the appropriate class in `brain.py` and that is the end of it. (When NCS adds entirely new types of objects, as is occasionally done, there is a bit more work, but even still it is usually just a matter of intelligently cloning an existing object to a new name and making a few changes.) The overall mechanism of `.in` file emission through recursive application of `__repr__()`'s to discovered objects starting at the top-level BRAIN extends quite easily.

The simple strategies of creating a Python object class for each `.in` file section, with automatic conversion from object to text through the `__repr__()` method, combined with the ability to reference one object from another, achieved all our design goals for a Pythonic modeling interface to NCS. The modeling power achieved by combining these few concepts in this way should not be underestimated.

BRAINLAB TO NCS INTERFACE FOR SIMULATION

Once the internal Pythonic neural network model is constructed inside the top-level BRAIN object, it can be simulated by invoking the BRAIN's `Run()` method. Since we elected to keep an arms-length interface between Brainlab and NCS, the invocation of NCS is done through the use of a `popen()` call, as follows. First, Brainlab determines through invocation options or a standard configuration `.rc` file whether the NCS process is to be invoked locally, or on a remote compute server (typically a cluster). The `.in` file generated from the `__repr__()` method of the top-level BRAIN object is stored in a disk file locally, then propagated to the remote compute server using `ssh`¹⁷ (secure shell) if necessary. Other support files, such as input stimulus patterns, are likewise generated and propagated as needed. Next, the NCS invocation command is constructed, again with appropriate references to remote servers with `ssh`, and then this command is executed using `popen()`. Brainlab monitors the realtime progress of the command as NCS reports the progress of the run through the file descriptors of the `popen()`. If an error condition is detected in the output, Brainlab either throws a Python exception, or an error code to the caller. When Brainlab detects that a run has completed, it constructs additional commands to retrieve output files from the remote compute server, as needed.

We felt it was essential to support all three stages of operation – model-building, simulation, and analysis – completely within the control of the Python Brainlab environment. This permits self-contained and reproducible experiments, in the form of Python Brainlab scripts. This also opens up the possibility of parameterized model search with feedback from model performance affecting parameters of the next iteration, or even the use of genetic programming techniques for parameter search, all within a Brainlab script.

BRAINLAB'S MODULE ORGANIZATION

Brainlab itself is implemented as two main Python modules, `brainlab.py` and `brain.py`. The `brain.py` module contains the parts of the system concerned with building a neural model using Python classes supplied by the module and other normal Python facilities, and then automatically converting this model to a format understandable to NCS (a `.in` file). The `brainlab.py` module contains support functions for invoking an NCS simulation on a model either locally or remotely on a remote cluster, and analyzing and documenting the results using plotting and other functions.

In addition to these two main modules, an optional module called `netplot` is available. This module can take a model built using the core BRAIN class of `brain.py` and convert it into a three-dimensional depiction using the model's architecture and hints provided during model construction. The three-dimensional depiction can be examined and explored interactively on a workstation or saved in a number of graphics file formats. The PyOpenGL¹⁸ package is used for the actual rendering.

BRAINLAB USAGE

BUILDING MODELS WITH BRAINLAB

In Brainlab, every brain model is an instance of a new Python object class called BRAIN. Once the `brainlab` library itself is brought into a Python program with the `import` command, creating a brain object is by the usual Python means:

```
import brainlab
b = brainlab.BRAIN()
```

The variable `b` then refers to the newly created, and initially empty, brain model. When a BRAIN object is created, it contains a default set of commonly used types of neural network modeling components. (There are initially no *instances* of these types in the brain model.) These component types can be directly instantiated and then used for construction of network models, or they can be modified in place and then used in a model, or they can be copied to new types with different names and then the copies can be modified and instantiated for use in a model. The component types are contained in Python dictionaries (hashes), and the keys of the dictionary are simply the text names of the components. These building blocks are automatically included within a Python dictionary called `libs` in each BRAIN instance. There can be multiple libraries of parts within a BRAIN. The library provided with the class is given the key name `standard`, and is itself a dictionary. In this dictionary are subdictionaries for the different types of

¹⁷<http://www.openssh.org/>

¹⁸<http://pyopengl.sourceforge.net/>

neural modeling components, such as channels (accessed with the `chanotypes` dictionary key), cell types (accessed with the `celltypes` key), synapse facilitation and depression profiles (under the `sfsds` key), and more as listed below.

The following interactive Python session shows how to view these different library components and shows how one could modify the negative Hebbian learning window duration parameter within the standard Hebbian learning profile:

```
>>> b.libs['standard'].keys()
['comptypes', 'spks', 'chanotypes', 'spsgs', 'cols',
 'celltypes', 'sls', \ 'syntypes', 'lays', 'sfsds']
>>> blib=b.libs['standard']
>>> blib['sls'].keys()
['OHebb', '-Hebb', 'BHebb', '+Hebb']
>>> blib['sls']['BHebb']
SYN_LEARNING
  TYPE                      BHebb
  LEARNING                  BOTH
  NEG_HEB_WINDOW            0.04000 0.00000
  NEG_HEB_PEAK_DELTA_USE    0.01000 0.00000
  NEG_HEB_PEAK_TIME         0.01000 0.00000
  POS_HEB_WINDOW            0.04000 0.00000
  POS_HEB_PEAK_DELTA_USE    0.00500 0.00000
  POS_HEB_PEAK_TIME         0.01000 0.00000
END_SYN_LEARNING

>>> blib['sls']['BHebb'].parms['NEG_HEB_WINDOW']=(.05,.01)
>>> blib['sls']['BHebb']
SYN_LEARNING
  TYPE                      BHebb
  LEARNING                  BOTH
  NEG_HEB_WINDOW            0.05000 0.01000
  NEG_HEB_PEAK_DELTA_USE    0.01000 0.00000
  NEG_HEB_PEAK_TIME         0.01000 0.00000
  POS_HEB_WINDOW            0.04000 0.00000
  POS_HEB_PEAK_DELTA_USE    0.00500 0.00000
  POS_HEB_PEAK_TIME         0.01000 0.00000
END_SYN_LEARNING
```

The Section “Usage Example: RAIN Network” contains another example of creating components based on the included standard library.

An NCS `.in` file contains a number of text blocks, with each block consisting of a number of parameter keywords on the left and their values to the right. The values can be of several types. In the example above, the numbers for the `NEG_HEB_WINDOW` are a mean and standard deviation. During model initialization, NCS assigns that parameter to a random value from a normal distribution with the mean and standard deviation requested. For other parameters, such as the `RSE_INIT` parameter of the synapse object, two numeric values specify a minimum and a maximum of a range. In the case of the `LEARNING` parameter in the example above, the value for a parameter is a text label that references another block defined within the file. The NCS documentation details each parameter and its expected values. In some cases, Brainlab allows commonly used and frequently modified parameter values to be changed in Brainlab function calls. For example, when specifying a synaptic connection, the probability of the connection and the conductance speed values can be set directly using the `prob=` and `speed=` keyword

arguments to the Brainlab `AddConnect()` method. In all cases however, NCS parameters can be set by modifying a dictionary value in the appropriate `parms` dictionary of the object with the key set to the text name of the NCS parameter name. This approach gives convenience to the programmer while allowing quick access to new NCS parameters as they are added to the system, by simply adding a keyword to a list in the Python class definition for that object.

In NCS, cells cannot exist on their own but rather only as part of a higher-level structure called a column. A column is composed of one or more layers, which in turn is composed of one or more groups of cells. Brainlab has `COLUMN`, `LAYER`, and `CELL` objects that correspond to these structures. A Brainlab script can build a column up from cell groups and layers, or instead use a convenience function that will add a pre-built column in a single step. The following Brainlab function adds to the model an instance of an ordinary column populated with a single cell:

```
newcol = b.Standard1CellColumn()
```

Additional optional parameters to the function can specify a cell type to use (other than the default), spatial coordinates for the cell, and more.

At this point the Brainlab script typically makes connections between the cells or cell groups. Brainlab functions such as `AddConnect()` are used for this. The Python variables for the objects are used as the point of contact for connection. An example of this is given in the Sections “Usage Example: Hebbian Learning” and “Usage Example: RAIN Network”. Report requests are also added to the brain at this time.

SIMULATING MODELS WITH BRAINLAB

Once the `BRAIN` object is created, simply printing it with the Python `print` command causes Brainlab to emit a complete, properly formatted `.in` file containing all the information added to the brain by the modeler. If desired, this file can be examined and manually submitted for simulation by NCS. This approach is occasionally useful for debugging purposes, but in practice it is seldom necessary to view the generated `.in` file directly. Instead, the modeler can simply leave the underlying `.in` file mechanism hidden and evoke an NCS simulation directly on the model using the `brainlab.Run()` function on the brain:

```
brainlab.Run(b, nprocs = 32)
```

In this example the simulation is evoked remotely on 32 processors. The `.in` file that results from the model is created by Brainlab behind the scenes, copied over to the compute cluster automatically by Brainlab, and the simulation results are fetched on demand as the data analysis portion of the Brainlab program requires them.

Brainlab is designed primarily to run on the user's workstation, and send jobs across a network to be simulated on a different computer (or cluster). There are several reasons for this focus. The user has more control over the software installed on a personal workstation than on a typical group or departmental compute server or Beowulf cluster, where it may be more difficult to get installed the libraries necessary to run Brainlab. Often data will be analyzed repeatedly, displayed and analyzed in a variety of ways,

and that is best done on a personal workstation so that specialized tools are guaranteed to be available and also so that other users of the simulation environment will not be affected. Also typically a personal workstation will have high-performance display hardware that will work more efficiently with extensive graphing, perhaps in three dimensions.

Brainlab can also be configured to run directly on the machine where NCS also does the simulation. With modern high-performance multi-core CPUs this is a good option for smaller exploratory simulations.

The encapsulation of the model construction, simulation, and data analysis loop within a single program, a Python Brainlab script, makes automatic model parameter search easier. In some of our work we have defined a mapping from artificial chromosome to neural network model, and used a standard Python genetic algorithm package to do a fitness search for the best functioning model (Drewes et al., 2004).

DATA ACCESS, ANALYSIS, AND PLOTTING WITH BRAINLAB

Brainlab provides a few convenience functions for loading, processing, and plotting standard NCS reports. In combination with the SciPy and Matplotlib packages, modelers can do sophisticated mathematical analyses and create complex graphics for view or publication. Efficient access to very large datasets is available to the modeler through Python's hdf5 interface, pytables. With the PyOpenGL libraries, Brainlab provides some limited three-dimensional plotting tools for viewing network models.

We will mention a few of the more commonly used Brainlab data access and plotting routines here. The Brainlab `LoadReport()` function returns a NumPy array containing all the data captured from a requested NCS report. The data to be loaded can be limited by time range or by range of cells. The returned data can then be processed further in the Brainlab program using the wide range of Python or NumPy tools. The Brainlab function `LoadSpikeData()` returns a list of just the spike times for a given range of cells for a given time. The `ReportPlot()` function gives a simple visual representation of continuous NCS report data (often voltages or currents) on screen or into a graphical file. Brainlab makes extensive use of the Matplotlib library for the actual generation of the plots.

Brainlab handles remotely invoking a simulation on a compute cluster, and it also simplifies accessing the resulting NCS report files. The same Brainlab `LoadReport()` function works whether the file data was captured remotely or on the local workstation. Brainlab also tries to use knowledge about the simulation environment to be efficient about management of report files. For example, rather than copying large report files across a network from the compute cluster to the workstation for processing, Brainlab can in some cases invoke itself remotely on the compute cluster for report processing, and then only copy back the much smaller amount of data that is the result of the processing. The programmer generally does not need to be aware, for either simulation or analysis, that the computation was done remotely.

Figure 2 is a sample compound plot, generated using Brainlab convenience functions and the Matplotlib library, from the Hebbian learning simulation detailed in the Section "Usage Example: Hebbian Learning". Refer to Drewes (2005b) for further 2D and 3D Brainlab plot examples.

USAGE EXAMPLE: HEBBIAN LEARNING

Following is a complete, functional example of Brainlab usage. The results of this Brainlab example are shown graphically in **Figure 2**, and referring to the plot while reviewing the explanation below will help to make the example clear. (Note however that to reduce space the code below draws only one of the subgraphs shown in **Figure 2**.) This simple example demonstrates positive Hebbian learning: when spikes are initially applied to cell A between time 0 s and 0.5 s, the target cell T spikes because the synaptic connection from A to T is initialized to a strong value. However the initial spikes forced onto B by external stimulus (during time 0.5 s to 1.0 s) do not result in the target cell T spiking, because the B to T synapse is initially weak. During time 1.5 s to 2.5 s, a series of three spikes are forced by external stimulus onto both cell A and B. The spike forced on cell A is sufficient to evoke an output spike on T, as we have already seen. The forced spike on B just before the evoked spike on T causes the B-to-T synapse to strengthen through positive Hebbian learning. In the final phase, from time 3.0 s to 3.5 s, we see that after the synaptic strengthening, forced spikes on B are now alone enough to evoke a spike on T. Here is the script:

```
import brainlab
import pylab
```

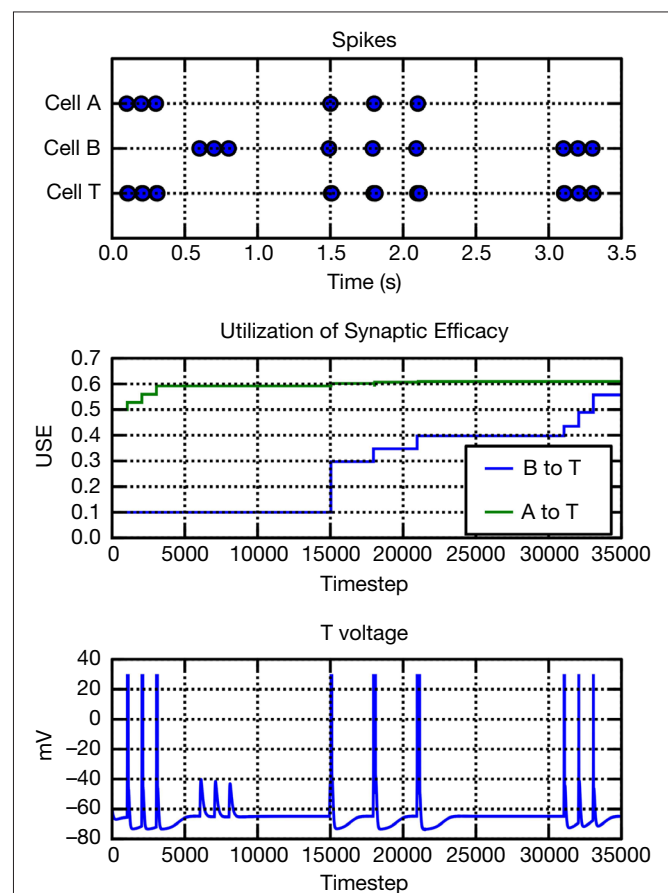


FIGURE 2 | Output of Hebbian learning example from the Section "Usage Example: Hebbian Learning".

```

brainname="HebbTest"    # output files begin with this name
endsim=3.5              # seconds to simulate
FSV=10000              # simulation timesteps per second
timesteps=FSV*endsim

# set up times (in secs) for two spike inputs, a and b:
eps=.010                # a small epsilon time offset
ain=[.1,.2,.3, 1.5, 1.8, 2.1]
bin=[.6,.7,.8, 1.5-eps, 1.8-eps, 2.1-eps, 3.1, 3.2, 3.3]

# create the brain object container:
newb=brainlab.BRAIN(simsecs=endsim, jobname=brainname,
                    fsv = FSV)

# create three cells in the brain:
A=newb.Standard1CellColumn("A")
B=newb.Standard1CellColumn("B")
T=newb.Standard1CellColumn("T")

# customize a standard synapse profile:
cs=newb.syntypes["C.strong"]
# BHebb references a standard synapse learning profile with
# both + and - Hebbian. Select that for our synapse, then
# modify:
cs.parms["LEARN_LABEL"]=newb.sls["BHebb"]
cs.parms["MAX_CONDUCT"]=0.10
cs.parms["ABSOLUTE_USE"]=(0.5, 0.0)    # initial synaptic
                                       efficacy parameter

# make a copy of this synapse to new name, then reduce
# initial strength:
cw=newb.Copy(newb.syntypes, "C.strong", "C.weak")
cw.parms["ABSOLUTE_USE"]=(0.1, 0.0)

# modify a Hebbian learning parameter in standard library:
hp=newb.sls["BHebb"]
hp.parms["POS_HEB_PEAK_DELTA_USE"]=(.20, 0)

newb.AddConnect(B, T, cw, prob=1.0, speed=10.0)
newb.AddConnect(A, T, cs, prob=1.0, speed=10.0)

d=(0.0, endsim)
# tell NCS to report on some voltage values:
newb.AddSimpleReport("AReport", A, reptime="v", dur=d)
newb.AddSimpleReport("BReport", B, reptime="v", dur=d)
newb.AddSimpleReport("TReport", T, reptime="v", dur=d)

# tell NCS to report on some absolute USE (synaptic efficacy)
# values:
newb.AddSimpleReport("BtoTUSE", T, reptime="a",
                    dur=d, synname=cw)
newb.AddSimpleReport("AtoTUSE", T, reptime="a",
                    dur=d, synname=cs)

# tell NCS to apply our spike inputs to A and B:
newb.AddSpikeTrainPulseStim("Astim", A, ain)
newb.AddSpikeTrainPulseStim("Bstim", B, bin)

# start the simulation:
brainlab.Run(newb, verbose=True, nprocs=1)

# load resulting NCS reports into Python variables:
adata=brainlab.LoadSpikeData(brainname, "AReport")
bdata=brainlab.LoadSpikeData(brainname, "BReport")

```

```

tdata=brainlab.LoadSpikeData(brainname, "TReport")

# create a simple plot using Brainlab's interface to
# matplotlib/pylab:
brainlab.ReportPlot(brainname, "BtoTUSE", plottitle="B
                    synapse on T", xlabel="Timestep",
                    ylab="USE", linelab=["B to T"])
pylab.show()        # display the plot

```

USAGE EXAMPLE: RAIN NETWORK

In this section, we give an example of how Brainlab is used to create a type of model that our lab has called RAIN (Recurrent Asynchronous Irregular Network). This type of asynchronous, irregularly firing network with persistent activity is similar to the models investigated by Vogels and Abbott (2005) and it is also a benchmark model used in the Brette et al. (2007) review of neural simulator systems. Our network has 4000 leaky integrate-and-fire neurons, 80% excitatory and 20% inhibitory. Each neuron is defined as a single compartment model with a time constant, $\tau = 20 \mu\text{s}$, $\bar{g}_{\text{leak}} = 5 \text{ ns}$, and $E_{\text{leak}} = -60 \text{ mV}$. The neuron will generate an action potential and the membrane potential will reset to the clamped resting potential for 5 ms whenever the membrane potential crosses the threshold at -50 mV . The excitatory neurons differ from the inhibitory ones with a depolarization-activated, noninactivating potassium channel (I_m current), which is responsible for the adaptation of firing rate of cortical pyramidal cells (Yamada et al., 1998).

Both excitatory and inhibitory type synapses are simulated as conductance changes with instantaneous jump at maximal value and exponential decays, i.e., a presynaptic event generates a synaptic conductance change of \bar{g} , which decays according to the following equation:

$$g(t) = \bar{g} \times e^{-t/\tau}$$

The synaptic time constants are 5 and 10 ms, and quantal conductances are 5 and 50 nS for excitatory and inhibitory synapses, respectively. All synapses are created with synaptic delay chosen from a normal distribution with a mean of 1 ms and standard deviation of 1 ms.

Neurons were randomly connected by a probability of 2% by conductance-based synapses (Gupta et al., 2000). For out-bound inhibitory connections, we incorporate the diversity of GABAergic interneurons. The experiment performed by Gupta et al. (2000) indicates that GABAergic synapses in neocortical layers II to IV have three statistically distinct types of synapses, where each type has particular temporal dynamics of synaptic transmission. The synapses were modeled according to the concepts of the refractoriness of the release process (Markram et al., 1998) as shown in **Table 1**. The Brainlab code below demonstrates the creation of a new synaptic facilitation and depression profile called `sfd_1` by copying a standard Brainlab library profile called `F1`. Once copied, the new profile is modified according to data in **Table 1**.

```

# Create SYN_FACIL_DEPRESS based on 'F1' from sfd library
sfd_1 = b.Copy(b.sfd, 'F1', 'sfd_1')
sfd_1.parms['SFD'] = 'BOTH'
sfd_1.parms['DEPR_TAU'] = (0.376, 0.253)
sfd_1.parms['FACIL_TAU'] = (0.045, 0.21)

```


Table 1 | Dynamic parameters of GABAergic synapses (Gupta et al., 2000).

	F1	F2	F3
INH to EXC (%)	7.6	76.3	16
INH to INH (%)	29.2	58.3	12.5
τ_{facil} (ms)	376	21	62
τ_{depr} (ms)	45	706	144
\bar{g} (nS)	3.24	7.76	3.44

Next we create a new inhibitory synapse profile called `InhSyn1` that is based on the Brainlab standard profile called `I`. The facilitation and depression profile just created is then embedded into the new synapse type. Note that some extraneous parameters inherited from the default profile are also deleted at this time, and note that the reference to the facilitation and depression profile is made to the newly-created variable, rather than the text string name of the profile (though Brainlab supports either, the former is generally easier and less error prone):

```
# Create SYNAPSE based on 'I' from syntypes library
InhSyn1 = b.Copy(b.syntypes, 'I', 'InhSyn1')
del InhSyn1.parms['PREV_SPIKE_RANGE']
del InhSyn1.parms['RSE_INIT']
del InhSyn1.parms['HEBB_END']
del InhSyn1.parms['HEBB_START']
newparms=[('ABSOLUTE_USE', (0.250, 0.0)), ('SYN_REVERSAL',
      (-80, 0.0)), ('SFD_LABEL', sfd_1),
      ('DELAY', (0.001, 0.001))]
InhSyn1.parms.update(newparms)

InhSyn1.parms['MAX_CONDUCT'] = ((G_inh/2.0), 0.0)
```

We omit the section of Brainlab code that creates the cells themselves, but the procedure is similar: a basic cell type is copied from the Brainlab library and a few parameters are selectively modified. The variables returned from the Brainlab function that creates the cells groups are stored in a Python list. So `e[0]` references the first group created, `e[1]` the second group created, and so on.

Brainlab provides a single, general `AddConnect(from, to)` method that can make connections at all three connection levels supported by NCS (within-layer, between-layer, and between-column). The modeler does not need to pay attention to NCS's distinction between these three levels of connection if this is not desired, and this encapsulation can hide much complexity from the user. Furthermore, connections can conveniently be made in Brainlab using the Python variables assigned to the created objects, rather than their underlying .in file text names (which the modeler can basically ignore). In our example of a 4000 neurons network, we do divide the network into five cell groups, so that it could be distributed to five computational nodes. The three types of inhibitory synapses connect to both inhibitory and excitatory neurons in the network:

```
# Connect inh RAIN network
b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, i0),
      InhSyn1, prob=0.00584, speed=0)
b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, i0),
      InhSyn2, prob=0.01166, speed=0)
```

```
b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, i0),
      InhSyn3, prob=0.00250, speed=0)

# Connect inh-exc rain network
for j in range(0, 4):
    tgt = e[j]
    b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, tgt),
          InhSyn1, prob=0.00152, speed=0)
    b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, tgt),
          InhSyn2, prob=0.01526, speed=0)
    b.AddConnect((col_0, lay_0, i0), (col_0, lay_0, tgt),
          InhSyn3, prob=0.00320, speed=0)
```

The short-term dynamics of inhibitory synapses not only maximize the synaptic diversity, but potentially constrain the functional impact of different interneurons on the long-term dynamics which exist among the excitatory neurons. To incorporate this idea into the model, we also include the spike timing dependent plasticity (STDP) within each cell group (Song et al., 2000). The Brainlab code for these connections is as follows:

```
for i in range(0, 4):
    src = e[i]
    # connect exc-inh rain network
    b.AddConnect((col_0, lay_0, src), (col_0, lay_0, i0),
          ExcSyn0, prob=0.02, speed=0)

    # connect exc-exc rain network
    for j in range(0, 4):
        tgt = e[j]
        if (i==j):
            b.AddConnect((col_0, lay_0, src), (col_0, lay_0,
                  tgt), ExcSyn1, prob=0.02, speed=0)
        else:
            b.AddConnect((col_0, lay_0, src), (col_0, lay_0,
                  tgt), ExcSyn0, prob=0.02, speed=0)
```

Even the fairly simple RAIN network example shown above results in a multi-thousand line .in file for NCS. The more concise, programmatic representation of the model in Brainlab makes it easier to create and also easier for others to quickly understand the true structure of the model.

DISCUSSION

We have shown elements of the design, implementation, and usage of Brainlab, a Python toolkit that leverages the strengths of Python to provide a more powerful and convenient interface to the NCS network simulator. We integrated Brainlab to NCS loosely, in a way that required no source code changes to NCS whatsoever. We were able to design a Pythonic neural modeling interface that can automatically convert an object representation into NCS's cumbersome .in representation. For simulation, we also integrate Brainlab loosely with NCS, using Python's sub-process management and standard operating system level tools like `ssh` for remote invocation as necessary.

Our approach gives us simplicity of implementation and ease of long-term maintainability, with no significant performance penalties on simulations, yet still extends to NCS all the considerable power and flexibility of Python and its numerical, graphical, special format file access, and other support packages.

Brainlab will likely remain the Python toolkit for NCS, and will see use for those applications where NCS's own strengths make it the tool of choice: large scale simulations with a medium degree of biological realism.

REFERENCES

- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., El Boustani, S., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398.
- Cannon, R., Gewaltig, M., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., and De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.
- Drewes, R. (2005a). Brainlab: A Toolkit to Aid in the Design, Simulation, and Analysis of Spiking Neural Networks with the NCS Environment. Master's Thesis, Reno, University of Nevada. Available at: <http://www.interstices.com/drewes/brain/thesis.pdf>
- Drewes, R. (2005b). Modeling the Brain with NCS and Brainlab. *Linux Journal*, pp. 58–61. Available at: <http://www.linuxjournal.com/article/8038>
- Drewes, R., Maciokas, J., Louis, S. J., and Goodman, P. (2004). An evolutionary autonomous agent with visual cortex and recurrent spiking columnar neural network. In Proceedings of the 2004 Genetic and Evolutionary Computing Conference (GECCO 2004), Vol. 3, Springer-Verlag, pp. 257–258.
- Goodman, P., Zou, Q., and Dascalu, S. (2008). Framework and implications of virtual neurorobotics. *Front. Neurosci.* 2, 123–129.
- Gupta, A., Wang, Y., and Markram, H. (2000). Organizing principles for a diversity of gabaergic interneurons and synapses in the neocortex. *Science* 287, 273–278.
- Markram, H., Wang, Y., and Tsodyks, M. (1998). Differential signaling via the same axon of neocortical pyramidal neurons. *Proc. Natl. Acad. Sci. U.S.A.* 95, 5323–5328.
- Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive Hebbian learning through spike-timing dependent synaptic plasticity. *Nat. Neurosci.* 3, 919–926.
- Vogels, T., and Abbott, L. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci.* 25, 10768–10795.
- Yamada, W.M., Koch, C., and Adams, P. (1998). Multiple channels and calcium dynamics, Chapter 4. In *Methods in Neuronal Modeling*, 2nd Edn. (Cambridge, MIT Press), pp. 97–133.

ACKNOWLEDGEMENTS

Portions of this work were supported by grants from the U.S. Office of Naval Research (grants N000140010420 and N000140510525).

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

Received: 17 September 2008; paper pending published: 21 October 2008; accepted: 09 May 2009; published online: 27 May 2009.
Citation: Drewes R, Zou Q and Goodman PH (2009) Brainlab: a Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the NeoCortical Simulator. *Front. Neuroinform.* (2009) 3:16. doi:10.3389/neuro.11.016.2009
Copyright © 2009 Drewes, Zou and Goodman. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



PCSIM: a parallel simulation environment for neural circuits fully integrated with Python

Dejan Pecevski^{1*}, Thomas Natschläger² and Klaus Schuch¹

¹ Institute for Theoretical Computer Science, Graz University of Technology, Graz, Austria

² Software Competence Center Hagenberg, Hagenberg, Austria

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Ingo Bojak, Radboud University
Nijmegen, The Netherlands
Abigail Morrison, RIKEN, Japan

*Correspondence:

Dejan Pecevski, Institute for Theoretical
Computer Science, Graz University of
Technology, Inffeldgasse 16b/1, A-8010
Graz, Austria.
e-mail: dejan@igi.tugraz.at

The Parallel Circuit SIMulator (PCSIM) is a software package for simulation of neural circuits. It is primarily designed for distributed simulation of large scale networks of spiking point neurons. Although its computational core is written in C++, PCSIM's primary interface is implemented in the Python programming language, which is a powerful programming environment and allows the user to easily integrate the neural circuit simulator with data analysis and visualization tools to manage the full neural modeling life cycle. The main focus of this paper is to describe PCSIM's full integration into Python and the benefits thereof. In particular we will investigate how the automatically generated bidirectional interface and PCSIM's object-oriented modular framework enable the user to adopt a hybrid modeling approach: using and extending PCSIM's functionality either employing pure Python or C++ and thus combining the advantages of both worlds. Furthermore, we describe several supplementary PCSIM packages written in pure Python and tailored towards setting up and analyzing neural simulations.

Keywords: neural simulator, parallel simulation, spiking neurons, Python, Boost.Python, Py++, PCSIM

INTRODUCTION

Given the complex nonlinear nature of the dynamics of biological neural systems, many of their properties can be investigated only through computer simulations. The need of researchers to increase their productivity while implementing increasingly complex models without each time having to reinvent the wheel has become a driving force to develop simulators for neural systems that incorporate best known practices in simulation algorithms and technologies, and make it accessible to the user through a high-level user-friendly interface (Brette et al., 2007). It has also been brought to attention that it is of importance to use large neural networks with biologically realistic connectivity (on the order of 10^4 synapses per neuron) as simulation models of mammalian cortical networks (Morrison et al., 2005). Simulation of such large models can practically be done only by exploiting the computing power and the memory of multiple computers by means of a distributed simulation.

There are different neural simulation environments presently available and although many of them were initially envisioned for a specific purpose and domain of applicability, during continuing development their set of features expanded to improve generality and support construction of a wide range of different neural models; see Brette et al. (2007) for a recent overview. The two most prominent tools are NEURON (Carnevale and Hines, 2006; Hines and Carnevale, 1997) and GENESIS (Bower and Beeman, 1998) which aim at simulation of detailed multi-compartmental neuron models and small networks of detailed neurons. Another class of quite general neural simulation environments which focus on the simulation of large-scale cortical network models and the improvement of their simulation efficiency through distributed computing include NEST (Gewaltig and Diesmann, 2007; Plesser et al., 2007), NCS (Brette et al., 2007) and SPLIT (Hammarlund and Ekeberg, 1998). There are also more dedicated neural simulation tools like

iNVT (iLab Neuromorphic Vision Toolkit)¹ which is an example of a package specifically tailored for the domain of brain-inspired neuromorphic vision. All of the above simulation environments support parallel simulation of one model on multiple processing nodes by using commodity clusters and many of them can also be run on super-computers. The simulation tool PCSIM described in this paper is designed for simulating neural circuits with a support for distributed simulation of large scale neural networks. Its development started as an effort to redesign the previous CSIM simulator² (Natschläger et al., 2003) and augment its capabilities, with the major extension being the implementation of a distributed simulation engine in C++ and a new convenient programming interface. The aim was to provide a general extensible framework for simulation of hybrid neural models that include both spiking and analog neural network components together with other abstract processing elements while making the setup and control of parallel simulations as convenient as possible for the user. Hence, given its current set of features, the PCSIM simulator is closest to the second group (NEST, NCS, SPLIT) of neural simulation environments mentioned above.

Performing a neural network simulation usually requires combined usage of several additional software tools together with the simulator, for stimulus preparation, analysis of output data and visualization. Being able to steer all the necessary tools from one programming environment reduces the complexity of setting up simulation experiments since all development can be done in a single programming language and the burden of developing utilities for conversion of data formats between heterogeneous tools is avoided. Given its object-oriented capabilities and its strong support

¹<http://ilab.usc.edu/toolkit/home.shtml>

²<http://www.lsm.tugraz.at/csim>

for integration with other programming languages, the Python programming language is a very promising candidate for providing such a unifying software environment for simultaneous use of various scientific software libraries. As Python is becoming increasingly popular in the scientific community as an interpreting language of choice for scientific applications, the developers of many neural simulator tools decided to provide a Python interface for their simulator in addition to its legacy interface in a custom scripting language. Moreover, a simulation tool called Brian which uses Python as an implementation language was recently developed to bring to the user the full flexibility of an interpreting language in specifying and manipulating neural models (Goodman and Brette, 2008).

In spite of the evident practical advantages in using Python as the single programming language for all tasks during a neural modeling life cycle, there is the apparent discrepancy between the need for computational performance of the simulation and construction of the model on one hand, and rapid development of the model on the other. Using C++ can solve the performance issue, but will decrease the productivity of the modeler and requires higher level of programming skills and experience. In contrast Python is easy to learn, flexible to use and significantly increases the productivity of the modeler, however it lags far behind C++ in performance³. Hence, instead of adopting a single language, an alternative is to enable an easy mix and match of both languages during the development of a model, i.e. to introduce a *hybrid modeling approach* (Abrahams and Grosse-Kunstleve, 2003).

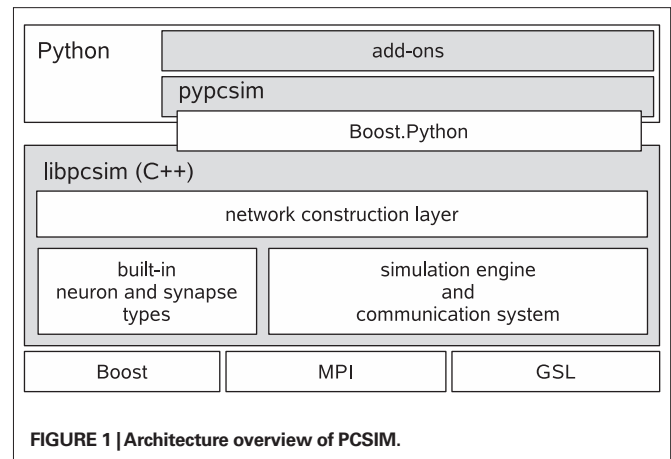
In this paper we will describe how the modular object-oriented framework of PCSIM in combination with an automated interface generation supports such a hybrid modeling approach.

In particular, we briefly review PCSIM's main features (see Overview) before we describe the automated process to generate the Python interface (see Python Interface Generation). In the Section "Network Construction" we detail PCSIM's network construction application programming interface (API), which is a central part of PCSIM's object-oriented modular framework. In the Section "Custom Network Elements" we demonstrate another advantage of the hybrid modeling approach: we show how PCSIM's concept of a general network element can be used as an interface to another simulation tool. While these examples concentrate on the Python aspect of the hybrid modeling, we show in the Section "Extending PCSIM Using C++" how the user can easily extend PCSIM's functionality using C++. Additional PCSIM packages implemented in Python are reviewed in the Section "PCSIM Add-Ons Implemented in Python". In the Section "Discussion" we discuss and summarize the presented concepts and approaches.

We would like to note that it is outside the scope of this article to describe the algorithmic aspects of PCSIM's computational C++ core (this will be reported elsewhere) and all the details of the full object-oriented modular framework.

OVERVIEW ARCHITECTURE

The high-level architecture of PCSIM is depicted in **Figure 1**. The PCSIM library written in C++ (`libpcsim`) constitutes the core



of the simulator. The API of the PCSIM library is exposed to the Python programming language by means of the Python extension module `pypcsim` (see Python Interface Generation for details). The library is made up of three main components: the simulation engine with its communication system, a pool of built-in network elements (i.e. neuron and synapse types) and the network construction layer. Before presenting the network construction layer in detail in the Section "Network Construction" we will briefly describe in the next paragraphs the main features of the underlying simulation engine and its communication system.

The simulation engine integrates all the network elements (typically neurons and synapses) and advances the simulation to the next time step, and uses its communication system to handle the routing and delivery of *discrete and analog messages* (i.e. spikes and e.g. firing rates or membrane voltages) between the connected network elements. PCSIM's simulation engine is capable of running distributed simulations where the individual network elements are located at different computing nodes. Setting up a distributed simulation is handled easily from a users point of view: there are no (or very little) code changes necessary when switching from a non-distributed to a distributed simulation. The distributed simulation mode is intended for employing a cluster of machines for simulation of one large network where each machine integrates the equations of a subset of neurons and synapses in the network. A distributed PCSIM simulation runs as an MPI⁴ based application composed of multiple MPI processes located on different machines⁵. The implementation of the spike routing, transfer and delivery algorithm between the nodes in a distributed simulation is based on the ideas presented in Morrison et al. (2005). In addition PCSIM offers the possibility to run a simulation as a multi-threaded application, both in a non-distributed and a distributed setup. The multi-threaded mode is intended for performing simulations on one multi-processor machine when one wants to split the computational workload among multiple threads in one process, each running on a different processor. However, we should note that the multi-threaded simulation engine is still undergoing optimization, as we are working on improvement of the scaling of the

⁴<http://www-unix.mcs.anl.gov/mpi/>

⁵To be precise, we use the C++ bindings offered by the MPICH2 library, where currently none of the advanced features of the MPI-2 standard are used.

³The simulation tool Brian mentioned above, heavily uses the numerical Python package `numpy` (Oliphant, 2007) written in C to achieve reasonable performance.

multi-threaded simulation to match the scaling achieved with an equivalent distributed simulation.

SCALABILITY AND DOMAIN OF APPLICABILITY

One of the goals of the development of PCSIM was enabling simulations of large neural networks on standard computer clusters through distributed computing. By utilizing the parallel capabilities of PCSIM the simulation time for a model can be reduced by using more processors (on multiple machines) as computing resources.

As a test of the scalability, we performed multiple simulations with the PCSIM implementation of the CUBA model described in Brette et al. (2007), with different number of leaky integrate-and-fire neurons (4000, 20000, 50000 and 100000) and distributed over a different number of processors (each processor on a different machine). We changed the resting potential in the neuron equations from -49 to -60 mV such that the network does not show any spontaneous activity. In order to elicit a spiking activity in the network, an input neuron population of 1000 neurons was connected randomly to it with probability 0.1, i.e. each neuron in the network receives inputs from on average 100 input neurons. The input neurons fired homogeneous Poisson spike trains at a rate of 5 Hz. The simulation was performed for 1 s biological time with a time step of 0.1 ms. We have set the connection probability within the network to 0.1, in order to reach realistic number of 10000 synapses per neuron for the network size of 100000 neurons. The transmission delay of spikes was set to 1 ms. We scaled the weights of the network so that the mean firing rate of the neurons was between 2.4 and 2.7 Hz for all network sizes (more precisely 2.68, 2.55, 2.52 and 2.45 Hz for the network with 4000, 20000, 50000 and 100000 neurons, respectively).

The used machines had Intel® Xeon™ 64 bit CPUs with 2.66 GHz and 4 MB level-2 processor cache, and 8 GB of RAM. They were connected in a 1 Gbit/s Ethernet LAN.

If we assume ideal linear speed-up, then the expected simulation time of a model on N machines given the actual simulation time on K machines is equal to the simulation time on K machines times K divided by N . In the evaluation of the scaling, for the estimation of the expected simulation time (see Figure 2) we used the measured simulation time of the model on the minimum number of machines used for that particular network size. Namely, we used the actual simulation time on $K = 1$ machine for the network sizes of 4000 and 20000 neurons, and the simulation time on $K = 4$ and $K = 16$ machines for the network sizes of 50000 and 100000 neurons respectively.

Figure 2 shows that in the case of 4000 neurons the computational load on each node is quite low, hence the cost of the spike message passing dominates the simulation time which results in sub-linear scaling. For the networks with 20000 and 50000 neurons the actual simulation time is shorter than the expected simulation time indicating a supra-linear speed-up for up to 24 nodes. For more than 24 nodes the actual simulation time approaches the expected simulation time. The reason for the supra-linear speed-up is more efficient usage of the processor cache when the network is distributed over larger number of nodes (Morrison et al., 2005). For the network with 100000 neurons the speed-up is not distinguishable from the expected linear speed-up (taking $K = 16$ nodes as the base measurement).

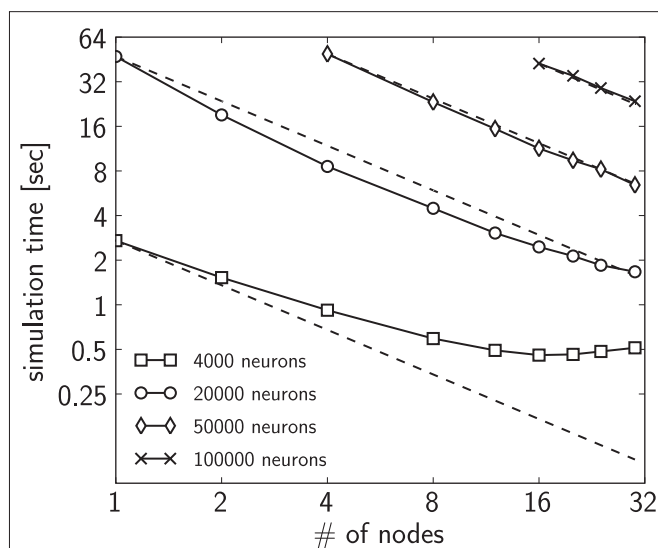


FIGURE 2 | Simulation times of the CUBA network distributed over different number of processing nodes, compared to the expected simulation time (dashed line) (see text for details). Four different sizes of networks were simulated: 4000 neurons with on average 1.6×10^6 synapses (squares), 20000 neurons with on average 40×10^6 synapses (circles), 50000 neurons with on average 250×10^6 synapses (diamonds) and 100000 neurons with on average 1×10^9 synapses (crosses). The plotted simulation times are averages over 12 simulation runs. The variation of simulation time between different simulation runs was small, therefore we did not show it.

The combination of features that PCSIM supports makes it suitable for various types of neural models. Its domain of applicability can be considered across two complementary aspects: the size of networks that can be simulated, and the variety of different models that can be constructed and simulated, determined by the available neuron and synapse models, plasticity mechanisms, construction algorithms and similar. Concerning the size of models, because of its distributed capabilities PCSIM is mainly targeted towards large neural systems with realistic cortical connectivity composed of 10^5 neurons and above. As the results from the scalability test show, a spiking network with 10^5 neurons and 10^4 synapses per neuron can be simulated in a reasonable time on a commodity cluster with about 20 machines, and the speed-up is linear when more machines are employed for the simulation. Regarding the support for construction of various different models in PCSIM, the generality of the communication system and the extensibility with custom network elements enables simulation of hybrid models (spiking and analog networks) incorporating different levels of abstraction. By utilizing the construction framework also structured models with diversity of neuron and synapse types and varying parameter values can be defined and simulated, and the built-in support for synaptic plasticity further expands the domain of usability towards models that investigate synaptic plasticity mechanisms.

PYTHON INTERFACE GENERATION

In order to enable a hybrid modeling approach we wanted to use a Python interface generation tool that was capable of wrapping PCSIM's object-oriented and modular API such that the Python

API will be as close as possible to the C++ API. Our choice for this purpose was the Boost.Python⁶ library (Abrahams and Grosse-Kunstleve, 2003). The strength of Boost.Python is that by using advanced C++ compile-time introspection and template meta-programming techniques it provides comprehensive mappings between C++ and Python constructs and idioms. There is support, amongst others, for exception handling, iterators, operator overloading, standard template library (STL) containers and Python collections, smart pointers and virtual functions that can be overridden in Python. The later feature makes the interface bidirectional, meaning that in addition to the possibility of calling C++ code from Python, user extension classes implemented in Python can be called from within the C++ framework. This is an enabler for the targeted hybrid modeling approach; we will see examples for this later on in this article.

However, using Boost.Python without any additional tools does not lead to a solution where the interface can be generated in an automatic fashion since for each new class added to the library's API one would have to write a substantial piece of Boost.Python code. As automatic Python wrapping of the C++ interface is one of the main prerequisites for leveraging a hybrid modeling approach, a solution is needed to automatically synchronize the Python and C++ API of a library like `libpcsim`. Fortunately, there exists the Py++ package⁷ which was developed to alleviate the repetitive process of writing and maintaining Boost.Python code. Py++ by itself is an object-oriented framework for creating custom Boost.Python code generators for an application library written in C++. It builds on GCC-XML⁸, a C++ parser based on the GCC compiler that outputs an XML representation of the C++ code. Py++ uses this structured information together with some user input, in form of a Python program, and produces the necessary Boost.Python code, constituting the Python interface for a specified set of C++ classes and functions (see Figure 3).

Finally the Boost.Python C++ code is compiled and linked together with the C++ library under consideration (`libpcsim` in our case) to produce the Python extension module containing the Python API of the library (`pypcsim` in our case). Thus, the work of the developer (and the user as we will see later on) reduces to a definition of high-level rules to select which classes and methods should be exposed.

For the generation of the PCSIM Python interface `pypcsim`, we split the rules Py++ needs into two subsets, inclusion and exclusion rules (see Figure 3). The inclusion rules contain the rules that mark a selected set of classes to be exposed to Python. The exclusion rules contain the post-processing, where some of the methods of the classes that were included in the inclusion rules are marked to be excluded, and call policies are defined for the included methods that require them⁹. Py++ allows to specify the rules in a high-level, generic fashion, making them robust to changes in the interface of the PCSIM C++ library. Hence, in most cases changes in the PCSIM API did not require changes in the Python program that generates the wrapper code, which simplified its maintenance. An example of such a high-level rule would be "In all classes that are derived from class A, do not expose the method that returns a pointer of type B". Such a general rule will then be still valid if for example we introduce more classes derived from A, or add additional functions that return a pointer of type B in some of the classes.

To summarize, the Python integration of PCSIM using Boost.Python together with the Py++ code generator allowed us to come up with a solution to automatically expose PCSIM's object-oriented and modular API bidirectionally in Python. In the following sections we will show how such an bidirectional integration of PCSIM into Python can practically be used and which possibilities and advantages arise.

NETWORK CONSTRUCTION

A large portion of the Python PCSIM interface is devoted to the construction of neural circuits. At the lowest level PCSIM provides methods to create individual network elements (i.e. neurons and synapses) and to connect them together.

On top of these primitives a powerful and extensible framework for circuit construction based on probabilistic rules is built. The source of inspiration for the interface of the framework was the Circuit Tool in the CSIM simulator¹⁰ and PyNN, an API for simulator-independent procedural definition of spiking neural networks (Davison et al., 2008). We will use a concrete example¹¹, described in more depth in the next subsection, to present the

⁶<http://www.boost.org/doc/libs/release/libs/python/doc/>

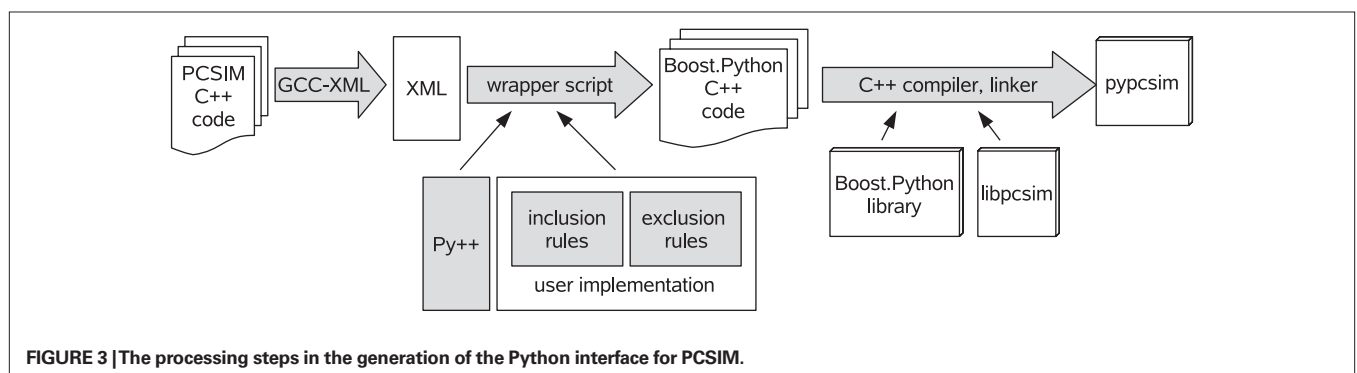
⁷<http://www.language-binding.net/>

⁸<http://www.gccxml.org>

⁹Call policies define the change of ownership of objects that cross the boundaries of the C++ library, i.e. the object passed from Python to the C++ library and from the C++ library to Python.

¹⁰<http://www.lsm.tugraz.at/circuits>

¹¹The full source code of this example is available in the Supplementary Material.



network construction framework and its typical use cases where emphasis is put on those features that were enabled by the bidirectional Python interface generated by the approach described in the Section “Python Interface Generation”.

THE EXAMPLE MODEL

We selected the model to be simple enough for didactic reasons, but complete enough with all the elements necessary to explain the main novel concepts of the interface and its Python extensibility features. The connectivity patterns are based on experimental data that we use in our current research work. The model consists of a spatial population of neurons located on a 3D grid with integer coordinates within a volume of $20 \times 20 \times 6$. 80% of the neurons in the model are excitatory, and the rest are inhibitory. The excitatory neurons are modeled as regular spiking and the inhibitory neurons as fast spiking Izhikevich neurons (Izhikevich, 2004). The connections between excitatory neurons in the network are created according to the trivariate probabilistic model defined in Buzas et al. (2006). This connectivity model describes the distribution of the excitatory patchy long-range lateral connections found in the superficial layers of the primary visual cortex in cats that depends on the lateral distance of the cells and their orientation preference. Orientation preference is the affinity of V1 cells to fire more when a bar with a specific orientation angle is present in their receptive fields. The connectivity rule is defined by the following equations that express the connectivity probability between two excitatory cells.

$$P(\mathbf{l}_i, \mathbf{l}_j, \phi_i, \phi_j) = CG(\mathbf{l}_i, \mathbf{l}_j)V(\phi_i, \phi_j) \quad (1)$$

$$G(\mathbf{l}_i, \mathbf{l}_j) = e^{-\frac{\|\mathbf{l}_i - \mathbf{l}_j\|^2}{2\sigma^2}} \quad (2)$$

$$V(\phi_i, \phi_j) = e^{\kappa \cos 2(\phi_i - \phi_j)} \quad (3)$$

$\mathbf{l}_i = (x_i, y_i)$ and $\mathbf{l}_j = (x_j, y_j)$ are the 2D locations and ϕ_i and ϕ_j are the orientation preferences of the pre- and post-synaptic neurons i and j . The function G introduces the dependence of the connectivity probability on the lateral distance between the neurons, and V models the dependency on the differences in the orientation preferences of the neurons. C , κ and σ are scaling coefficients. The values for the preferred orientation angles of the neurons in the example are generated by evolving a self-organizing map (SOM) (Obermayer and Blasdel, 1993). Additionally the conduction delay of a connection between

excitatory neurons is probabilistically dependent on the distance between the 3D locations of its pre- and post-synaptic neurons.

$$D(\mathbf{l}_i, \mathbf{l}_j) = D_0 \frac{|\mathbf{l}_i - \mathbf{l}_j|}{N(\mu, \sigma, b_l, b_u)} \quad (4)$$

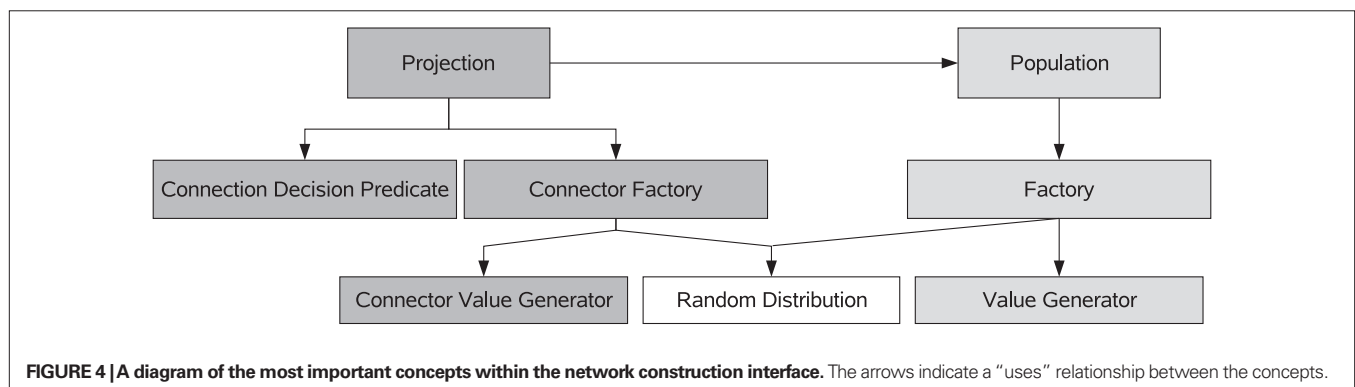
Here $N(\mu, \sigma, b_l, b_u)$ is a bounded normal distribution representing the transmission velocity of the axon. The $\mathbf{l}_i = (x_i, y_i, z_i)$ and $\mathbf{l}_j = (x_j, y_j, z_j)$ denote the 3D locations of the pre- and post-synaptic neurons i and j . A random value from $N(\mu, \sigma, b_l, b_u)$ is sampled as follows: first a random number from a normal distribution with mean μ and standard deviation σ is drawn and if that value is not within the range $[b_l, b_u]$, then another value is drawn from a uniform distribution with that range. D_0 represents a proper scaling factor in the formula.

THE FRAMEWORK: OBJECT-ORIENTED, MODULAR AND EXTENSIBLE

Figure 4 shows the basic concepts of PCSIM’s construction framework together with their interactions during the construction process. This framework allows model specification in terms of *populations* of neurons connected by probabilistically defined connectivity patterns called *projections*.

A population of network elements utilizes several object *factories* to generate the network elements. A factory encapsulates the logic for the neuron and synapse generation decoupled from the other parts of the construction process. Every time a new neuron is to be created in a population the factory is used to generate the neuron object. The object factories can use either *random distribution* objects or *value generators* to generate values for the *parameters* and *attributes* of the network element instances. When we talk about a parameter we mean a parameter of the differential equations used to model a neuron or synapse. In contrast an attribute describes any other (more abstract) property of a network element. In our example the orientation preference ϕ will be such an attribute of an excitatory neuron.

A projection manages connections between two populations. During the construction phase of a projection a *connection decision predicate* is used to determine whether a connection should be created for a pair of neurons. A *connector factory* is then used to create instances of the connector elements like synapses (this is analogous to the object factory for populations). The connector factory also uses *random distributions* or *connector value generators* for the parameter values of the connector elements. In order to implement a specific construction algorithm, the user typically just needs to implement custom *value generator* and *connection decision predicate* classes, as we will demonstrate in the following subsections.



FACTORIES: CREATING NETWORK ELEMENTS FROM MODELS

We will start constructing the network model by defining the classes (or families) of neuron models: inhibitory and excitatory neurons. This is accomplished by defining an element factory for each family. As explained in the definition of “The Example Model” the excitatory neurons have an orientation preference ϕ which depends on the location of the neuron in the population. For this reason we will associate the attribute `phi` with each excitatory neuron:

```
exc_factory = Factory
    (model = IzhiNeuron ( type = "RS" ),
     Vinit = UniformDistribution (-50e-3, -60e-3 ),
     attrs = dict( phi = OrientationPreferValGen() ) )
```

The statement above creates a factory for the excitatory family of neurons based on a regular spiking (RS) Izhikevich neuron model (Izhikevich, 2004) where `IzhiNeuron` is a built-in network element class. The keyword argument `Vinit = UniformDistribution(...)` associates a uniform random number generator with the initial membrane voltage `Vinit`. This has the effect that whenever the factory is used to generate an actual instance of an excitatory neuron, the parameter `Vinit` will be randomly chosen from the interval $[-50, -60]$ mV. Finally the keyword argument `attrs = dict(phi = ...)` has two effects: a) the attribute `phi` is attached to `exc_factory` and b) the custom *value generator* `OrientationPreferValGen` is used to generate a particular value for `phi` each time `exc_factory` is asked to generate an instance of an excitatory model neuron. The value of the `phi` attribute will be used afterwards for the creation of synaptic connections.

In the example we implement the custom value generator `OrientationPreferValGen` in pure Python. This is enabled by the particular feature of Boost.Python which allows C++ virtual functions to be overridden from within Python.

```
class OrientationPreferValGen(
PyAttributePopObjectValueGenerator):

    def __init__(self):
        PyAttributePopObjectValueGenerator.__init__(self)
        self.map = som.OrientationMapSOM([20,20])

    def generate(self, rng):
        return self.map.pref( self.loc().x(), self.loc().y() )
```

Value generators (in this case to be derived from `PyAttributePopObjectValueGenerator`) have a simple interface composed of the constructor `__init__` and the method `generate` which have to be implemented by the user. In our particular example we create the orientation map, that maps 2D coordinates to an orientation preference angle in the constructor, and will use it in the method `generate`. The map is based on the SOM algorithm encapsulated in the Python class `OrientationMapSOM` (details not relevant here). The `generate` method is called to determine the value of the orientation angle attribute `phi` whenever a neuron instance from the factory has to be created. The value generator inherits several convenient methods from its base class that one can use for accessing properties of the neuron for which `generate` is called, like `self.loc` to get the 3D location of the neuron within a population (see next section). We then pass the x and y coordinates to the orientation map (method `pref`) in order to calculate the value of the orientation preference angle.

For the inhibitory neuron model we create a similar factory:

```
inh_factory = Factory
    ( model = IzhiNeuron( type = "FS" ),
      Vinit = UniformDistribution(-50e-3, -60e-3),
      attrs = dict( ) )
```

The difference to the excitatory neuron model is that a fast spiking (FS) Izhikevich neuron model is used and the attribute dictionary `attrs = dict()` is empty. This is because there is no orientation preference of the inhibitory cells in the considered model.

NEURON POPULATIONS

A population in PCSIM represents an organized set of neurons that can be manipulated as one structural unit in the model. In the `AugmentedSpatialPopulation` that we will use in this example, the neurons have associated 3D coordinates, a family identifier, and an extensible set of custom attributes that the user can attach to each of the neurons. We already encountered this in the previous section. The family identifier allows the definition of multiple families/classes of neurons, i.e. subsets of neurons with similar properties, within a single population. Our population will have two families of neurons, the family of excitatory and the family of inhibitory neurons. For each of the two families of neurons we have specified in the previous section a factory that will be used to generate the neuron instances within the population.

```
pop = AugmentedSpatialPopulation
    ( net, [ exc_factory(), inh_factory() ],
      RatioBasedFamilies( [ 4, 1 ] ),
      CuboidIntegerGrid3D( 20, 20, 6 ) )

exc_pop, inh_pop = pop.splitFamilies()
```

Note that the first argument (`net`) specifies the overall network to which this population of neurons will belong. The class `CuboidIntegerGrid3D`, which is a built-in specialization of the more general concept of an arbitrary set of points in 3D, defines the possible locations for the neurons (integer coordinates within a volume of $20 \times 20 \times 6$). The population is to be composed of two families of neurons (excitatory and inhibitory), created by the two given factories (`exc_factory` and `inh_factory`). To accomplish this we use a `RatioBasedFamilies` object which randomly chooses for each 3D location from which family of neurons the particular instance will be created. Specifying the ratio 4:1 for excitatory to inhibitory neurons yields the desired 80% excitatory neurons. The class `RatioBasedFamilies` is a built-in specialization of the general concept of a *spatial family identifier generator* which encapsulates the logic for deciding which factory to use depending on the 3D location.

For the purpose of more convenient setup of connections later on, the created population is split into two sub-populations, one for each family.

PROJECTIONS: MANAGING SYNAPTIC CONNECTIONS

The synaptic connections in the network construction interface are created by means of projections. A projection is a construct that represents a set of synaptic connections originating from one population of neurons and terminating at another population¹².

¹²The source and destination populations can be the same if the goal is to create recurrent connections in one population.

PCSIM has built-in construction algorithms for creating various types of connection projections, like constant probability random connectivity or random connectivity with probability dependent on the distance (or lateral distance) between the neurons.

However, to create a projection with a specific connectivity pattern, one usually defines a custom *connection decision predicate*. A decision predicate decides for an individual pair of neurons whether to form a connection based on the parameters and attributes of those neurons. In our example we implemented the connection decision predicate `OrientationSpecificConnPredicate` in pure Python, encapsulating the probabilistic rule for connection making from Eq. 1, which states that the connection probability depends on the distance between, and the orientation preferences of the pre- and post-synaptic neurons.

```
class OrientationSpecificConnPredicate
(PyAugmentedConnectionDecisionPredicate):

    def __init__(self, C):
        PyAugmentedConnectionDecisionPredicate.__init__(self)
        self.orient_conn_prob = OrientationSpecConnProbability(C)
        self.unidist = UniformDistribution(0.0, 1.0)

    def decide(self, src, dst, rnd):
        prob = self.orient_conn_prob(self.src_attr(src, 'phi'),
                                     self.dest_attr(dst, 'phi'),
                                     self.dist_2d(src, dst))
        return self.unidist(rnd) < prob
```

The `PyAugmentedConnectionDecisionPredicate` base class is used when one has to define a custom connection decision predicate that uses the neuron attributes and connects neurons from populations of type `AugmentedSpatialPopulation`. To complete the implementation of the predicate, it is required to override the `decide` method and fill the constructor with the necessary initializations. The method `decide` is called within the connection construction process for each candidate pair of neurons that could be connected and is expected to output true (make a connection) or false (no connection). In our example, we create an instance (`orient_conn_prob`) of the `OrientationSpecConnProbability` class to calculate the probability according to the Eq. 1 (the full implementation of the class is available in the Supplementary Material). This instance is called in the `decide` method with the orientation preferences of the candidate source and destination neurons and their lateral distance as arguments. The orientation preferences are obtained via the `src_attr` and `dest_attr` methods (inherited from the base class), and the lateral distance via the `dist_2d` method. By comparing a uniformly distributed random number to the calculated probability a Bernoulli distribution with the desired probability for the outcome true is generated.

Before we can create the projection we have to define a connector factory (class `ConnFactory`) that will be used to generate the synapse objects within the projection.

```
ee_syn_factory = ConnFactory
    ( model = StaticSpikingSynapse(W = 1e-4),
      delay = DelayCond(v_mean = 2e2, v_SH = 0.2,
                       v_min = 0.1e-3, v_max = 5e-3) )
```

The connector factory differs from the element factory objects used in conjunction with neuron populations, in that the parameters of the created objects (typically synapses) can depend on the attributes of the source and destination network elements they are

connecting. In our example, the connector factory for the connections between excitatory neurons is based on a current-based synapse model with exponentially decaying post-synaptic response (class `StaticSpikingSynapse` in PCSIM). Additionally, the `DelayCond` value generator is associated to the delay parameter of the synapse, which produces distance dependent delay values according to Eq. 4. The `DelayCond` is a built-in value generator in PCSIM.

Now we can create the projection that will generate all recurrent connections between the excitatory neurons.

```
ee_proj = ConnectionsProjection
    ( exc_pop, exc_pop, ee_syn_factory(),
      PredicateBasedConnections
    ( OrientationSpecificConnPredicate( 1.0 ) ) )
```

We specify in the constructor of the projection the connectorfactory for generation of the synapses and the `PredicateBasedConnections` class instance that iterates over all candidate pre- and post-synaptic neurons and delegates the decision whether to make a connection to the connection decision predicate `OrientationSpecificConnPredicate` given as an argument.

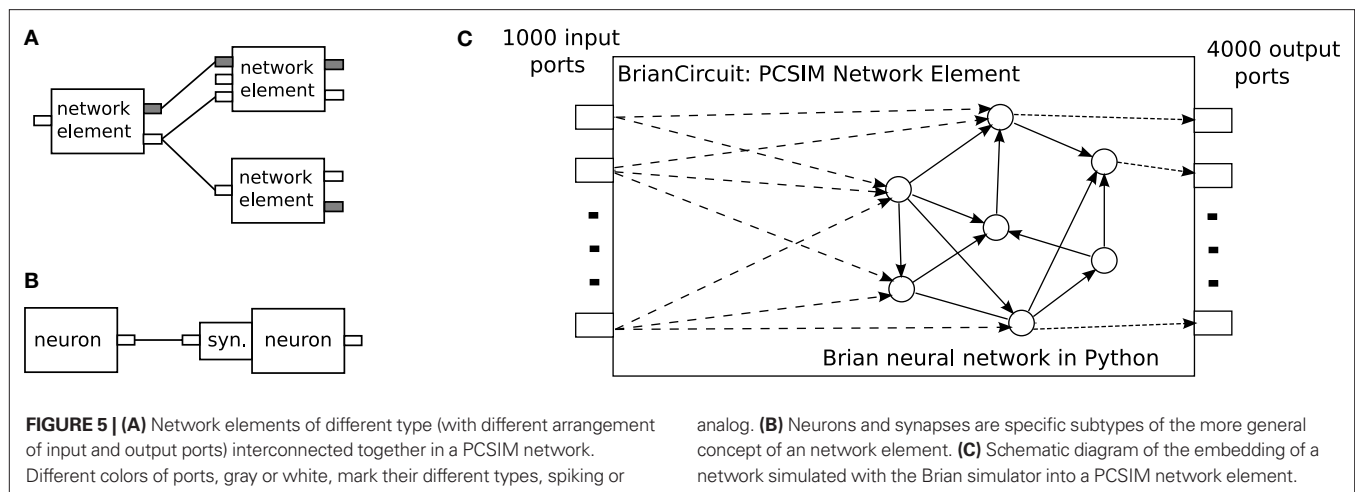
A connection decision predicate is typically used when in the probabilistic connectivity definition the probability that two neurons are connected depends on the attributes and parameters of the two neurons and is independent from the other created connections. In the general case, with such a connectivity, a separate decision whether to make a connection has to be made at each candidate neuron pair, yielding a complexity of the wiring algorithm that is quadratic with respect to the number of neurons. In a distributed scenario, a speed-up of the construction is possible by splitting the wiring workload among the multiple machines the model is simulated on. If the number of machines is increased with the number of neurons, keeping the number of neurons per node fixed, and if we assume that the number of input synapses per neuron does not increase, then the wiring time will scale linearly with the number of neurons.

For other connectivity schemes where further optimizations are possible, a faster wiring algorithm can be implemented directly in the class that iterates over the neuron pairs. For example, for the case of constant probability random connections, a special `RandomConnections` class that implements faster wiring can be used instead of `PredicateBasedConnections`. When using the `RandomConnections`, the wiring time is proportional to the number of created connections if the network is constructed on a single machine, and remains constant in the distributed case with the assumption that the number of machines is increased proportionally with the number of neurons¹³.

CUSTOM NETWORK ELEMENTS

The PCSIM communication system is general in a sense that it supports spiking and analog messages as communication between network elements. The network elements are not restricted to one type of message and can have multiple input and output ports, each of them capable of either receiving or sending spiking or analog messages (see **Figures 5A,B**).

¹³It is out of scope of this article to detail the algorithms behind the efficient implementation of the network construction framework in the distributed simulation scenario; this will be reported elsewhere.



The generality of the framework allows the user to implement custom processing elements that map multiple inputs to multiple outputs and plug them in a network model inter-connected together with spiking or analog neural networks. Such custom network elements can either be implemented in C++ (see Extending PCSIM Using C++) or in pure Python. This feature of PCSIM has various potential uses. For example the user can implement new neuron types for a preliminary experiment in Python first, instead of directly implementing them in C++. Another possible usage is to implement more abstract or complex elements like a whole population of spiking neurons in Python by using vectors from the numerical Python package *numpy*¹⁴ (Oliphant, 2007) for step-by-step integration of the equations. This approach has been shown to have good performance, and is applicable for homogeneous neuron populations, where all neuron instances have the same neuron model (Brian simulator, Goodman and Brette, 2008).

We detail such an example in this section, where the Brian simulator is used to implement a population of spiking neurons as a *single* network element, and then plug it into a PCSIM simulation together with other built-in network elements.

The spiking neural network model we will simulate with Brian is the modified version of the CUBA benchmark model described in the Section “Overview”, with a network size of 4000 neurons. We have used the same connectivity probability of 0.02 and the same weights as in Brette et al. (2007), instead of the modified 0.1 connectivity probability and scaled weights in the Section “Overview”. The PCSIM network element that we will create to encapsulate the Brain network has 1000 spiking input ports and 4000 spiking output ports (see Figure 5C). Each of the output ports is associated to one neuron.

To implement this model as a PCSIM network element, one has to implement a Python class *BrianCircuit* derived from *PySimObject*. In the constructor of this class the Brian spiking network is created and initialized.

```
class BrianCircuit(PySimObject):
    def __init__( self ):
        PySimObject.__init__( self )
```

```
self.registerSpikingOutputPorts(arange(4000))
self.registerSpikingInputPorts(arange(1000))
input = PCSIMInputNeuronGroup(1000, self)
self.P = P = brian.NeuronGroup(4000, model = eqs,
                                threshold = -50*mV, reset = -60*mV)

Pe = P.subgroup(3200)
Pi = P.subgroup(800)
Ce = brian.Connection(Pe, P, 'ge' )
Ci = brian.Connection(Pi, P, 'gi' )
Ce.connect_random( Pe, P, p = 0.02, weight = 1.62*mV )
Ci.connect_random( Pi, P, p = 0.02, weight = -9*mV )
Cinp = brian.Connection( input, P, 'ge' )
Cinp.connect_random( input, P, p = 0.1,
                    weight = 3.5*mV)

self.brian = brian.Network(input, P, Ce, Ci, Cinp )
self.brian.prepare()
self.brian.clock.set_duration(2.0*second)
```

The mapping of the PCSIM input ports to a Brian neuron group is managed by the simple auxiliary neuron group named *PCSIMInputNeuronGroup* (see the Supplementary Material for the implementation). The *reset* method resets the state of the network to time step $t = 0$, which is achieved by calling the *reinit* method of the Brian network, and initializing the membrane potential vector $P.V$ to random values from an uniform distribution.

```
def reset(self, dt):
    self.brian.reinit()
    self.P.V = -60*mV + 10*mV*rand(len(self.P))
    return 0
```

The step-by-step iteration of the network is done in the overridden *advance* method which performs one time-step update of the Brian network with the *update* method and the *tick* method of the associated Brian clock object. At the end of each time step the generated spikes of the population are gathered and delivered to the output ports of the PCSIM network element.

```
def advance(self, ai):
    self.brian.update()
    self.brian.clock.tick()
    self.setOutputSpikes( ai, self.P.get_spikes() )
    self.clearSpikeBuf()
    return 0
```

¹⁴<http://numpy.scipy.org>

Note that no Python loops are present, the `setOutputSpikes` method that transfers the spikes is implemented in C++ in the base class `PySimObject`, so there is no performance loss caused by the transfer of spikes from Brian to PCSIM and vice versa.

The new `BrianCircuit` network element class can then be instantiated and added to a PCSIM simulation. The following code segment creates an instance of the Brian spiking network, adds it as a network element, sets up the input and runs the simulation for 2.0 s [1000 neurons that emit Poisson spike trains at rate 5 Hz (`PoissonInputNeuron`) are connected to the 1000 input ports of the Brian network element]¹⁵.

```
net = SingleThreadNetwork()
inpNrnPop = SimObjectPopulation
    ( net, PoissonInputNeuron( rate = 5,
        duration = 1000 ), 1000 )

pycirc = BrianCircuit()
pycirc_id = net.add(pycirc)

for i in range(inpNrnPop.size()):
    net.connect(inpNrnPop[i], 0, pycirc_id, i)

net.reset()
net.simulate( 2.0 )
```

EXTENDING PCSIM USING C++

The object-oriented framework of PCSIM can be extended by the user at many different levels. Typical extensions of PCSIM include either implementations of new neuron and synapse types, or implementations of classes encapsulating custom construction rules in the network construction interface, as we have illustrated in the previous sections. By utilizing the features of the Boost.Python library and Py++, the extensions can be implemented either in pure Python as already shown or in C++.

For creating C++ extensions, PCSIM provides a tool that compiles the custom C++ classes, *automatically* generates the Python wrapper interface for these and packs everything into a separate Python extension module. In order to simplify the procedure of creating a custom extension, the user starts the implementation from an extension template contained in the PCSIM distribution. Let us assume that we want to implement two classes: a new neuron type `MyNeuron` and a new synapse type `MySynapse`. Once the C++ implementation is finished, there are three additional steps that have to be done to produce the PCSIM extension module.

First, the C++ source files of the extension have to be enlisted in the file `module_recipe.cmake`. This file is read by PCSIM's C++ build tool CMake¹⁶.

```
SET( MODULE_SOURCES
    src/MySynapse.cpp
    src/MyNeuron.cpp
)
```

As the second step, we have to specify the names of the classes we want to include in the Python interface in the file `python_interface_specification.py` which holds the extension

module interface specification. For our example the inserted lines should look like:

```
def specify( M, options ):
    M.class_( 'MySynapse' ).include()
    M.class_( 'MyNeuron' ).include()
    return M
```

Note that the argument `M` in the code above denotes the Py++ representation of the C++ code of the custom PCSIM extension to be built, with its rather intuitive query interface.

The name of the extension module (in our example `my_pcsim_module`) is specified in both `module_recipe.cmake` and `python_interface_specification.py` files. Finally, the compilation is done using the special purpose command-line compilation tool for PCSIM extensions:

```
> python pcsim_extension.py build
```

The compiled extension module then can be imported and used within Python as any other module.

```
import pypcsim
import my_pcsim_module
```

The main `pypcsim` module should always be imported before any PCSIM extension modules, because the classes in the extension are derived from classes in `pypcsim` and these classes should be already in the Python namespace. The user can develop multiple PCSIM extension modules that can be used simultaneously in one simulation.

The creation of a PCSIM extension as a separate Python extension module relies on the support of Boost.Python and Py++ for component-based development, so that C++ types from one Python extension module can be passed to functions from another extension module while still preserving the information about the cross-module C++ inheritance relationships. This enables object instances from the classes in the extension module to be used within the PCSIM object-oriented framework in the main `pypcsim` module. The component-based development has also the advantage that during the development of new custom classes only the extension module has to be recompiled, not the whole `pypcsim` library.

During the compilation of the PCSIM extension module the same processing steps happen as for the main `pypcsim` module (see **Figure 3**). We use the same scripts both for generation of the Python interface of the main PCSIM package and for the Python integration of PCSIM extension modules. Since the post-processing exclusion rules are expressed with the Py++ query interface in a generic way, they are applicable also to the wrapping of the extension classes. This is due to the fact that extension classes are derived from base classes in the PCSIM object-oriented framework and as such share their common properties on which the rules are based. Hence, the interaction of the user with the interface generation and the module compilation reduces to specifying a list of the C++ source files, and a list of classes to be exposed in Python. The rest of the process is automatized and the details are hidden behind the command-line interface of the special compilation tool for PCSIM extensions.

PCSIM ADD-ONS IMPLEMENTED IN PYTHON

On top of the main PCSIM Python API (encapsulated in `pypcsim`) several additional packages have been developed. They are

¹⁵The `net.connect(src_id, src_port, dest_id, dest_port)` method connects the port number `src_port` of the element with id `src_id`, to the port number `dest_port` of the element with id `dest_id`.

¹⁶<http://www.cmake.org>

implemented in pure Python and heavily rely on many third party scientific Python packages. The purpose of these packages is either to augment the capabilities of PCSIM, or add additional separate functionalities that are suitable to be used together with PCSIM.

PyNN.PCSIM

The objective of the PCSIM development to adopt ongoing initiatives to define standards for model specification of neural networks that would foster interoperability between different simulators is reflected in the support of the PyNN project¹⁷ (Davison et al., 2008). The PyNN project is an effort to create a standardized, unified Python-based API for procedural specification of neural network models aiming at easier exchange of models between simulators. The user interface of PCSIM has been augmented with an additional software layer to support the PyNN API making it possible to use models specified in PyNN within PCSIM. Due to the fact that PyNN was one of the sources for inspiration of the PCSIM interface, the concepts between the two interfaces match closely, so the translation of the PyNN statements in corresponding PCSIM statements was straightforward and did not require substantial programming logic that could have hindered the performance of the interface. The `pyNN.pcsim` package is an integral part of the PyNN distribution.

PYPCSIMPLUS

After we started to use PCSIM for our simulation purposes, it was becoming apparent that adding another layer above the interface of the `pypcsim` module can greatly simplify the routine tasks that are usually performed while setting up and running simulations. The `pypcsimplus` package was created with the intention to fill this gap. Note that the `pypcsimplus` package is dependent on PCSIM. For a more comprehensive, simulator independent tool-set for neural simulations, we refer the reader to the NeuroTools package¹⁸. In the following paragraphs we will describe two main components of the `pypcsimplus` package and give a demonstration of its use¹⁹.

Recordings

In PCSIM the value of a parameter or output port is recorded during a simulation by connecting it to a proper recording network element. The purpose of the `Recordings` class is to provide simpler means to set up recorders and saving the recorded data during a PCSIM simulation. For example it allows to create a population of recorders that record the activity of a population of elements with each recorder connected to one of the elements (e.g. the spiking output of a population of neurons). For example

```
r = Recordings(net)

r.spikes = nrn_popul.record( SpikeTimeRecorder() )
r.Vm     = net.record( my_nrn, 'Vm', AnalogRecorder() )
r.weights = synapses.record( AnalogRecorder
                             ( samplingTime ), 'w' )
```

¹⁷<http://neuralensemble.org/trac/PyNN>

¹⁸<http://neuralensemble.org/trac/NeuroTools>

¹⁹There are other miscellaneous utilities present within the `pypcsimplus` package, as for example tools for easier management of IPython parallel computing cluster instances, routines for inspection of the structure of an already created networks in PCSIM and routines for processing and analysis of spike train data.

schedules the recording of all spikes in the population `nrn_popul`, the membrane potential `Vm` of a single neuron (`my_nrn`), and the weights of a group of plastic synapses. To save that data to an HDF5 file²⁰ one would use the command

```
r.saveInOneH5File(f)
```

At any time later on, the saved data can be loaded from the file in a new `Recordings` object.

```
r = constructRecordingsFromH5File(f)
plot(r.Vm)
```

The members and attributes of the newly created `Recordings` object `r` are numpy arrays or Python lists holding the recorded data. For example `r.Vm` and `r.W` will be numpy arrays with the recorded values of the membrane potential of the neuron and with the evolution of the recorded synaptic weights during the simulation, respectively. Note that if the user switches to a distributed simulation the same code, without any changes, can be used.

To summarize, the `Recordings` class simplifies the specification, storage and retrieval of recorded data by

- providing automatic detection of the type of the recorded data based on the recorder classes, and conversion of the recorded data to appropriate HDF5 data structures.
- implementing automatic gathering and sorting of recorded data from all processing nodes in a distributed simulation, and saving it in HDF5 in the same format as if the simulation was executed on a single node.

These functionalities are hidden behind a convenient user interface and are manipulated in the same manner in both single-node and distributed simulation modes. For the implementation of the `Recordings` class, the `mpi4py`²¹ (Dalcín et al., 2008) and `pytables`²² packages were used.

Experiment-model framework

Simulation, modeling and development environments in various fields (e.g. electronic circuit design, software engineering, signal processing, mechanical engineering) usually include a library of already developed reusable components that are readily available to the modeler. In the area of computational neuroscience, there is a similar effort to provide resources for easier reusability of models, e.g. online databases of already published models (Hines et al., 2004), or constructs within the simulator that allow encapsulation of a simpler model as a well-defined component that can be used as a building block at a higher-level of abstraction. As a first step towards a component-based modeling with PCSIM, we have set up a light-weight framework that could leverage and encourage encapsulation of some generic parts of a model as reusable components, which can be exchanged among modelers.

The basis of the framework is composed of three classes: `Model`, `Experiment` and `Parameters`. The `Model` is a base class which the user inherits from when he wants to develop a model component. Several model components can be combined together to create a

²⁰<http://www.hdfgroup.org/HDF5/>

²¹<http://mpi4py.scipy.org>

²²<http://www.pytables.org/moin>

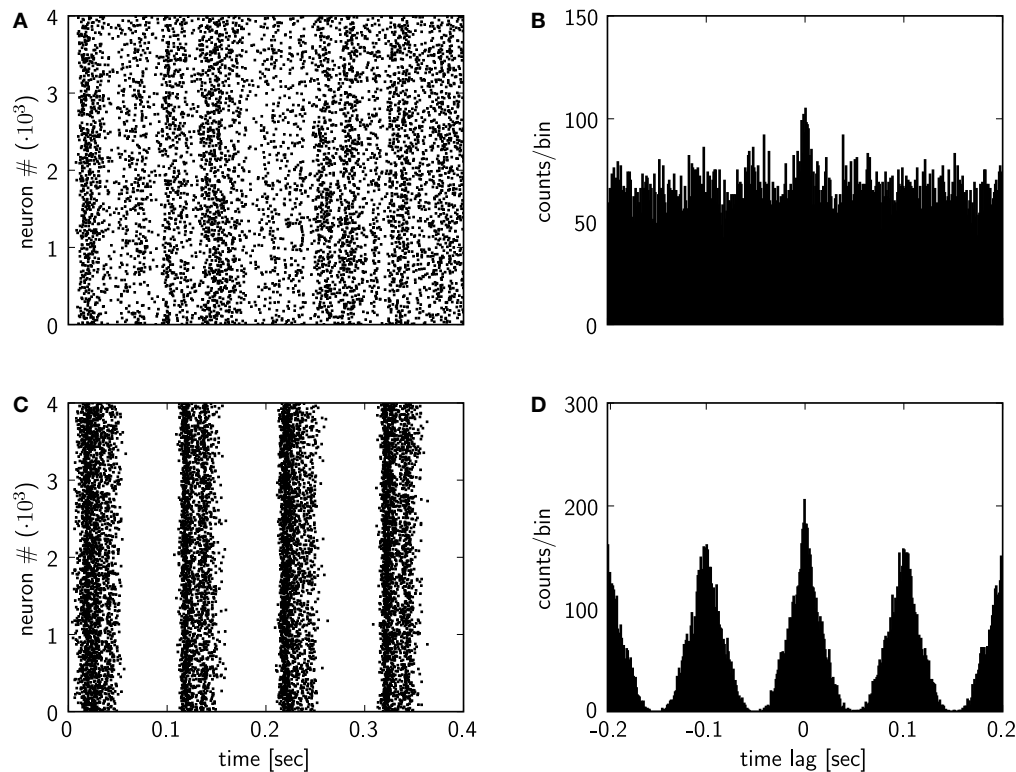


FIGURE 6 | Plots from the output analysis example with the **pypcsimplus** package. **(A)** Spike response of the spiking network implemented in the Section “Custom Network Elements”, with input neurons emitting spikes generated from a homogeneous Poisson process with a rate of 5 Hz, for the first 0.4 s of the simulation. **(B)** Cross-correlogram of the spike

response of the network model from **(A)**. **(C)** Spike response of the spiking network implemented in the Section “Custom Network Elements”, when the input neurons emit spikes generated from an inhomogeneous Poisson process with a rate changing according to a sinusoidal function (see text for details). **(D)** Cross-correlogram of the spike response of the network model from **(C)**.

new model component. The `Experiment` class provides means to perform a controlled simulation with an already developed custom `Model` class. It encapsulates different facilities regarding saving output data to files, configuration of models, saving the current version of the scripts, naming of different runs of experiments etc. The configuration of the models is done with a `Parameters` class holding the model parameters in a hierarchical structure. For creating instances of the `Experiment` and `Model` classes remotely within the IPython parallel computing framework²³ (Pérez and Granger, 2007) there are `RemoteExperiment` and `RemoteModel` proxy classes, which can be used to manipulate remote experiment and model instances in the same way as if they were local.

Pypcsimplus in action

We will demonstrate in the following paragraphs how **pypcsimplus**, together with other general scientific and computational neuroscience Python packages, can be utilized to perform an analysis of the activity of the Brian spiking network example from the Section “Custom Network Elements”. In particular we will investigate what effect a change in the injected input in the network will have on the cross-correlogram of its spike response.

At the beginning we will set up the recording of the spiking output of all 4000 neurons in the network. After creating a `Recordings`

object, we create a population of recorders to record the spikes from the 4000 output ports of the `BrianCircuit` network element.

```
r = Recordings()
r.spikes = record_ports(net, pycirc_id, range(4000),
                        SpikeTimeRecorder())

net.simulate(2.0)

r.saveInOneH5File('results.h5')
```

We have accomplished this by using the `record_ports` function from the **pypcsimplus** package, used to specify recording of a set of output ports. After the simulation is performed, the recordings are saved in a HDF5 file for subsequent retrieval.

In another script we setup the analysis of the output data and the plotting. After the creation of the `Recordings` object by loading the recorded data from the saved HDF5 file, we plot the spiking activity of the network for the first 0.4 s of the simulation with the `plot_raster` function in **pypcsimplus** (see **Figure 6A**).

```
r = constructRecordingsFromH5File('results.h5')

figure(1)
plot_raster(r.spikes, time_range = (0,0.4), fmt = ',')
```

`plot_raster` uses the plotting routines from the `matplotlib`²⁴ package (Hunter, 2007) to realize the plotting.

²³<http://ipython.scipy.org>

²⁴<http://matplotlib.sourceforge.net>

Additionally we will calculate and plot the cross-correlogram of the spiking activity, defined as the histogram of time differences between the spike times from two different spike trains, calculated and summed over a set of randomly chosen pairs of neurons from the network. To achieve this, we utilize the `pypcsimplus` function `avg_cross_correlate_spikes`.

```
corr = avg_cross_correlate_spikes(r.spikes, num_pairs = 2000,
                                binsize = 1e-3,
                                corr_range = (-200e-3, 200e-3))

figure(2)
bar(arange(-200e-3, 201e-3, 1e-3), corr, width = 1e-3,
    color = 'k')
```

In our case the cross-correlogram is calculated from the spike times of 2000 randomly chosen pairs of neurons from the network, for time lags within the range $[-200 \text{ ms}, 200 \text{ ms}]$ and a bin size of 1 ms. We then plot the cross-correlogram values with the `bar` function from `matplotlib` (the plot is shown in **Figure 6B**)²⁵.

In the example in the Section “Custom Network Elements”, the input neurons were setup to generate a homogeneous Poisson spike trains with 5 Hz rate. Now we will modify the input generation so that the input neurons will emit inhomogeneous Poisson spike trains, with a firing rate $r(t) = 5(1 + \sin(2\pi 10t))$. First we create a population of input neurons of type `SpikingInputNeuron` that emit an explicitly given sequence of spike times.

```
inpNrnPop = SimObjectPopulation
            (net, SpikingInputNeuron(), 1000)
```

Then we iterate through all the input neurons and set the spike sequence of each input neuron according to the previously defined inhomogeneous Poisson process. For the generation of the inhomogeneous Poisson spike time sequences we invoke the `inh_poisson_generator` method of an instance of the `StGen` (stimulus generator) class available in the `NeuroTools` Python package for computational neuroscience. The method accepts three parameters, a sequence specifying the time moments where the rate changes (parameter `t`), the sequence of the new firing rate values at these time moments (parameter `rate`) and the duration of the spiking process (parameter `t_stop`)²⁶.

```
time_steps = arange(0, 2000, 1); stgen = StGen()
for i in range(inpNrnPop.size()):
    spikelist = stgen.inh_poisson_generator
                (rate = 5*(1 + sin(time_steps/1000.0*20*pi)),
                 t = time_steps, t_stop = 2000.0)
    inpNrnPop.object(i).setSpikes(spikelist.spike_times/1000)
```

The spike raster and the cross-correlogram obtained after rerunning the simulation with the newly defined input are shown in **Figures 6C,D**, respectively.

Through this demo we have elucidated to the reader how a typical PCSIM simulation run is performed in Python, and the benefits that come from the utilization of Python as a unifying

scripting environment within which PCSIM is used together with its add-on `pypcsimplus` and other scientific and computational neuroscience Python packages. Additionally to their side-by-side usage with PCSIM, the Python scientific packages are harnessed also in the bundling of common recipes and reoccurring usage patterns in the PCSIM extra add-on packages, as in the case of `pypcsimplus`. The collection of Python scientific packages presently available cover a broad enough range of functionalities to enable, in almost all cases, handling all of the steps of a modeling effort in Python (e.g. stimulus preparation, response analysis and plotting as shown in the demo). The data communication between the different packages and PCSIM typically reduces to passing Python sequences (lists or numpy arrays) from one package to another.

PYLSM

The `pylsm` package is aimed to support the analysis of the computational properties of cortical microcircuits within the liquid state machine (LSM) approach (Maass et al., 2002). In this approach multiple simulation trials are performed where input spike trains, drawn from a defined input distribution, are injected in the cortical circuit, and a readout which reads the spiking activity of the circuit is trained by a supervised learning algorithm to approximate some function of these inputs.

The framework contains all the necessary machinery for performing the simulations and the training of the readout²⁷. In a typical task the user defines the neural circuit to be used as a liquid, chooses the desired input distribution, the input-output mapping function, and the learning algorithm for the readout from the ones available in the package, and then performs the LSM training and testing procedures. For example, the user can define a distribution of inputs which consist of different time segments, and each of these time segments contains a jittered version of some predefined spike train template. In the available learning algorithms for the readout a least-square algorithm with non-negative constraints is also included. It can be used to train a linear readout with the biologically more realistic constraint that all the weights originating from excitatory (inhibitory) neurons are positive (negative) (Haeusler and Maass, 2007).

DISCUSSION

The application programming interface of PCSIM is an object-oriented framework composed of many classes interacting together to achieve the desired operation. Within this framework we introduced several novel concepts like element and connector factories, value generators and connection decision predicates. The user can customize and extend this framework by deriving from the interface classes of the API to implement his own specific network elements or network construction algorithms.

THE WRAPPING APPROACH

There exist several possible approaches for implementing a Python interface of a simulation software library implemented in C/C++. An extension to the NCS software called Brainlab (Drewes, 2005) uses generation of a file from Python with declarative specification

²⁵For clarity reasons, we only give the main `matplotlib` plotting command in the example code blocks, and omit the additional formatting commands used for **Figure 6**.

²⁶Time in `neurotools` is specified in milliseconds, hence the division by 1000 when we need to convert the spike time sequence in seconds before inserting it in a PCSIM neuron.

²⁷It has similar features as the package described in Natschlager et al. (2003), which was implemented in Matlab and was part of the CSIM package.

of the model which is then loaded in the simulator. Another common method is to use interpreter-to-interpreter interaction with the conversion of data structures between Python and C++ handled by means of the Python/C API, an approach adopted by NEURON (Hines et al., 2009) and NEST (Eppler et al., 2008). This method is applicable only if the simulator already has an interpreting interface. For the creation of PyMoose (Ray and Bhalla, 2008), the Python interface of MOOSE²⁸, the developers applied the interface generator tool SWIG²⁹ (Beazley, 2003). Certainly, one can also implement a Python interface by using solely the Python/C API.

Since PCSIM's Python interface was to be newly developed, only the later two options were applicable. We opted for the interface generator tool approach combined with automatic wrapper code generation, since from the available options it seemed to us the fastest way, in terms of the amount of development effort required, to achieve the desired Python wrapping of the PCSIM object-oriented framework. One of our goals for the integration of PCSIM with Python was to simplify and support a hybrid modeling approach by enabling the user to implement extensions of the PCSIM object-oriented framework in Python and/or C++, while not having to bother with details regarding the interoperability between these two programming languages.

The excellent support of Boost.Python for advanced C++ concepts and appropriate mapping of corresponding idioms between the two languages allowed us to expose the complete PCSIM API, currently ≈ 300 classes, to Python in a non-intrusive way. This means that the fact that the PCSIM API is to be exposed to Python does not impose any changes at the C++ level nor does it put any constraints on its design. Furthermore the compilation of the `libpcsim` library itself does not depend on any Python library or wrapping code.

BIDIRECTIONAL INTERFACE AND HYBRID MODEL DEFINITION

One of the features of Boost.Python enabling the hybrid approach is the ability to derive Python classes from the wrapped interface classes, and override the virtual functions. Hence, such custom Python class methods can be called from within C++ and thus allow an integration of Python code into the PCSIM C++ code. A similar bidirectional interface has been implemented between Python and NEURON (Hines et al., 2009), where Python can issue commands towards NEURON, but also Python code can be called and executed from within NEURON in an active Hoc session³⁰. In PCSIM the two-way interaction between Python and C++ enables user customizations to be coded in pure Python, and then plugged into the PCSIM C++ framework. This brings additional flexibility and freedom to the user, meaning that he can first do fast implementations in Python, e.g. extensions to the network construction interface (see Network Construction), in the prototyping phase, and afterwards the implementation can be ported to C++ to gain maximum performance.

The ability to define PCSIM network elements in Python opens a possibility for a seamless Python-C++ integration also during the simulation, not only in the network construction stage. The example described in the Section "Custom Network Elements" shows that network elements can be implemented in Python, by using vectorized

techniques employing the highly efficient numerical Python package `numpy` (which is implemented in C). This adds flexibility, since the equations describing the element can be changed quickly without any necessary compilation while not sacrificing performance, since by using `numpy` vectors, the integration algorithm is broken down in elementary vector operations thus avoiding any loops within Python that could be detrimental for the performance.

This approach seems also to be advantageous when one wants to implement network elements that have some abstract processing logic, e.g. signal processing filters, machine learning algorithms or similar. In this case one can utilize a large set of available C++ libraries that have Python bindings, for an efficient implementation, and handle in Python the transfer of data from the input ports of the network element to the input methods of the library, and from the output of the library to the output ports of the network element.

The possibility to implement PCSIM network elements in pure Python offers a convenient way to achieve run-time interoperability between PCSIM and other neural network simulators (Cannon et al., 2007), provided that the simulator has a Python interface, allows control of the simulation process at individual time steps, and has the possibility to write input and read output data during the simulation at each time step. As shown in the example in the Section "Custom Network Elements", we have successfully implemented interoperability with the Brian simulator, which possesses the aforementioned capabilities. One interesting further application of this interoperability could be a distributed simulation of a large neural network where the sub-networks on each node are implemented with the Brian simulator, and the parallel communication is handled by PCSIM's communication system. Another possible approach of using Python as a glue language to achieve simulator interoperability is to setup a Python script as a top-level coordinator of a step-by-step simultaneous execution of two simulators, where the necessary data transfer between the simulators is realized through intermediate Python data structures (Ray and Bhalla, 2008).

HIGH-LEVEL WRAPPING SPECIFICATION AND EXTENSIBILITY

Since the interface of PCSIM has a fine granular structure, composed of many decoupled classes (≈ 300) this implies that there are many classes to be wrapped and exposed to Python. It would simply be impossible to manually manage all the necessary Boost.Python wrapper code. Furthermore, the possibility of adding extensions to the interface puts additional constraints to the wrapping approach to be robust enough to work for the extension classes too, without any significant intervention from the user. Nevertheless, by exploiting the powerful interface generator tool Py++ the wrapping of such a large number of classes is rendered feasible³¹. We were able to specify high-level generic rules within Py++ for the definition of the wrapping of all the classes in the PCSIM API and their sensible extensions. To be precise, the Python program that specifies the rules for the Python interface generation for ≈ 300 classes is about 400 lines of Python code. As these rules apply for the extensions too, the user can easily extend the PCSIM simulator with its own custom C++ classes and compile them in a separate Python extension

²⁸<http://moose.sourceforge.net/>

²⁹<http://www.swig.org>

³⁰Hoc is the native NEURON interpreting language.

³¹The only drawback we encounter is the rather long compile time when recompiling the whole Python interface. This is due to the fact that Boost.Python heavily uses C++ templates.

package, which can be used together with the main `pypcsim` package (the tool support for this is included in PCSIM). This was made possible by the Boost.Python and Py++ support for cross-module inheritance relationships and component-based development (see “Extending PCSIM Using C++”).

To summarize, by the easy extensibility of its interface both in Python and C++, PCSIM enables the modelers to *think hybrid* when developing their models (Abrahams and Grosse-Kunstleve, 2003).

PYTHON AS A SCRIPTING ENVIRONMENT

Providing a Python interface to a neural simulator increases its versatility and consequently the productivity of the modelers in many ways. The object oriented design of the language, its expressive and clean syntax, allows the modeler to focus on the high-level logic of the model instead of struggling with the intricacies and the nuts and bolts of the programming language. Furthermore, there is a growing number of general scientific and specific computational neuroscience software tools available as Python packages, for numerical calculations, scientific functions, plotting, saving data to files, parallel computing etc. We have used several scientific Python packages to enhance PCSIM with useful utilities on top of its basic interface. As we have illustrated through a simple example in the Section “PCSIM Add-Ons Implemented in Python”, in combination with such Python packages PCSIM can be used as the main component of a Python-based neural simulation environment where all steps within a neural model development life-cycle, from the specification of the model and performing the simulations, to storage of simulation output data, data analysis and visualization can be performed. Overall, the integration of PCSIM with Python

added additional valuable facilities to the user, turning PCSIM into a full-fledged neural simulation environment.

PCSIM RESOURCES

Many resources for PCSIM can be found at its web page³². The web page contains a user manual, examples, installation instructions, complete class reference documentation and the complete material for the tutorial that was given at the FIAS Theoretical Neuroscience and Complex Systems summer school held in Frankfurt, Germany in August, 2008. The users can discuss topics and pose questions concerning usage and installation of PCSIM on the *pcsim-users* mailing list on Sourceforge³³ where the PCSIM development project is hosted. In the future, the user manual will continuously undergo extensions and revisions to better organize the content and to include additional topics and more elaborate information about the PCSIM concepts and constructs. Additional examples covering various PCSIM features will also be made available on the web site.

ACKNOWLEDGMENTS

We would like to thank Eilif Muller and Andrew P. Davison for helpful discussions, and the two reviewers for providing valuable suggestions that helped improving the manuscript. Written under partial support of the Austrian Science Fund FWF, project #S9102-N04, as well as project #FP6-015879 (FACETS, <http://facets.kip.uni-heidelberg.de>) and #216593 (SECO, <http://www.seco-project.eu>) of the European Union.

³²<http://www.igi.tugraz.at/pcsim>

³³<http://www.sourceforge.net/projects/pcsim>

REFERENCES

- Abrahams, D., and Grosse-Kunstleve, R. W. (2003). Building hybrid systems with Boost.Python. *C/C++ Users J.* 21, 29–36.
- Beazley, D. (2003). Automated scientific software scripting with SWIG. *Future Generat. Comput. Syst.* 19, 599–609.
- Bower, J. M., and Beeman, D. (1998). The Book of GENESIS (2nd ed.): Exploring Realistic Neural Models With the GENeral NEural Simulation System. New York, Springer-Verlag New York, Inc.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Jr., Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., Boustani, S. E., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398.
- Buzas, P., Kovacs, K., Ferecsko, A. S., Budd, J. M. L., Eysel, U. T., and Kisvarday, Z. F. (2006). Model-based analysis of excitatory lateral connections in the visual cortex. *J. Comp. Neurol.* 499, 861–881.
- Cannon, R., Gewaltig, M.-O., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., and Schutter, E. D. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.
- Carnevale, N. T., and Hines, M. L. (2006). The NEURON Book. New York, Cambridge University Press.
- Dalcín, L., Paz, R., Storti, M., and D’Elia, J. (2008). Mpi for python: performance improvements and mpi-2 extensions. *J. Parallel Distrib. Comput.* 68, 655–662.
- Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2, 11.
- Drewes, R. (2005). Modeling the brain with NCS and brainlab. *Linux Journal* 2005, 2.
- Eppler, J. M., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2008). Pynest: a convenient interface to the nest simulator. *Front. Neuroinform.* 2, 12.
- Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430.
- Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Front. Neuroinform.* 2, 5.
- Hauesler, S., and Maass, W. (2007). A statistical analysis of information-processing properties of lamina-specific cortical microcircuit models. *Cereb. Cortex* 17, 149–162.
- Hammarlund, P., and Ekeberg, O. (1998). Large neural network simulations on multiple hardware platforms. *J. Comput. Neurosci.* 5, 443–459.
- Hines, M., Davison, A. P., and Muller, E. (2009). Neuron and python. *Front. Neuroinform.* 3, 13.
- Hines, M. L., and Carnevale, N. T. (1997). The neuron simulation environment. *Neural Comput.* 9, 1179–1209.
- Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). ModelDB: a database to support computational neuroscience. *J. Comput. Neurosci.* 17, 7–11.
- Hunter, J. D. (2007). Matplotlib: a 2d graphics environment. *Comput. Sci. Eng.* 9, 90–95.
- Izhikevich, E. (2004). Which model to use for cortical spiking neurons? *IEEE Trans. Neural Netw.* 15, 1063–1070.
- Maass, W., Natschlager, T., and Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput.* 14, 2531–2560.
- Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.
- Natschlager, T., Markram, H., and Maass, W. (2003). Computer models and analysis tools for neural microcircuits. In *Neuroscience Databases. A Practical Guide*, R. Kötter, ed. (Boston, Kluwer Academic Publishers), Ch. 9, pp. 123–138.

- Obermayer, K., and Blasdel, G. G. (1993). Geometry of orientation and ocular dominance columns in monkey striate cortex. *J. Neurosci.* 13, 4114–4129.
- Oliphant, T. E. (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20.
- Pérez, F., and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Comput. Sci. Eng.* 9, 21–29.
- Plesser, H., Eppler, J., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Lect. Notes Comput. Sci.* 4641, 672–681.
- Ray, S., and Bhalla, U. S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2, 6.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 14 September 2008; paper pending published: 21 October 2008; accepted: 21 April 2009; published online: 27 May 2009.
Citation: Pecevski D, Natschläger T and Schuch K (2009) PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Front. Neuroinform.* (2009) 3:11. doi: 10.3389/neuro.11.011.2009
Copyright © 2009 Pecevski, Natschläger and Schuch. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



OpenElectrophy: an electrophysiological data- and analysis-sharing framework

Samuel Garcia* and Nicolas Fourcaud-Trocmé

Neurosciences Sensorielles Comportement Cognition, CNRS – UMR5020 – Université Claude Bernard Lyon 1, Lyon, France

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Robert Oostenveld, Cognition and
Behaviour Centre for Cognitive
Neuroimaging, The Netherlands
Elif Muller, Brain Mind Institute, EPFL,
Switzerland

*Correspondence:

Samuel Garcia, Laboratoire de
Neurosciences Sensorielles
Comportement Cognition, CNRS –
UMR5020 – Université Claude Bernard
Lyon 1, Equipe logistique et technique,
50 Avenue Tony Garnier, 69366 Lyon
Cedex 07, France.
e-mail: sgarcia@olfac.univ-lyon1.fr

Progress in experimental tools and design is allowing the acquisition of increasingly large datasets. Storage, manipulation and efficient analyses of such large amounts of data is now a primary issue. We present OpenElectrophy, an electrophysiological data- and analysis-sharing framework developed to fill this niche. It stores all experiment data and meta-data in a single central MySQL database, and provides a graphic user interface to visualize and explore the data, and a library of functions for user analysis scripting in Python. It implements multiple spike-sorting methods, and oscillation detection based on the ridge extraction methods due to Roux et al. (2007). OpenElectrophy is open source and is freely available for download at <http://neuralensemble.org/trac/OpenElectrophy>.

Keywords: python, electrophysiology, analysis, oscillation, spike sorting, database, SQL

INTRODUCTION

Recent developments in electrophysiology experimental techniques have lead to increases in the amount of data produced. It is now common to record continuous signals simultaneously from many electrodes with a sampling rate of 10 kHz or more. This increase in raw data flow has been accompanied by an increase in the complexity of the experimental protocol and the subsequent analyses. Indeed, each experiment is controlled by a large number of parameters that are either set by the experimenter (e.g., according to the stimuli applied or the state of the subject) or constrained by the experimental setup (e.g., electrode properties). These parameters are the meta-data associated with the experiment. A variety of new software aiming to facilitate data storage, exploration and analysis are appearing to help scientists handle such large amounts of data and experimental parameters.

Several commercial software products have been developed to tackle the increasing data management demands of state-of-the-art electrophysiology. However, as such commercial software products have not always evolved as rapidly as the needs of the field, several open source projects have appeared which are developed by the researcher community. Among them are open source software that performs commonly used analysis methods (e.g., averaging, time–frequency analysis) for analyzing magnetoencephalography (MEG) or electroencephalography (EEG) data. These programs generally have a highly developed graphical user interface (GUI). In contrast, in the field of spike sorting, various toolboxes are available, and these toolboxes usually require the researcher to write-specific scripts in order to use the toolbox for a specific set of data. Thus, there are at least two different approaches with regard to purpose and user interface in open source software design. None of the available software or toolboxes addresses the problem of

how to simply and conjointly manipulate experimental data and meta-data.

OpenElectrophy was designed more as a framework for data analysis than a piece of completely frozen analysis software. For example, it is not specific to a given type of electrophysiological signal, and does not directly perform a specific type of analysis at the request of a researcher with a “point-and-click” scheme. Rather, it provides tools to facilitate data storage, exploration and analysis script writing. It gathers the best of the two open source approaches described previously, both in terms of purpose (time–frequency analysis and spike sorting) and in terms of user interface (GUI and toolboxes). In addition, it includes generic tools for conjointly manipulating both experimental data and meta-data. The project’s main philosophy has three parts: first, for each experiment, the data and meta-data are all stored in a single central database. This strategy allows for flexibility in mixing both types of data in the subsequent analyses. Second, it provides a GUI that is useful for exploring the data and detecting events of interest (oscillations or spikes). Third, it contains a library of “methods” (high-level functions) to aid in the writing of analysis scripts, both in the interfacing of these scripts with the database and in the manipulation of the data.

OpenElectrophy was developed through a collaboration of people working on electrophysiological signals, such as extra- or intracellular recordings or EEG signals. In these fields, people are especially interested in detecting and analyzing transient oscillations or neuronal spikes. When this project was started, the conjoint analysis of both spikes and oscillations could not be performed using any available software. Thus, one of the main goals of OpenElectrophy was to provide a complete and convenient way to detect spikes and transient oscillations, store all of the detected events in the same

database as the original data, and then manipulate them conjointly. Subsequent analyses could then include simultaneously detected events, raw data and meta-data. We emphasize that OpenElectrophy is one of the few currently available open source tools designed to work simultaneously with spikes and oscillations.

This article presents the design and use of OpenElectrophy. It is organized into five sections. We first compare OpenElectrophy to similar projects and detail the advantages, drawbacks and differences of purpose for each project. Second, we explain how we used the database manager MySQL and the scripting language Python (and its scientific module SciPy) to construct the core architecture of OpenElectrophy. Third, we present the OpenElectrophy work flow and the general way in which it is used. Fourth, we briefly describe the spike and oscillation detection methods that are currently implemented. Last, we present an example of the standard usage of OpenElectrophy to analyze extracellular local field potential (LFP) recordings and obtain information about action potential locking on LFP oscillation.

COMPARISONS WITH OTHER PROJECTS AND THE MAIN GOALS OF OpenElectrophy

Commercial products like Plexon¹, Tucker Davis² or Spike2³ exist for the analysis of electrophysiological signals and are in wide-spread use. We will not go into detail about these software programs, but we will point out that despite their high quality GUIs, support and continuous development, they use proprietary languages, which present barriers for code sharing and reuse, and which have limited uptake of tools being developed by the scientific computing community compared to languages such as Python. Moreover, the file format specifications are generally not available, making long-term storage or sharing of data problematic since anyone who wants to access the data needs the right software. To deal with this issue, Neuroshare⁴ was created in an attempt to provide standardized libraries that can access proprietary file formats. However, Neuroshare provides only reading functionality, the code is not open source, and libraries are available only for the Windows 32 platform.

The various open source projects belong to two families: software and toolboxes for analyzing EEG or MEG data, and software for spike sorting. Few projects mix spike and spectral analyses.

In the EEG/MEG family, visible projects include EEGLab⁵, FieldTrip⁶ and SPM⁷ (EEG sub-package). These three projects are all written with MATLAB, have a comprehensive GUI for non-programmer users, use a homemade data format based on MATLAB structures and store data in the MATLAB file format. Their main features include analyses of event-related potentials, time–frequency analyses, independent component analyses (ICA) and 3D plotting methods. They also implement methods for source detection.

In the spike-sorting software family, most projects can be separated into two classes. The first class includes tools dedicated

solely to spike sorting: WavClus⁸, Mclus⁹, Spike-O-Matic¹⁰ and Klustakwik¹¹. They do not perform any data management, but can load one or several data formats and store the results (detected spikes) in custom file formats. They generally provide only basic GUIs, except for Klustakwik, which provides no GUI. WavClus and Mclus are written in MATLAB; Spike-O-Matic is written with R; Klustakwik is a C++ library. In general, these projects were written to introduce a new spike-sorting method: WavClus is based on superparamagnetic clustering (SPC) and wavelet projection, Spike-O-Matic is based on Monte Carlo Markov Chain methods, and Mclus and Klustakwik are based on a classification expectation maximization algorithm. The second class of projects is dedicated to the analysis of spike trains: Spike Train Analysis Toolkit¹², NeuroTools¹³, and Pandora¹⁴. These three projects are collections of scripts for analyzing spike trains after spike sorting has already been completed. The Spike Train Analysis Toolkit is based on MATLAB and provides functions related to entropy and information theory. NeuroTools is written in Python and provides functions for analyzing simulated datasets generated from models. Pandora is MATLAB-based; it is one of the few projects that uses the concept of a database for managing datasets, but it uses a custom-built database system written in MATLAB, as opposed to employing an established database system such as MySQL.

Finally, we must mention three projects that mix spike firing analyses and spectral analyses on an LFP signal: FIND¹⁵, MEA-tools¹⁶ and Chronux¹⁷. These projects were all written with the same primary goal as that of OpenElectrophy: to function as a framework for sharing analyses. They provide most of the standard analysis tools and others developed more recently, all written in MATLAB, but they include no database framework or meta-data management.

OpenElectrophy was written for several reasons:

- To have a project that is useful for all types of electrophysiological signals and experiments that mix time–frequency studies, spike-sorting and spike train analyses, and that uses pre-existing scripts or toolboxes whenever possible.
- To have a project that includes various spike-sorting methods and allows the user to choose which one best fits his data.
- To have a project that directly manages data and meta-data through a MySQL database that allows for sustainable data storage. Most previously developed projects use custom-built and language-dependent file formats. MySQL is open source and well established; datasets can be accessed with many scripting languages (Python, MATLAB, Excel, R, Statistica) and with most of the traditional software used in a neuroscience laboratory.

¹<http://www.plexoninc.com/>

²<http://www.tdt.com/>

³<http://www.ced.co.uk/>

⁴<http://neuroshare.org/>

⁵<http://scn.ucsd.edu/eeglab>

⁶<http://www.ru.nl/neuroimaging/fieldtrip>

⁷<http://www.fil.ion.ucl.ac.uk/spm/>

⁸http://www.vis.caltech.edu/~rodri/Wave_clus/Wave_clus_home.htm

⁹<http://www.neuroinf.org/lists/comp-neuro/Archive/2000/0065.html>

¹⁰<http://www.biomedicale.univ-paris5.fr/SpikeOMatic>

¹¹<http://klustakwik.sourceforge.net/>

¹²<http://neuroanalysis.org/toolkit/>

¹³<http://neuralensemble.org/trac/NeuroTools>

¹⁴<http://userwww.service.emory.edu/~cgunay/pandora/>

¹⁵<http://find.bccn.uni-freiburg.de/>

¹⁶<http://material.brainworks.uni-freiburg.de/research/meatools/>

¹⁷<http://chronux.org/>

- To have a free project that relies only on other open source projects. Most previously developed projects are based on MATLAB; it is quite contradictory to have an open source project that forces the community to pay a license to a third party (Matworks) while free alternatives exist (Python and its scientific module SciPy).
- To have the capability to quickly design a high quality GUI. This goal is achievable with PyQt, a Python wrapper for the modern graphics library Qt. This is in contrast to MATLAB, which possesses a less appropriate object-oriented programming approach and GUI toolkit.

We must emphasize that OpenElectrophy is neither a simple GUI interface nor a library of functions, but rather a combination of both, depending on what needs to be done with the data. Hence, the GUI is used mainly for data storage, visualization, and exploration; it also guides the initial analysis steps, such as the detection of events of interest (e.g., spikes or oscillations). Script writing is necessary to perform the specific analyses that are needed by the researcher. In order to make analysis writing as simple and as flexible as possible, OpenElectrophy provides Python methods to appropriately query the database and manipulate the electrophysiological data.

Finally, we must point out that at its current development stage, OpenElectrophy is primarily designed for the LFP and spike community rather than the multi-channel EEG community. For example, it does not currently include any advanced visualization tools, such as 3D scalp plot or source localization techniques.

TECHNICAL CHOICES

The development of OpenElectrophy is based on two technologies: MySQL, an open source database server, and SciPy, the Python scientific module. The GUI was implemented with PyQt4. We chose to rely on these open sources projects because they are widely used and have strong support communities that ensure free availability and reliability. Moreover, they provide efficient interfaces with other scripting or compiled languages (e.g., MATLAB, R, C/C++, Statistica, Excel). These interfaces are important to allow for interaction with previously developed methods from other open source projects. Lastly, Python is an object-oriented language that is well adapted to developing long-term projects with highly structured designs, thus facilitating collaboration between developers and users.

In this section, we present a summary of the core architecture of OpenElectrophy. In particular, we show how MySQL and Python are used to help fulfill OpenElectrophy's goals.

MySQL

Briefly, as a reminder, it should be stated that the intrinsic concept of a database system is a collection of tables. Each table has a collection of fields of different types. Tables are linked to one another by indexes or keys. Putting data into a database is equivalent to splitting it up in an atomic way and organizing it into different tables. The logical or hierarchical organization between tables is not known a priori, but is formed while exploring the data, as opposed to file systems, which are organized into directories and sub-directories with a fixed organization. Thus, it is possible to have multiple views of the

same database. This mechanism, while apparently basic, proves to be flexible and efficient. To work with this system, the user must learn structured query language (SQL). This language permits the user to reconstruct, filter and sort the data. The user can also add fields or tables at a later point without affecting previous work.

A crucial point is the design of the table's schema: the list of tables, and their contents and links. The idea was to design a generic core schema that can deal as naturally as possible with any electrophysiological dataset. In electrophysiology, people manipulate two main types of signals: continuous signals, which come from electrodes, and discrete or stepwise signals such as triggers or time events, which come from the context of the data acquisition (e.g., stimulus, subject states). Based on this requirement, the core schema that was chosen for OpenElectrophy is detailed in **Figure 1**. The three central tables are *trial*, *epoch* and *electrode*. The table *trial* includes a coherent recording of continuous or discrete events. The table *electrode* holds the raw continuous signals from each physical electrode. The table *epoch* manages all discrete events: trigger times, periods of stimulation, animal states or event markers. These three tables can accommodate a generic electrophysiological recording. The tables *spike*, *spiketrain* and *cell* were then added to manage neuron spike discharge. The table *oscillation* manages transient oscillatory events in the LFP.

This schema has already been proven to be flexible enough to fit several types of experimental setups, such as one-cell intracellular recordings, extracellular multi-electrode recordings, short- or long-protocol recordings, LFP studies, multiple repetitions of stimuli, and animal behavior data. For each experiment, this design is at the core of the data management; however, each new study usually requires a short extension of the table schema. Extra fields commonly need to be added to the original tables, and new tables must sometimes be added to address new concepts such as animal position or heartbeat. The versatility of the database allows for this kind of customization without interfering with the core of OpenElectrophy.

Today, many data manipulation tools include an SQL interface; MySQL is a kind of "universal" data format that does not depend on a particular language. Another advantage of this type of data storage is the MySQL client/server design. Indeed, all of the data is collected on a single server that is simultaneously accessible by many users of OpenElectrophy (or other tools). This access does not need to be local, such that collaborations between labs working on the same dataset are possible. We note here that transferring large sets of raw data over the Internet can take a prohibitively long time, but it is generally not a problem to transfer only discrete events such as spike times (also present in the database), which can be done by using appropriate SQL queries. Another benefit of the database scheme is that each time someone makes a new entry into the database (e.g., raw data or meta-data, spikes, oscillations, a new field with a specific type of information), that information is immediately available to all of his collaborators. Lastly, MySQL offers many efficient backup capabilities (from single global or partial transfers of the whole database to continuous incremental saves) to secure the data or make them portable.

PYTHON

Python is a high-level object-oriented programming language. It is available for a wide range of platforms and comes with a large

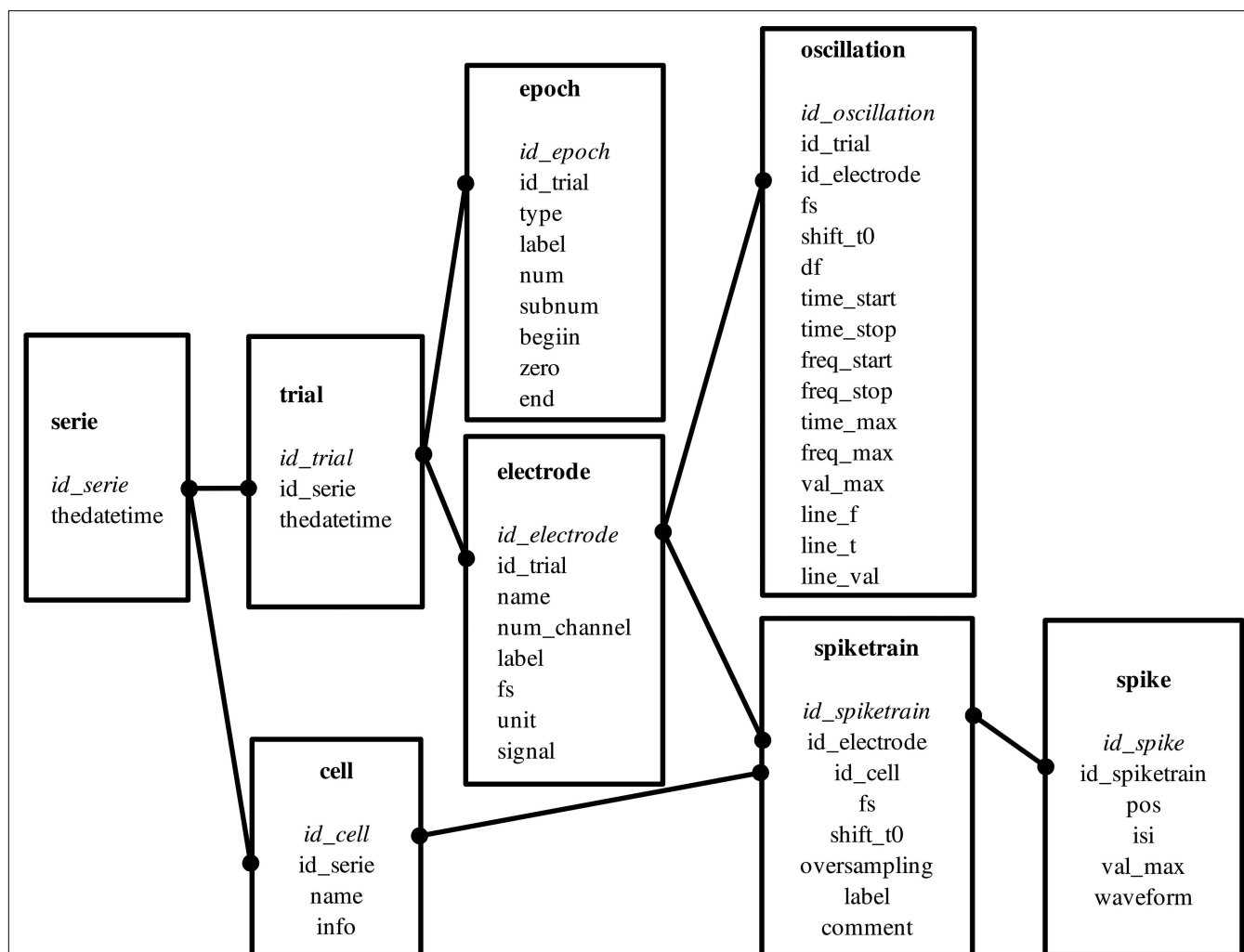


FIGURE 1 | Database schema. This is a classical relational design. Each frame corresponds to a table that holds all of the properties of an element in its fields. For example, the table spike holds for each spike its own index (*id_spike*), the index of the spike train it belongs to (*id_spiketrain*), its position (*pos*), the maximum amplitude (*val_max*) and its raw waveform (*waveform*). All of the tables and fields are natively generated by OpenElectrophy; the schema is flexible and extensible to accommodate specific needs. The core of the schema includes the trial, electrode and epoch tables. A trial is a combination of several simultaneous coherent recordings. These recordings are continuous or discrete,

and are stored in the electrode or epoch tables, respectively. Additional tables are as follows. The series table, which gathers a set of trials (e.g., those recorded in the same location). The spike table contains all detected spikes and their positions and shapes. The spikes are grouped according to their spike train (there may be many spike trains per electrode). The cell table groups spike trains that were recorded from the same cell but in different trials; thus, the cell table groups them relationally. Finally, the oscillation table contains all of the information related to transient oscillatory events (see Section “Oscillation Extraction”).

collection of libraries (modules). For scientists and engineers dealing with computing, one of the most interesting Python modules is SciPy. This module provides *N*-dimensional array manipulation with the NumPy module and a fast implementation of an extensive set of scientific algorithms, such as filtering, statistics, interpolation, and linear algebra. For neuroscience studies that generate increasingly large datasets, Python is the equivalent of a Swiss army knife.

Using MySQL to explore and select data is efficient, but creating a table schema and inserting or modifying data is a repetitive and tedious task with pure SQL. Object-relational mapping (ORM) is a technical programming method that converts between a database and a Python object. Thus, OpenElectrophy incorporates a custom-built

ORM to simplify read/write database access (see Section “A Typical Use-case” for an example of its use). This SQL mapper, a Python class included in OpenElectrophy, allows the user to declare a table structure with field names and types with only a few lines of code. Each instance of this class can directly map onto all of the fields of a table entry. Each SQL field becomes a member of the class instance. There are two methods (*load_from_db* and *save_to_db*) for automatically loading or saving all fields from the database without writing any SQL. The conversion from Python types to MySQL types is straightforward for basic types (int, float, str). For *numpy.array* (the basic type for *N*-dimensional arrays of the SciPy module), the conversion is automatically done by OpenElectrophy in three fields: one blob field for the buffer of the array, one field for the dimensions

and one for the array element type. Thus, the user can store vectors or matrices in MySQL, which is normally not allowed. Each MySQL table corresponds to a specific Python class that inherits the SQL mapper base class and that implements methods that are specific to the table content. For example, the *Electrode* class can query the *electrode* table and implements different plotting methods (raw or filtered signals and time–frequency maps).

As a further example, the *SpikeTrain* class offers methods for reconstructing a spike train in different ways, such as a vector of time stamps, as sample indexes, as intervals or in Boolean form for information theoretical methods. Of course, all of these methods directly query the *spike* table, collect all individual spikes that are linked with an element of the *spiketrain* table and reorganize the results into the appropriate format. Three plotting methods are also available: raster, large dots superposed on the electrode signal and cumulative waveform.

Currently, only tables and fields that are present in the schema in **Figure 1** are loaded by OpenElectrophy classes. Thus, additional fields or tables that are created for specific experiments must be accessed via SQL queries directly with a Python script. Alternatively, existing OpenElectrophy Python classes can be manually overloaded to take into account the new elements.

WORK FLOW

In this section, we describe the general workflow of OpenElectrophy as summarized as a series of steps in **Figure 2**. Each step of the workflow is discussed in detail in turn below.

DATA INTEGRATION

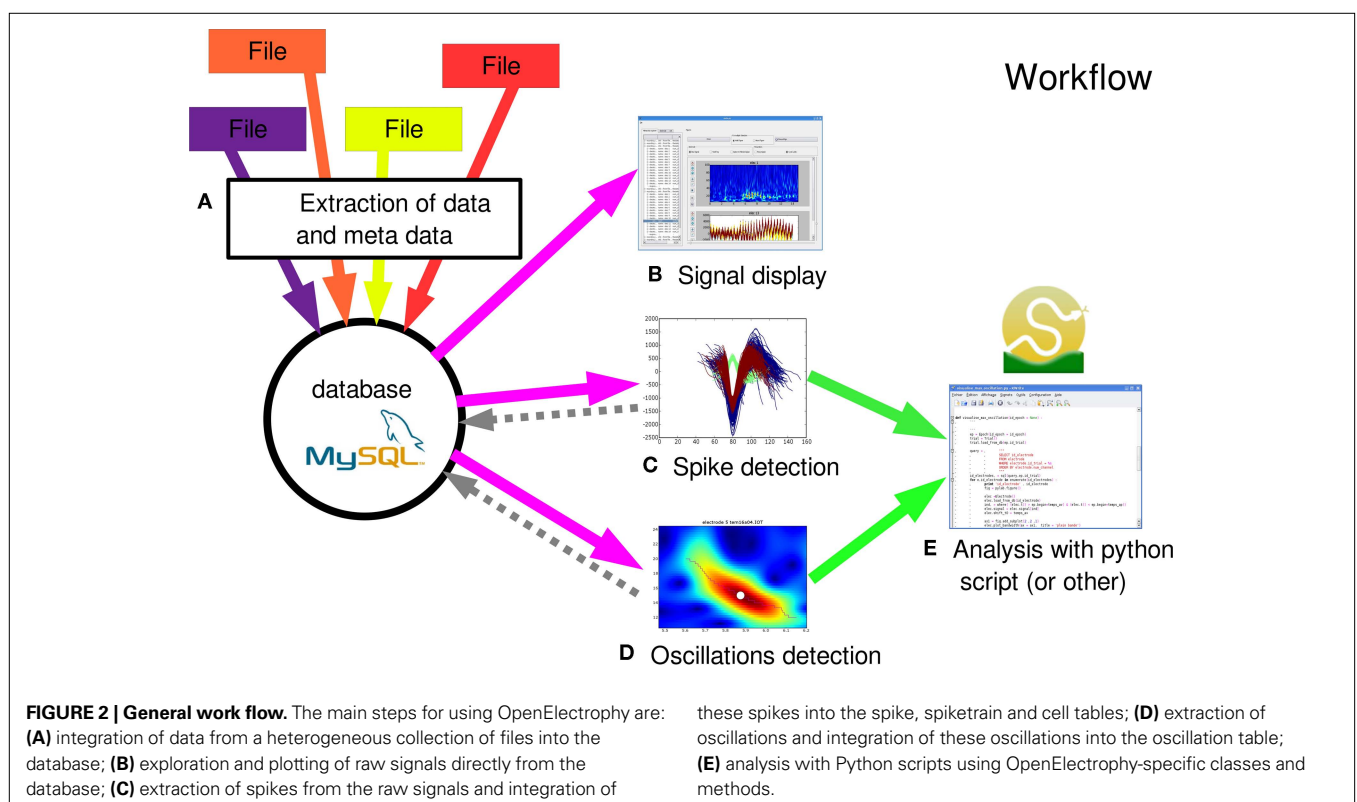
The first step of OpenElectrophy workflow involves importing data into the database. The idea is to integrate all available

information into the database, including data (e.g., signal, triggers, events) and meta-data (e.g., protocol context, date, time). In so doing, during the analysis, the user no longer has to work with a heterogeneous collection of files; instead, the user works directly with the database system. OpenElectrophy is already able to integrate into the database data that is stored in different file formats, including ASCII, raw binary, Elan, TDT, Elphy, and Micromed. In the near future, many additional data formats will be incorporated. The end user can go deeper into data integration by writing new scripts that not only incorporate neural data but also setup-specific meta-data. For instance, stimulus generation software often provides lists of stimuli and context information in a clear file format. These files can be parsed and integrated during the integration of neural data. Finally, note that the database can also be directly accessed and filled or edited with a basic MySQL client editor.

At this stage, it is possible to explore the database using different hierarchical tree views and to plot raw signals (bandwidth or filtered) or wavelet-based time–frequency maps.

SPIKE AND OSCILLATION DETECTION

The next major step is the extraction of the phenomena of interest: spikes and transient oscillatory events. In these two cases, a graphical interface helps in searching for parameters that allow for good detection. This step is crucial for subsequent stages of the analysis. There are two possible methods for detection: individual detection, which is done signal-by-signal, or bulk detection, which is done by applying the same parameters to an ensemble of signals targeted by an SQL query that is directly written in the OpenElectrophy GUI.



With regard to spike detection, methods used in OpenElectrophy will be detailed in Section “Spike Extraction”. The central idea of the framework is to store individual spike events in the MySQL *spike* table and group them using the *spiketrain* and *cell* tables. All studies on spike discharge will deal directly with these three tables using SQL queries, but will benefit from all of the tables when working with protocol information and context meta-data.

The oscillation detection method is based on a new approach detailed in Section “Oscillation Extraction”. The result is a list of oscillations for each signal. In this case, each oscillation is stored in the *oscillation* table. Thus, for studies on multi-frequency oscillatory regimes (e.g., theta, gamma, and beta bands), the analysis is computed directly in this table, although it again also benefits from the data stored in all other tables.

SQL FILTERING

The ability of SQL to dynamically provide different views of the database is heavily exploited during analysis. Consequently, a basic knowledge of this language is required. Data analysis can be summarized as applying an algorithm or a statistical measure or plotting synthetic views of a subset of the data. The traditional method for analyzing data is to manipulate and aggregate the data by hand, creating a text list for each condition or factor or constructing many synthetic tables with an external knowledge of protocol factors. With SQL, this tedious work is done directly. With a few lines of code, values of interest can be rapidly and efficiently aggregated. Using SQL, there is no need to store lists, sublists or sub-sublists of data; the user only needs to store the queries and use them for each analysis. Overall, the user must manage queries that are useful for selecting data according to context and factor fields; then, the user must write new analyses in Python that can be applied to a subset of the data that is extracted with these SQL queries.

ANALYSIS

Analysis is the final stage of the OpenElectrophy workflow, which transforms the now pre-processed data into meaningful results. The OpenElectrophy framework does not provide ready-made “point-and-click” analyses for obtaining a given result. Rather, it is necessary to write scripts in Python to perform statistical tests or other specific analyses. Here, the management of the data in a central database simplifies the selection of the data to analyze (see Section “SQL Filtering”), and the Python classes provided by OpenElectrophy ease the manipulation of the data to match a given analysis. Additionally, the Python SciPy module provides many standard and high-level analysis tools, and the Matplotlib module offers extensive 2D plotting methods.

Writing analysis scripts can seem difficult for researchers not familiar with programming, but the power and flexibility of this approach is quickly preferred over the restrictive convenience of a GUI. For example, to our knowledge none of the available software for doing spike analysis provides a GUI as an alternative to analysis scripting. Starting with simple script examples is usually sufficient to allow beginners to compose very sophisticated analyses. Thus, OpenElectrophy does not constrain data analysis with a fixed GUI, but allows for the use of user programmable scripts.

As already mentioned, a major advantage of using the Python scripting language is its ability to interface with other languages.

Packages like Mlabwrap¹⁸, rpy¹⁹, cython²⁰ or SciPy.weave²¹ enable to use pre-existent code from MATLAB, R, or C/C++. Employing these tools, the list of external modules that can be linked to OpenElectrophy to help write analysis scripts is long: the International Neuroinformatics Coordinating Facility provides a list of tools available for studying neural data²². In particular, OpenElectrophy, as a framework for managing data, would likely complement recent Python-based approaches to neural data studies, such as PyEntropy (Ince et al., 2009) for information theory and PyMVA (Hanke et al., 2009) for machine learning.

Details on how to use OpenElectrophy classes for scripting are available on the OpenElectrophy wiki page²³.

DETAILS OF EXTRACTION METHODS

SPIKE EXTRACTION

One crucial part of multi-extracellular electrophysiological recordings is spike detection and sorting. All subsequent interpretations rely on the accuracy of these steps. Many approaches to this challenge already exist. Some systems use in-line, real-time, and unsupervised spike sorting, while others, including OpenElectrophy, prefer off-line and semi-automatic spike sorting. There is no perfect method; a compromise must exist between fully automatic and fully supervised processing. Several numerical algorithms for spike sorting have been published. Processing can be separated into four steps: filtering, detection, decomposition (or projection) and clustering. The literature on projection and clustering is extensive (Lewicki, 1998; Pouzat et al., 2004; Quiroga et al., 2004; Wood et al., 2006). Less effort has been put into filtering and detection. These two steps cannot be neglected, however, as bad filtering directly influences spike shape, and can thereby generate strange results even with a good clustering algorithm. To overcome these difficulties, OpenElectrophy is designed in a modular way and offers several methods for each step. Thus, spike extraction can be tuned for many experimental setups, and new methods can be added to the framework by external contributors.

At the moment, the implemented algorithms are:

- Filtering: “fast Fourier transform”-based filter, Bessel, Butterworth, median sliding filter for removing slow components.
- Detection: threshold on maximum amplitude.
- Projection: principal component (PCA) of the spike shape, independent component (ICA), raw waveform shape. Wavelet projection will be implemented soon. PCA and ICA projections are done with the Modular toolkit for Data Processing, a machine learning package for Python (Zito et al., 2008).
- Clustering: “k-means” method and “SPC” (Blatt et al., 1996).

A future step for OpenElectrophy will be to incorporate additional spike-sorting methods developed in other open source

¹⁸<http://mlabwrap.sourceforge.net/>

¹⁹<http://rpy.sourceforge.net/>

²⁰<http://www.cython.org/>

²¹<http://www.scipy.org/Weave>

²²<http://software.incf.org>

²³<http://neuralensemble.org/trac/OpenElectrophy/wiki/OEScriptTutorial>

projects (see Section “Comparisons with Other Projects and the Main Goals of OpenElectrophy”).

OSCILLATION EXTRACTION

The detection of non-stationary oscillations in LFPs by OpenElectrophy is based on a new method described by S. Roux (Roux et al., 2007). Classical studies on oscillatory phenomena have used time–frequency Morlet scalograms. The Roux method goes further to use the scalogram to extract individual oscillations with a ridge extraction method. This method is useful when signals have oscillations in different frequency bands, when oscillation frequencies shift as a function of time or when there is no a priori knowledge of the signal.

The main steps in the processing are:

- computing the Morlet scalogram
- choosing a significant threshold for the detection of oscillations
- detecting local maxima above this threshold in the frequency bands of interest
- extracting ridges point by point, starting from the maximum and continuing until the threshold is reached.

Finally, each ridge is a time–frequency line that describes a trajectory in time and frequency point by point; it is a complex number. The complex modulus estimates the energy envelope of the oscillation, and the angle from the real axis estimates its instantaneous phase.

From each extracted line, the oscillatory epoch duration and onset can be estimated, as well as the frequency, phase and amplitude evolution as a function of time. In short, this method allows for the extraction of all oscillation parameters.

This approach introduces a more intuitive and more accurate method to analyze non-stationary local fields with oscillations. Statistics can be applied, for example to the duration or frequency shift, as analyses become quantitative.

A Python class associated with a MySQL table manages all oscillations. All parameters are stored in the appropriate fields; the time–frequency line itself is directly stored as a “numpy.array”.

A TYPICAL USE CASE

In this section, we will present an example of how OpenElectrophy might typically be used and demonstrate its GUI. We consider an experiment in which the extracellular LFP was recorded in the piriform cortex of an anesthetized rat. The aim of this experiment was to study the relationship between local field oscillatory activity (network level) and single unit activity (neuron level) (Litaudon et al., 2008).

The raw signal was first saved into the database as previously explained. The next step was then to extract oscillatory events. Upon completion of the extraction, the GUI is as shown in **Figure 3A**. On the left of the screen are all of the parameters that are used for detection; these parameters can be modified by the user and saved for later use. These parameters cover the time/frequency space and the precision used for the detection, as well as the threshold above which oscillations are detected (an absolute level or relative to a reference period in the same signal); in addition, some of these parameters

are used to remove overlapping or unwanted short oscillations. On the right of the screen, the list of oscillations detected for this electrode is shown. Below, their trajectories are plotted superimposed both on the electrode Morlet scalogram and on the electrode raw signal (lower right of the screen). When the user is satisfied with the results, he can save it to the database. Note that in this example, the detection of oscillations was done for a single electrode. Another GUI can be used to detect oscillations for many electrodes simultaneously. In this case, the GUI presents the same parameters as for the single electrode GUI, but with an additional window in which the user may provide the SQL query to select the electrodes for detection.

The next step was the detection of spikes in the same signal. The GUI shown in **Figure 3B** presents five tabs corresponding to the four steps used in the spike detection (see Section “MySQL”) and a fifth for the database options (which summarizes the results and, in the case of multiple detected spike trains, allows the user to choose which results should be saved to the database). At any step, the parameters can be set and saved for later use. Spike detection can be done in its entirety or in a step-by-step fashion, with various plots on each tab dedicated to the intermediate results. Again, this task can be performed for multiple electrodes simultaneously with a similar GUI that includes all tabs (without graphic feedback) and a window to specify the SQL query. A special case is the detection of spikes from the same electrode channel across all trials from a given series. In this case, the signals from all trials are pooled before spike detection, and the resulting spike trains (one for each trial) are linked in the database via the *cell* table, so that it can be documented that they are all associated with the same neuron.

The final step was the analysis of the results, which here consisted of a histogram of spike phases (relative to the oscillations). This analysis has already been implemented in OpenElectrophy, and the user needs only to specify a list of oscillations and a list of spike trains with two SQL queries to obtain the graph in **Figure 3C**. To demonstrate how this process can be done using an external script, we present here the Python code that was used in this analysis:

```
# Initialize result array
phase_spike = empty((0))

# write a query for spike train of interest
# for example, spike trains of electrode 5
query_spiketrain = """
    SELECT spiketrain.id_spiketrain
    FROM spiketrain, electrode
    WHERE electrode.id_
    electrode = spiketrain.id_electrode
    AND electrode.num_channel = 5
    """

# execute query and get id list
list_id_spiketrain, = sql(query_spiketrain)
for id_spiketrain in list_id_spiketrain:
    # Python class implemented by OpenElectrophy
    # that maps one spike train
    sptr = SpikeTrain()
    # method to load spiketrain properties from the
    # database
    sptr.load_from_db(id_spiketrain)
    # method to get spike positions of the spiketrain
    pos_spike = sptr.pos_spike()
```

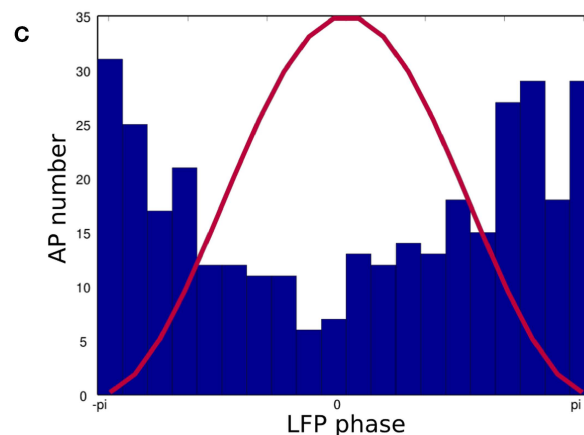
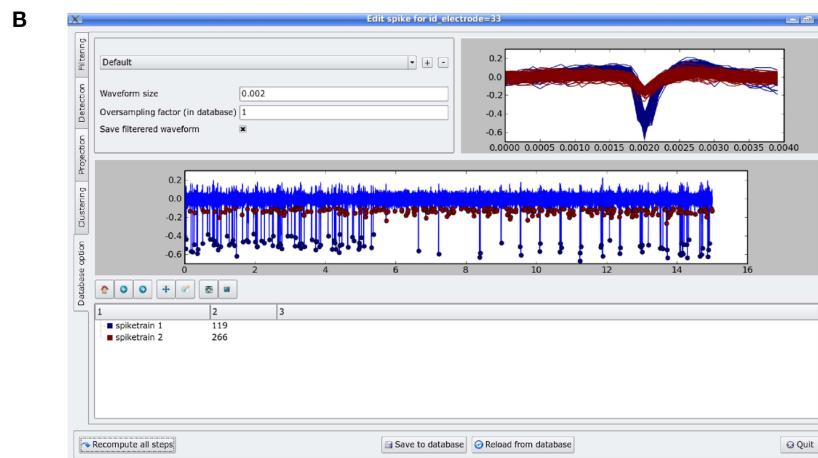
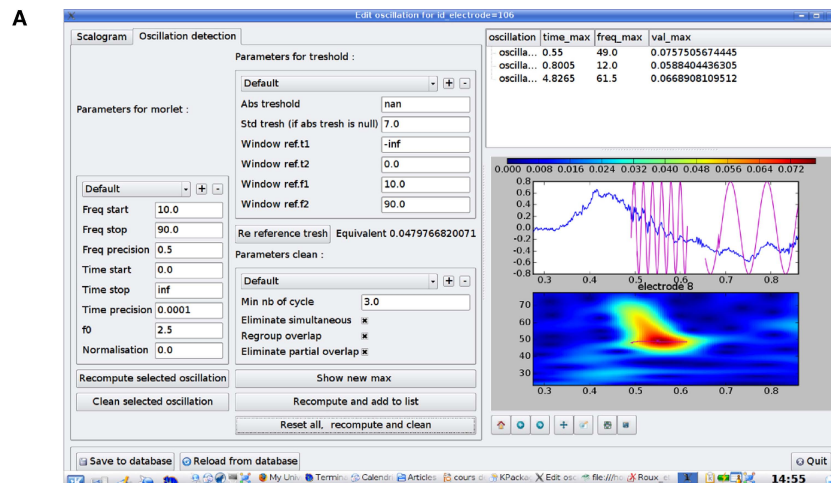



FIGURE 3 | (A) Snapshot of the oscillation detection dialog. On the left side, frames encapsulate different kinds of parameters: for the Morlet scalogram, the threshold definition and “cleaning” the detection. On upper right, there is a list of detected oscillations. On lower right, there is a zoomed picture of one time–frequency line, which represents an oscillatory event, and the relative phase reconstruction superimposed on the raw signal. When the detection is done, the results can be stored in the MySQL database. **(B)** Snapshot of the spike

detection dialog. On the left, there are different tabs corresponding to the different steps of spike extraction: filtering, detection, projection and clustering. The result of a particular detection that can be saved into the database is on the shown tab. **(C)** Example of how spike and oscillatory events can be mixed, showing how a spike train is phase locked on the LFP phase. One oscillation cycle is depicted in red, and a histogram of the phases of spike discharge is shown in blue.

```

# Select oscillations from the same trial as the
# spike train
# and that are in the gamma band
query = """ SELECT oscillation.id_oscillation
            FROM oscillation
            WHERE oscillation.id_trial = %s
            AND oscillation.freq_max > 35
            AND oscillation.freq_max < 100
            """
list_id_oscillation, = sql(query, (sptr.id_trial),
                          Array=True)

for id_oscillation in list_id_oscillation:
    # Python class implemented by
    # OpenElectrophy
    # that maps one oscillation
    osci = Oscillation()
    # method to load the oscillation's
    # properties from the database
    osci.load_from_db(id_oscillation)
    # get phase of spikes that are on the
    # oscillation
    phase_spike = r_[phase_spike, angle(osci.
                                         line_val[setmemberid
                                         (osci.line_t, pos_spike)])]

# Plot the result...

```

CONCLUSION

In summary, we have presented OpenElectrophy, an open source project aimed at facilitating the management and manipulation of electrophysiological data along with experiment meta-data. The key contribution of OpenElectrophy is the framework architecture: MySQL married to Python + SciPy, all of which are reliable, widely used and free tools. We have shown how the use of a MySQL database allows for long-term storage, easy access and sharing of data. In particular, all of the data and meta-data are recorded in a central database and can be combined for further analyses, allowing the user, for example, to fuse electrophysiological and behavioral data. We have also shown how OpenElectrophy uses the Python language to simplify interaction with the database and manipulation of data during the writing of analysis scripts. Another primary feature of OpenElectrophy is the integration of the detection and storage of spikes and transient oscillatory events found in electrophysiological recordings. We note that the CARMEN²⁴ project has recently been created and appears to pursue goals similar to ours, but it is now primarily a repository of diverse methods without much global integration.

The OpenElectrophy project is free and open source, which means that anyone can download, use, modify or extend it and then share his work with the whole user community. It is hosted in a forge with a Trac system²⁵, which offers SVN as a version control system and a wiki for live documentation. A mailing listing for discussion between users and developers is available²⁶.

Like many other free projects, the success of OpenElectrophy depends on the size of the community using it and developing

it. For the moment, OpenElectrophy is a young project and the community is relatively small (about 20 people). Its development has thus mostly involved addressing the needs of this small community. Nonetheless, we hope to have designed the foundations of OpenElectrophy with enough care in terms of flexibility and technological choices such that adapting it to a wider range of needs and use cases would require minimal effort.

At the moment, the OpenElectrophy GUI adequately covers the exploration of data, spike sorting and detection of transient oscillations. The analyses must be computed with Python scripts, which need to be provided by the user. Obviously, these scripts can be written from scratch, but as we already have mentioned, one of the advantages of Python is that it can be interfaced with previously developed analysis toolboxes. Thus, it will be useful in the future to provide, either directly in OpenElectrophy or as script examples (which could be available on the wiki pages for OpenElectrophy), simple ways to interface the data managed by OpenElectrophy with other open source toolboxes, such as the ones presented in this issue, e.g., PyMVPA, PyEntropy or NeuroTools. Additionally, one possible extension would be to write an intuitive GUI for launching some simple analyses in order to make OpenElectrophy more attractive to users who do not write scripts.

Finally, with regard to the more technical aspects of OpenElectrophy, we must mention two future improvements. The first is the integration of a standard ORM such as SQLAlchemy for mapping data to OpenElectrophy objects. At the moment, the SQL mapper is home-made, but it has the advantage of incorporating “numpy.array”. Using SQLAlchemy instead will allow for the direct use of database systems other than MySQL, such as SQLite or PostgreSQL. Second, use of the concept of BLOB streaming²⁷ while using MySQL to read continuous electrode data should be a great improvement. This technique consists in loading BLOB (binary) fields into a stream chunk by chunk. This is a definitive solution for long recordings at high sample rates and solves memory problems.

In the present article, it has been argued that MySQL is a good choice for a data storage architecture, given its powerful features to store, organize, and provide dynamic views of data. However, its socket-based architecture might raise concerns of performance over other scientific binary formats. A simple comparison of read and write performance with the widely used and performant hdf5²⁸ for an array of 10e7 elements shows OpenElectrophy (local MySQL server) has only slight penalties for read (factor of 1.4) and moderate penalties for write (factor of 4). Assuming that in a normal study cycle, we spend more time in reading than writing, we believe that the architectural advantages of MySQL mentioned previously counterbalance the moderate performance penalty, and it remains an attractive alternative to hdf5.

In developing OpenElectrophy, we have endeavored to follow the fundamental philosophy that the integration of database storage and object-oriented programming paves the way for more efficient and usable data management and analysis systems. In this task, we have built on open source tools as other researchers in the growing community of neuroscientist Python users,

²⁴<http://www.carmen.org.uk/>

²⁵<http://neuralensemble.org/trac/OpenElectrophy>

²⁶<http://groups.google.fr/group/openelectrophy>

²⁷<http://blobstreaming.org/>

²⁸<http://www.hdfgroup.org/HDF5/>

represented in this special issue. Emerging from this community are new solutions to promote data and code sharing, and we encourage others to participate and join in the development of a new generation of software to benefit to the whole neuroscience community.

REFERENCES

- Blatt, M., Wiseman, S., and Domany, E. (1996). Superparamagnetic clustering of data. *Phys. Rev. Lett.* 76, 3251–3254.
- Hanke, M., Halchenko, Y.O., Sederberg, P.B., Olivetti, E., Fründ, I., Rieger, J.W., Herrmann, C.S., Haxby, J.V., Hanson, S., and Pollmann, S. (2009). PyMVPA: a unifying approach to the analysis of neuroscientific data. *Front. Neuroinform.* 3,3. doi: 10.3389/neuro.11.003.2009.
- Ince, R.A., Petersen, R.S., Swan, D.C., and Panzeri, S. (2009). Python for information theoretic analysis of neural data. *Front. Neuroinform.* 3,4. doi: 10.3389/neuro.11.004.2009.
- Lewicki, M.S. (1998). A review of methods for spike sorting: the detection and classification of neural action potentials. *Network* 9, R53–R78.
- Litaudon, P., Garcia, S., and Buonviso, N. (2008). Strong coupling between pyramidal cell activity and network oscillations in the olfactory cortex. *Neuroscience* 156, 781–787.
- Pouzat, C., Delescluse, M., Viot, P., and Diebolt, J. (2004). Improved spike-sorting by modeling firing statistics and burst-dependent spike amplitude attenuation: a Markov chain Monte Carlo approach. *J. Neurophysiol.* 91, 2910–2928.
- Quiroga, R.Q., Nadasdy, Z., and Ben-Shaul, Y. (2004). Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. *Neural Comput.* 16, 1661–1687.
- Roux, S.G., Cenier, T., Garcia, S., Litaudon, P., and Buonviso, N. (2007). A wavelet-based method for local phase extraction from a multi-frequency oscillatory signal. *J. Neurosci. Methods* 160, 135–143.
- Wood, F., Goldwater, S., and Black, M.J. (2006). A non-parametric Bayesian approach to spike sorting. *Conf. Proc. IEEE Eng. Med. Biol. Soc.* 1, 1165–1168.
- Zito, T., Wilbert, N., Wiskott, L., and Berkes, P. (2008). Modular toolkit for data processing (MDP): a Python data processing framework. *Front. Neuroinform.* 2, 8. doi: 10.3389/neuro.11.008.2008.
- or financial relationships that could be construed as a potential conflict of interest.

ACKNOWLEDGMENTS

We thank Nathalie Buonviso, Tristan Cenier and Phillipe Litaudon for being the courageous first users of OpenElectrophy, Stephan Roux for original contributions to this project, and Eilif Muller for helpful proofreading of this manuscript.

Received: 12 September 2008; paper pending published: 27 October 2008; accepted: 30 April 2009; published online: 27 May 2009.

Citation: Garcia S and Fourcaud-Trocmé N (2009) OpenElectrophy: an electrophysiological data- and analysis-sharing framework. *Front. Neuroinform.* (2009) 3:14. doi: 10.3389/neuro.11.014.2009

Copyright © 2009 Garcia and Fourcaud-Trocmé. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial



DataViewer3D: an open-source, cross-platform multi-modal neuroimaging data visualization tool

André Gouws*, Will Woods, Rebecca Millman, Antony Morland and Gary Green

Department of Psychology, York NeuroImaging Centre, University of York, UK

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Stephen C. Strother, Baycrest, Canada;
University of Toronto, Canada
David Kennedy, Harvard Medical
School, USA

*Correspondence:

André Gouws, York NeuroImaging
Centre, University of York, York Science
Park, York YO10 5DG, UK.
e-mail: andre.gouws@ynic.york.ac.uk

Integration and display of results from multiple neuroimaging modalities [e.g. magnetic resonance imaging (MRI), magnetoencephalography, EEG] relies on display of a diverse range of data within a common, defined coordinate frame. DataViewer3D (DV3D) is a multi-modal imaging data visualization tool offering a cross-platform, open-source solution to simultaneous data overlay visualization requirements of imaging studies. While DV3D is primarily a visualization tool, the package allows an analysis approach where results from one imaging modality can guide comparative analysis of another modality in a single coordinate space. DV3D is built on Python, a dynamic object-oriented programming language with support for integration of modular toolkits, and development of cross-platform software for neuroimaging. DV3D harnesses the power of the Visualization Toolkit (VTK) for two-dimensional (2D) and 3D rendering, calling VTK's low level C++ functions from Python. Users interact with data via an intuitive interface that uses Python to bind wxWidgets, which in turn calls the user's operating system dialogs and graphical user interface tools. DV3D currently supports NIFTI-1, ANALYZE™ and DICOM formats for MRI data display (including statistical data overlay). Formats for other data types are supported. The modularity of DV3D and ease of use of Python allows rapid integration of additional format support and user development. DV3D has been tested on Mac OSX, RedHat Linux and Microsoft Windows XP. DV3D is offered for free download with an extensive set of tutorial resources and example data.

Keywords: visualization software, multi-modal neuroimaging, Python, VTK, fMRI, MEG, DTI, DV3D

INTRODUCTION

This paper describes DataViewer3D (DV3D), a software package built with Python¹ and designed and optimized to address many of the issues encountered when visualizing multi-modal neuroimaging data.

The combination of analyses from multiple imaging modalities is an important and growing trend in neuroimaging (e.g. McDonald, 2008; Stufflebeam and Rosen, 2007). Researchers are conscious of the limitations of individual imaging techniques and their associated analysis methods (e.g. Coltheart, 2006). With sites having access to more than one data acquisition technology, the neuroimaging community has the opportunity to compare and contrast results from different modalities and analysis approaches. Multi-modal techniques are used to exploit differences in results obtained from different techniques (e.g. Liu et al., 2006) and potentially provide converging evidence concerning researchers' hypotheses.

A variety of neuroimaging analysis packages are available to researchers, facilitating analysis of data from a complex and diverse range of data acquisition techniques. The Neuroimaging Informatics Tools and Resources Clearinghouse² list many of these tools. Commercial analysis software packages include ANALYZE™³

and BrainVoyager⁴. Widely used open-source analysis toolboxes for MATLAB⁵ are exemplified by Statistical Parametric Mapping (Frackowiak et al., 1997), Fieldtrip⁶, EEGLAB (Delorme and Makeig, 2004), mrVista (Teo et al., 1997; Wandell et al., 2000) and NUTMEG⁷. Stand-alone, cross-platform analysis packages include FSL⁸ and FreeSurfer⁹. In addition to analysis packages, a number of stand-alone visualization packages have been developed, some to complement particular analysis packages (e.g. FSL's FSLView¹⁰) and others independently of analysis packages (MRICron¹¹; 3D Slicer¹²).

Both analysis and stand-alone visualization packages are often customized solutions developed by a site to address their specific requirements. Many software packages are later extended to provide analysis frameworks for a more diverse range of hardware platforms, data types and analysis methods. Sharing and distribution of platform independent software with unified data formats allows the neuroimaging community increased access to analysis

¹<http://www.python.org/>

²<http://www.nitrc.org/>

³<http://www.analyzedirect.com/Analyze/>

⁴<http://www.brainvoyager.com/>

⁵<http://www.mathworks.com/products/matlab/>

⁶<http://www.ru.nl/fcdonders/fieldtrip/>

⁷<http://nutmeg.berkeley.edu/>

⁸<http://www.fmrib.ox.ac.uk/fsl/>

⁹<http://surfer.nmr.mgh.harvard.edu/>

¹⁰<http://www.fmrib.ox.ac.uk/fslview>

¹¹<http://www.sph.sc.edu/comd/rorden/mricron/>

¹²<http://slicer.org/>

methods. Researchers may have to compare the visual outputs of two or more different packages side by side, often comparing two-dimensional (2D) outputs from one to 3D outputs of another. The lack of a like-for-like comparison of results in a uniform coordinate space can increase the potential for misinterpretation of results. Reproducibility of results and consistency in analysis, interpretation, and display of results may be compromised when comparing results from different analyses and visualization software (e.g. Mackenzie-Graham et al., 2008).

DV3D does not attempt to compete with existing analyses packages in terms of analysis routines but rather acts as a support tool for neuroimaging analysis packages. DV3D allows users to integrate results from a number of different analysis packages, in a variety of formats and in an open-source, platform independent implementation. DV3D is designed to offer 2D and 3D visualization support for results from a number of neuroimaging acquisition modes and analysis techniques including magnetic resonance imaging (MRI), magnetoencephalography (MEG), positron emission tomography, computed axial tomography and diffuse optical imaging. DV3D has a highly modular, transparent design and is optimized for integration of additional display routines and file format support. DV3D provides export routines for high-resolution images, movies and objects created by the program for data sharing.

FSLView, 3D Slicer and MRICron are three of the most widely used stand-alone packages for visualizing neuroimaging data, and thus DV3D's functionality will be most closely compared and contrasted to them. None of these packages (and no other single stand-alone package to the best of our knowledge) offer support for all of the multiple analysis outputs of the aforementioned imaging technologies. DV3D is designed to fill this gap.

DV3D is built on Python, a cross-platform interpreted programming language. In DV3D, Python is used to wrap familiar, system-native Graphical User Interface (GUI) functionality using wxWidgets¹³ and powerful graphics rendering using the Visualization Toolkit¹⁴ (VTK). DV3D's code base is completely platform independent allowing code to run on any system with Python, VTK and wxWidgets installed. This minimizes code translation time and system-dependent error handling, increasing the efficiency of software development and new process integration.

First we outline the design objectives for DV3D. Following this we will discuss the value of using an open-source, platform independent framework for developing such a package, focusing on Python as the programming language to facilitate cross-platform software development. We will then outline the current functionality of the release package of DV3D and how it achieves our design objectives. We will conclude by comparing DV3D's functionality to similar existing tools, highlighting how DV3D currently provides more comprehensive functionality in a single package, as well as an accessible framework for future development by the neuroimaging community.

SOFTWARE DESIGN AND FRAMEWORK: DESIGN OBJECTIVES

While the exact requirements of every neuroimaging research environment are different, we note that many researchers regularly use a

number of core functions when either exploring their data visually or reporting results to their peers. The key requirements that we have tried to address in the development of DV3D are discussed below. They are:

- Dealing with different data types
- A common space for data
- Co-registration with atlases
- Export routines for sharing and publication
- An efficient working environment.
- A flexible, scalable and accessible open-source framework

DEALING WITH DIFFERENT DATA TYPES

Considering the number of different data sources in neuroimaging, many different ways to display the results of neuroimaging data have been adopted.

Due to the nature of their individual underlying analysis methods, many existing software packages are optimized for displaying results in their own preferred way. **Figure 1** summarizes some of these conventions using FSL, SPM, DTI-Studio¹⁵, FreeSurfer, mrVista and EEGLab as examples. Most packages are, understandably, optimized for the display of imaging results from a limited number of technologies, protocols, analysis methods and file formats. DV3D provides a platform in which the user can display a wider range of data in a number of different formats, be they 2D or 3D.

When considering the data types that a multi-modal neuroimaging visualization tool may be required to handle, there are at least four levels of abstraction we need to consider. An example of the complexity of the data structures that require consideration for neuroimaging data processing streams is shown in **Figure 2**. Analyzing and presenting data from MRI protocol subtypes alone requires a support for a broad range of data formats. A software package capable of supporting multi-modal data thus needs to consider: (a) the technology being used to acquire the different data types, (b) the acquisition settings (or protocol) being used to acquire the data, (c) the analysis techniques used to analyze the acquired data, and (d) the format in which the data and results are stored.

The first key objective of DV3D is to ensure flexibility in design that will enable users to integrate neuroimaging data whether it comes from different technologies, from different acquisition protocols, from different analysis approaches and independently of which data format they are saved in.

A COMMON SPACE FOR DATA

In order to sensibly overlay data for visualization of multi-modal analyses, we need to display the data in a common reference frame. An MEG data set, for example, will typically have a coordinate space defining the sensor positions, the participant's head shape and head position relative to the sensors. To overlay this data onto, for example, a surface extracted from an MRI scan, we need to align the coordinate space of the MRI scanner to that of the MEG scanner. Many analysis packages already have algorithms and processes for computing these alignments. Affine 3D transformation matrices are used to describe linear transformations as in FLIRT

¹³<http://www.wxwidgets.org/>

¹⁴<http://www.vtk.org/>

¹⁵<https://www.mristudio.org/>

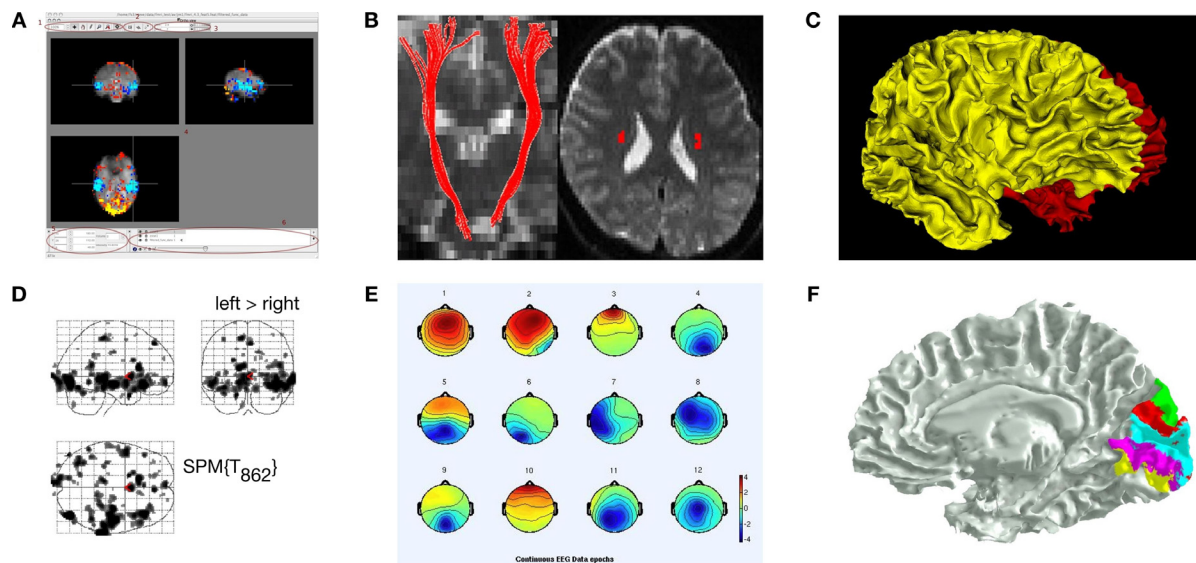


FIGURE 1 | Some common display conventions for neuroimaging data.

Examples of some of the methods commonly used to display neuroimaging data. **(A)** FSL's FSLView is used in this example to show the overlay of fMRI data onto three orthogonal planes generated for a 3D MRI volume. **(B)** DTIStudio can display DTI-fiber paths as streamlines mapped onto orthogonal planes generated from 3D MRI Volumes. **(C)** FreeSurfer can be used to display surfaces extracted from MRI data. In this example the grey matter to white matter

boundary is displayed in 3D, with separate surfaces for the left (red) and right (yellow) hemispheres of the brain. **(D)** SPM can be used to output 2D projections of regions of statistical significance to a 'glass brain' view. **(E)** EEGLab can be used to show iso-contour patterns of changing electrical fields over the scalp in 2D. **(F)** mrVista can be used to map scalar values (here different visual areas are represented by different colors) to a cortical surface.

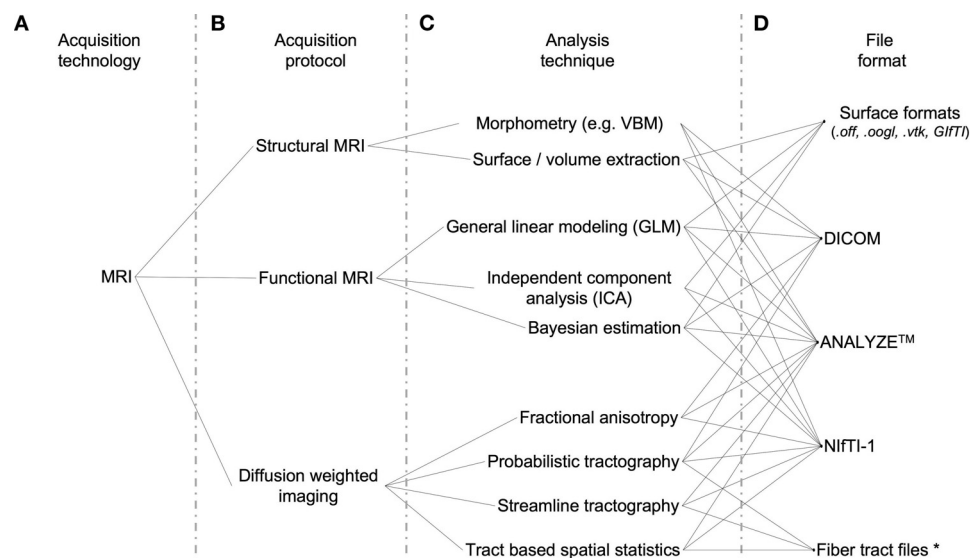


FIGURE 2 | Data handling complexity in MRI analysis streams. A schematic representation of the some of the levels of abstraction considered when preparing software capable of handling multi-modal neuroimaging data. **(A)** The technology type used: Here we use MRI as an example. **(B)** Some MRI acquisition protocols or sub-types: a researcher using a combination of protocols may, for example, be looking for changes in blood oxygenation using functional MRI, localizing the regions of activation to specific brain regions using structural MRI, and then looking for anatomical connections between these regions using Diffusion weighted MRI. They may then wish to overlay the results from each modality to explore spatial relationships. **(C)** Examples of the types of different analysis algorithms and routines for any given protocol. **(D)** Examples

of data formats: although researchers may use the same technology, the same protocol, and even the same analysis technique/algorithm, they may save their results in different file formats not immediately accessible to software utilized at other sites. * In the case of Fiber tract files, few standard file formats have been developed specifically for DTI data, and even fewer for saving the results of fiber tracking algorithm output. The .nrrd file format (http://www.na-mic.org/Wiki/index.php/NAMIC_Wiki:DTI:Nrrd_format) is used by 3D Slicer to load DTI values and parameters into memory. Fibers are subsequently calculated and can be saved to a vtk file format, unspecific for DTI fibers but useful for import and conversion by any VTK based programs, including DV3D.

(Jenkinson et al., 2002). Non-linear coregistration routines, as used in SPM (Ashburner et al., 1999) and FNIRT¹⁶, provide nonlinear, one-to-one coordinate mapping between data sets.

Data overlay in some existing packages is also limited by the resolution of the inputs. In FSL's current FSLView, for example, MRI data with a voxel resolution of $2 \times 2 \times 2 \text{ mm}^3$ cannot be overlaid onto a data set with a $1 \times 1 \times 1 \text{ mm}^3$ resolution, even if the data sets are defined in the same coordinate space.

The second key objective of DV3D is to enable users to align different data sets into a common reference space. As DV3D is not an analytical tool, we will refrain from calculating alignments on the fly. The alternative is to facilitate alignment by providing tools to load previously calculated transformations from other software packages. Additionally, once data sets are aligned, the resolution of the data sets should already have been interpreted and processed accordingly to allow sensible overlay and corresponding scaling.

CO-REGISTRATION WITH ATLASES

Neuroimaging analysis results often describe spatial distributions of significant activity in the brain. These *maps* are typically overlaid in 2D onto an individual or group brain data, as in **Figure 1A**, so that this spatial distribution can be seen.

In addition to viewing data in an individual or across a group, it is common practice in many neuroimaging data modalities to compare these spatial distributions to equivalent positions, and thus brain structures, in some reference brain space. These reference brains, or atlases, include the MNI brain (Mazziotta et al., 2001), the Talairach brain (Talairach and Tournoux, 1988), the Harvard-Oxford cortical and sub-cortical structural atlases¹⁷ and the ICBM-DTI-81 white-matter labels atlas (Wakana et al., 2004). At the time of this submission, the current version of FSLView cross-references and reports information for the equivalent structures in all of the above atlases if the data set loaded has been transformed into the MNI coordinate space. An alternative for users not using FSLView would be to transform their data into the MNI coordinate space and then use the online MNI-Talairach daemon¹⁸ to manually check every point of interest – a more time-consuming process.

Incorporation of functionality to allow cross-referencing with other standardized brain volume data is thus the third key objective of DV3D. The ability to do this in real-time, without any additional software dependencies is also preferable.

EXPORT ROUTINES FOR SHARING AND PUBLICATION

The production of informative, high-resolution images for communication of results in publications, presentations and educational material is a fundamental requirement in neuroimaging. Many neuroimaging data analysis packages have export routines to capture screen contents to static reports, individual frames to high-resolution images and even short movies of rotating 3D objects or time-series data. Researchers using a specific analysis package can also share data sets with each other. By providing another user with a data set and a set of instructions, the secondary user can reproduce the same analysis or visualization result.

As a fourth objective, DV3D should facilitate the export of data from the visualization screen to a number of formats with options for control of resolution. Movie export options should allow users more freedom in terms of temporal and spatial interaction with data visible on the screen. DV3D should also provide a functionality for users to share results, even without having to provide raw data sets from which the results have been produced.

AN EFFICIENT WORKING ENVIRONMENT

Analysis of neuroimaging data can be a very labor-intensive process. Visualization and interpretation of obtained results adds significantly to this workload. Any functionality that saves the user a significant amount of time and effort is valuable. Many approaches can be taken to increase the efficiency of processing pipelines in software. Perhaps the most obvious is to ensure that, at the design stage, the processing pipeline for a software package is optimized for the hardware and software framework it is built on.

Current computing gives researchers access to multiple processors that can handle computations independently or in parallel. Many computing facilities extend this model to computing clusters with multiple nodes across which processes can be distributed or parallelized. Access to parallel processing is already a feature of a few of the existing neuroimaging software packages. FSL's Bayesian Estimation of Diffusion Parameters Obtained using Sampling Techniques (BEDPOST) toolbox¹⁹, for example, can be easily configured to run over Sun Grid Engine²⁰, or even simply distributed across any additional local processors.

While parallel processing in the context of BEDPOST is utilized to reduce the amount of processing time required to generate results, the principle can be applied to computationally expensive visualization routines when viewing results. Loading surfaces with millions of vertices and rendering them is an example; a user wanting to load multiple surfaces into memory may still have to wait in the order of minutes for them to load and render. While computers have increasingly large amounts of memory, allocation and management of memory is still a problem that any software designer needs to take into account. This is especially poignant when handling neuroimaging data where data sets can be very large. It is common for MEG data sets acquired at high sampling rates to exceed 1 GB in size. Memory allocation errors are often terminal, causing a computer program to crash if allocation fails. This can be both frustrating and inefficient.

Many of the analysis routines applied to neuroimaging data are repetitive; analysis of data from each individual in a group is an example. Automation of processing streams for similar data sets is an increasing feature in neuroimaging data analysis. Users often use scripts to pass list of arguments and settings into a program that can be accessed via a command line. This can help to reduce the overheads associated with repetitive GUI interaction. In this way, a researcher can apply the same processing, thresholding, and result export routines for each individual in a large group with a single file and a single button press, even if they then do have to wait several hours for the process to complete. This principle can be a useful feature for the visualization of results. A user may want

¹⁶<http://www.fmrib.ox.ac.uk/fsl/fnirt/>

¹⁷<http://www.cma.mgh.harvard.edu/>

¹⁸<http://www.talairach.org/applet/>

¹⁹http://www.fmrib.ox.ac.uk/fsl/fdt/fdt_bedpostx.html

²⁰<http://gridengine.sunsource.net/>

to, for example, provide an instruction list to a program to load a particular surface, overlay a statistical result file, threshold to a specified value, export a high-resolution image from a top-down view and save a movie. The user would then have a template to process different statistical results, different thresholds, or simply different participants without having to manually run each individual through a GUI.

Some software packages help to increase user productivity by saving metadata files that describe the current status of the workspace the user is working in. The MATLAB toolbox, mrVista, is a good example. In this package users have a session file for each individual. Many settings, file paths, and associated analysis outputs are automatically loaded for the user the next time they load a previously processed participant's data. Evidently, a metadata file describing the processes applied to a data set, its overlays, and dependent thresholds is potentially time-saving when dealing with the visualization of neuroimaging data sets. Furthermore, such a file could easily be shared with another researcher to ensure a consistent result when viewing the same input data.

Saving of processing metadata and automated processing scripts both provide a reference which describes the processes and routines used to produce a set of results. The use of scripts to drive analysis and visualization routines decreases the chances of inconsistencies due to user error. Provenance, the description of the history of a set of data, is important with the recent increases in cross-site collaboration and data sharing (e.g. Mackenzie-Graham et al., 2008). The LONI Inspector²¹, an application for examining medical image files, is an example of a tool developed for the comparison of the metadata stored with and between different file types. Metadata is particularly informative when files are converted from one format to another. Assumptions about default orientations, for example, can cause left-right flipping of the data during the conversion process and can cause errors in subsequent visualization and interpretation.

Access to parallel processing, command line scripting, session or workspace metadata and efficient memory management are all ways in which a neuroimaging visualization tool can increase user productivity. As such, the fifth objective in the development of DV3D is to utilize a software and hardware framework that encompasses as many of these features as possible.

A FLEXIBLE, SCALABLE AND ACCESSIBLE OPEN-SOURCE FRAMEWORK

An open-source software package with a self-supporting user community can be a viable solution for scientific software development. With a community contributing to code development and maintenance, costs can be minimized. Other factors need to be considered when developing useful, sustainable open-source software packages.

Transparency is a factor that concerns many researchers, although this is more often related to the implementation of analysis algorithms. While there is very little analysis *per se* in stand-alone visualization packages, researchers should have access to processing routines that generate the visual output (e.g. the color lookup tables applied to thresholded statistical overlay data and interpolation routines applied to loaded data).

Accessibility of the code base can be an issue that restricts interested users from understanding and developing programs. At least three factors can be considered to affect the accessibility of software:

- *Educational resources* are crucial to aid users in learning how to use a package. Documentation and tutorial routines are often lacking in software packages restricting the range of potential users.
- *Platform independence* is an increasingly common feature in neuroimaging software packages. Software that runs on any hardware platform is not only more accessible to any individual site, but aids collaboration across different sites with potentially different hardware infrastructures.
- *Coding language*. Some coding languages are more complex and / or less intuitive than others. While it is impossible to provide a coding language that every programmer would like, it may be sensible to settle for a compromise between a language that is simple to read and use, and one that is very powerful and efficient.

Extendibility and flexibility of software is a measure of how easily the software can be expanded to incorporate additional processing routines. Since the authors have not set out to predict every possible permutation of input-to-output requirement of potential users, it is crucial that the software framework is designed to facilitate incorporation of additional routines with minimal effort. A modular software framework not only facilitates such independent development, but allows for incorporation of appropriate tools and routines often developed for completely different purposes. We could, for example, choose to incorporate an implementation of an algorithm for decimating surfaces, borrowing the code from an external mathematics toolbox. Once imported into the package as an independent module one could simply pass a brain surface to this module as a set of vertices and run the module to down-sample the number of vertices for increased rendering speed.

DV3D has been designed with an open-source, user community developed model in mind. As such it is imperative that the package is built on a software framework that is accessible to a wide variety of users on a wide range of hardware platforms, extendible by non-specialist developers, intuitive to use, and well documented.

METHODS: IMPLEMENTING A Python FRAMEWORK

Having outlined the key objectives for a new multi-modal neuroimaging data visualization tool, we can now consider the implementation of the project. The software package can be considered to consist of three main components:

1. The visualization engine: this is the lowest level of the program, i.e., the functions that actually do the rendering of the images to the screen.
2. A user interaction interface: this is the component of the program that allows users to control the rendering routines of the visualization engine in an interactive and intuitive manner.
3. A master control program: the component of the program that binds or wraps the functionality of the underlying components and allows them to run on the operating system.

²¹<http://www.loni.ucla.edu/Software/>

We will discuss each of these components in turn, highlighting the requirements and implemented solution for each.

THE VISUALIZATION ENGINE: VTK

The Visualization ToolKit (VTK) is a widely used, free, open-source software package for data visualization and image processing, with support for 2D and 3D graphics rendering. With an active and vast international development community, VTK is a model for open-source software development.

VTK has an extensive set of implemented visualization algorithms. Routines for processing scalar, vector, tensor, texture, and volumetric methods exist. VTK offers a large variety of complex algorithms as part of the standard toolkit, many of which are directly useful for visualizing neuroimaging data. Contouring, surface decimation and triangulation, re-sampling, cutting, and interception detection are just a few examples. Many of these algorithms are directly integrated into widgets allowing users to interactively interrogate combinations of 2D and 3D data in real time. VTK is licensed under the BSD license. VTK is reported to have been installed and tested on nearly every Unix-based platform, Windows PC, and Mac OSX Jaguar or later. VTK is an efficient and fast toolkit consisting of an extensive C++ class library, access to which is available via several interpreted interface layers including Tcl/Tk, Java, and Python.

USER INTERFACE: WXWIDGETS

Learning to use a new software package can be challenging. In a program with a number of complicated functions, the provision of a highly interactive GUI and familiar workspace environment should benefit the user. wxWidgets is a free, open-source toolkit that provides developers with an API (application programming interface) for writing GUI applications on multiple platforms. wxWidgets is licensed under the *wxWindows license*, essentially the L-GPL (Library General Public License), with an exception stating that derived works in binary form may be distributed on the user's own terms. By using each platform's own native controls rather than emulating them, wxWidgets applications look and feel familiar to the operating system's, and should thus be immediately more familiar to the user. The list of widgets and features offered is extensive and the code base is very mature. wxWidgets can be called via interface layers for a variety of languages including C++, Python, and Perl.

Either C++ code or Python could be used to produce a program with a GUI in wxWidgets containing a VTK window for rendering. The relative ease of use of Python over C++, combined with the large array of readily accessible functionality offered by Python, makes this the preferred choice for our application.

THE MASTER ENVIRONMENT: Python

Python is a dynamic, object-oriented programming language that is reported to run successfully on Linux, Windows, FreeBSD, Macintosh, Solaris, and other operating systems. Since Python is an interpreted language, it internally converts and translates source code into the native language of the computer and then runs it. Once Python has been installed on a system, users do not have to compile a Python program or worry about library linkage and loading. Python programs are portable: copying the source code from

one operating system onto another (which has Python installed) will allow the software to run.

The Python-specific *Python license* is compatible with GPL licensing. Python is distributed with extensive standard libraries. The list of functions implemented in Python is extensive. Additional modules for Python include a number of mathematical, numerical methods and plotting toolboxes that are useful for manipulating numerical lists and arrays, before passing data into VTK for rendering. Some Python modules support parallel processing and threading often with as few as three lines of additional code (an example is provided in **Figure 10**). Modules allowing access to system command calls and environmental variables are abundant, allowing the user to spawn and even control external processes and applications from within the Python environment application. Python supports integration with other languages and tools (including wxWidgets and VTK), which are often loaded by nothing more than using the *import* command.

Python and individually distributed toolboxes can be built from source and installed independently. At the time of this submission an increasing number of developers are producing binary installers for entire Python distributions with many core modules including VTK. Using the academic download of the Enthought Python Distribution²², users on Windows, Mac OSX, or RedHat Linux have access to a 'one click installation' of the Python framework required to run DV3D.

In short Python was chosen over C++ for the development of DV3D because of its relative ease of use, the vast array of additional functionality available, and because it allows access to the core underlying components (wxWidgets and VTK) in a single programming language.

DEPENDENCIES AND INSTALLATION

Dependencies

For the reasons we have already discussed in detail above, DV3D is designed to be as platform independent as possible.

DV3D has few software or hardware dependencies and requires only the following to run:

- Python 2.4.1 or later
- wxPython 2.6 or later
- VTK 5.0.3 or later
- The Numpy module for the appropriate version of Python installed
- A Windows, Mac OSX, or Linux platform.

Installation

We have already outlined that Enthought provide a binary installer for Microsoft Windows, Mac OSX, and RedHat Linux. Use of these installers provides a comprehensive build of the core components and additional modules required to run DV3D. Use of the Enthought installers is currently free for academic use. Users with platforms not supported by these installers can often find binary installers for the individual components on operating specific support sites. All modules can be built from source on platforms by users wanting additional installation options and control.

²²<http://www.enthought.com/products/epd.php>

DATA IMPORT

Supported formats

DV3D currently supports the following formats:

- DICOM. Digital Imaging and Communications in Medicine is a standard for handling, storing, printing, and transmitting information in medical imaging²³. Many MRI scanners now export their data directly to this format. The DICOM format provides private header fields that can be utilized to store additional scan information. Unfortunately many sites now use these fields in a non-uniform manner (according to the DICOM standard). Different DICOM readers do not always correctly interpret metadata describing data acquisition and storage protocols in the file. DV3D addresses inconsistencies in DICOM headers by adjusting the DICOM reading routines provided by Python to specific scan protocols and scanner types.
- ANALYZE™ (*.hdr* and *.img*) is an image processing program developed by The Biomedical Imaging Resource at the Mayo Foundation. This program uses the ANALYZE™ format (www.mayo.edu/bir/PDF/ANALYZE75.pdf) which is currently widely used in neuroimaging. Many programs (including FSL, SPM, AFNI, Cox, 1996, FreeSurfer and MRICron) are able to read and write the format. The files typically store voxel-based volumes in two files: the binary data itself is stored with a filename extension *.img*; another file acts as a header (*.hdr*) describing information about the data such as voxel size, slice numbers and data origin. As with DICOM, some software packages use the ANALYZE™ format header in different ways. Some software packages interpret ANALYZE™ volumes differently due to differences in header writing conventions across sites. DV3D addresses inconsistencies in ANALYZE™ headers by adjusting the reading routines to detect which program was used to produce the file (where possible).
- NIfTI-1 (*.nii* or *.nii.gz*) is an adaptation of the ANALYZE™ 7.5 file format²⁴. NIfTI-1 uses unassigned spaces in the ANALYZE 7.5 header to add several new features. Since it is possible to compress data stored in NIfTI-1 files the *.nii.gz* file format is often utilized. DV3D supports the *.nii* or *.nii.gz* file formats.
- GIFTI (*.gii*). Support for the unified XML-based GIFTI file format²⁵ is provided.
- VTK polydata files (*.vtk*). VTK provides routines for exporting objects in memory to its own native polygon data files. Additional routines allow these objects to be read into VTK applications at a later date. This offers an incredibly useful tool for users wanting to save objects created in a VTK session for sharing or later access without the need for regeneration. DV3D offers visualization routines for *.vtk* files in binary or ascii format.
- OFF (*.off*). The Object File Format is described by the Geomview package²⁶. It is used to represent collections of planar polygons with possibly shared vertices. This is a useful format used to

describe surfaces by programs including SurfRelax (Larsson, 2001). DV3D offers visualization routines for *.off* files in binary or ascii format.

- FREESURFER surfaces (*lh.** and *rh.** are examples). Surfaces generated by typical default processing in FreeSurfer include left and right hemisphere cortices representing the white matter and grey matter surfaces, with anatomically correct and inflated versions. DV3D offers support for these standard surfaces and additional surfaces generated by post-processing routines (an extracted scalp for example). DV3D is also capable of handling additional scalar descriptors for these files, including curvature values. DV3D offers visualization routines for FreeSurfer files in binary or ascii format.
- 4-D Neuroimaging (4DNI) MEG data (*.m4d*). Creation of a *.m4d* file using the *pdf2set* program allows direct reading of 4DNI MEG data. DV3D currently supports the 4DNI output format, but could easily be extended to support other MEG and EEG time-series formats.

Although many of the formats discussed above have a standard description, i.e., a set of instructions for file creation designed to maintain conformity across sites, not all packages use these formats to read and write files in the standardized way. There will always be corner-cases where the readers used to import data into DV3D may fail. Fortunately, the previously discussed power of Python allows developers to easily amend existing readers or write new ones to handle these inconsistencies. Users are actively invited to submit failing data sets with descriptions of acquisition parameters and header formats so that current readers can be amended or new readers developed.

Supported software packages

Since DV3D currently supports all the data formats outlined above, it should, in theory, support at least some of the formats from a wide range of existing neuroimaging analysis packages. Any package capable of writing these formats could be used. This is not so simple in practice, as we have alluded to in the Section ‘Supported Formats’ of this paper. There are complications when different sites and packages adopt varying standards for data export to specific formats. We look forward to collaborating with sites with additional data sets in order to resolve as many of these disparities as possible.

Program processing pipeline

On startup, the user can choose to launch DV3D in one of two modes.

- *MRI-overlay mode*. This mode is traditionally used where a ‘base’ MRI volume is initially loaded. Other objects aligned to the coordinate space of this volume can then be loaded and overlaid onto the base volume. The ‘base’ MRI volume thus defines the coordinate space into which additional objects are loaded.
- *Non-overlay mode*. The user can choose to not load a base volume. In this case the program will launch with an empty renderer and pre-created 2D or 3D objects can be loaded by the user.

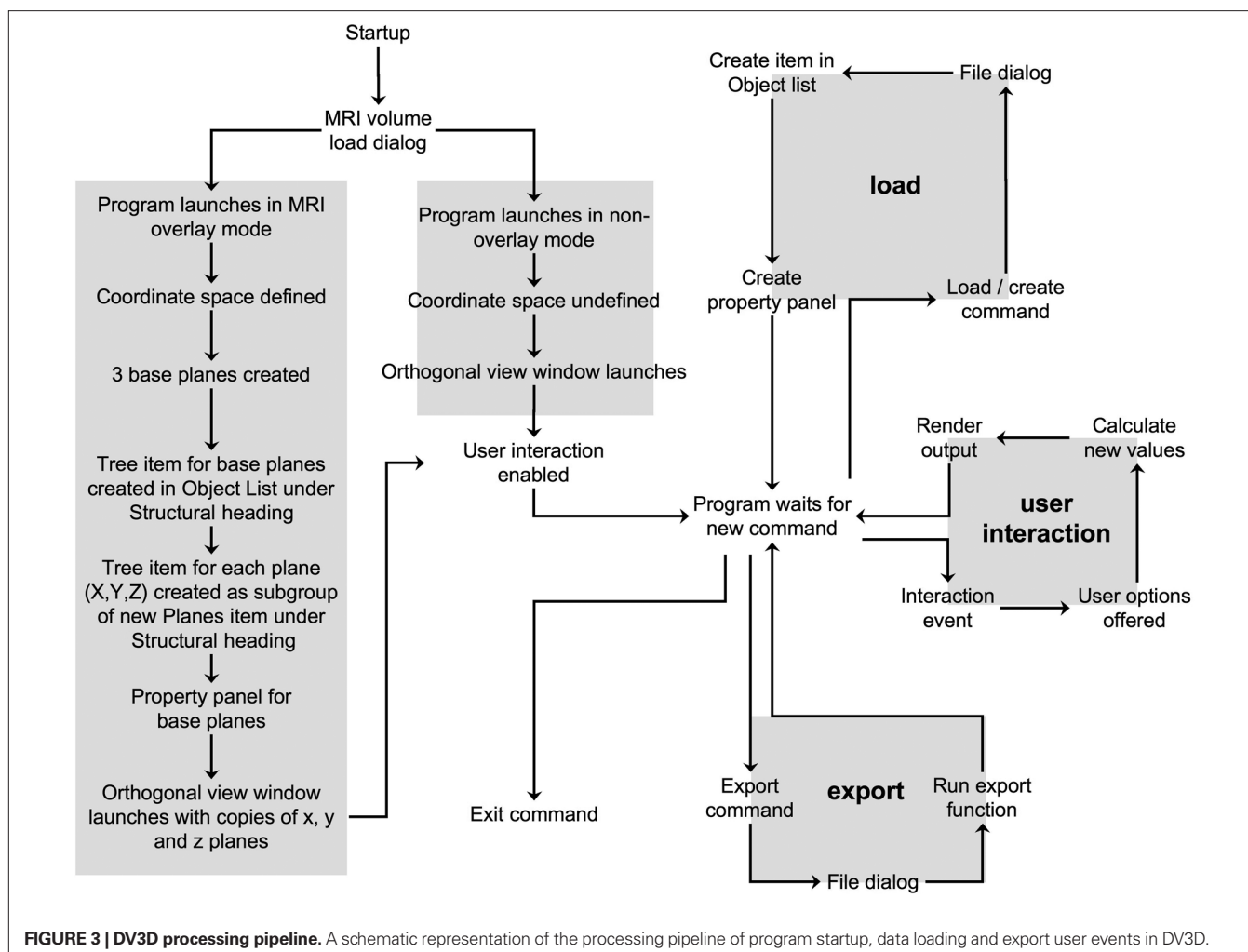
A graphical representation outlining DV3D’s processing pipeline is shown in **Figure 3**.

²³<http://medical.nema.org/>

²⁴<http://nifti.nimh.nih.gov>

²⁵<http://www.nitrc.org/projects/gifti/>

²⁶<http://www.geomview.org/docs/html/OFF.html#OFF>



RESULTS

DV3D is accompanied by user documentation, example data sets and tutorial videos. Links to this information are provided in the Supplementary Material section of this paper. The fine detail describing interaction with the application is described in these documents and tutorials. Here instead we will discuss the broad concepts and functions of the program, and how they satisfy our design objectives.

DESIGN OBJECTIVE: A COMMON SPACE FOR MULTIPLE DATA TYPES

DV3D's workspace

DV3D provides a single, common workspace for viewing neuroimaging data, simultaneously in 2D and 3D. The main workspace environment of DV3D consists of two windows:

Main application window (Figure 4). This window is divided into quadrants:

- *VTK window.* The bottom-right quadrant holds the *wxVTKRenderWindowInteractor*, the VTK class that allows a functional VTK session to be embedded in a *wxPython* program. We will refer to this as the *VTK window*. When data objects are loaded into or created by DV3D they are added to

this window. The *VTK window* is the core tool allowing us to provide a common space for simultaneous multi-modal data overlay.

- *Button Panel.* The top-right quadrant is constructed from a *wxNotebook* object that we will refer to as the *Button Panel*. It consists of a number of pages which each contain a panel of buttons and widgets which allow the user to interact with the *VTK window*. A tab labeled with the title of the panel denotes each page. Each page is brought to the front by clicking on its tab. Pages group functions of similar types together for ease of navigation. The *Button Panel* can be extended to have many more pages, allowing for a multitude of additional functions to be added to DV3D at a later date without excessively cluttering an individual button page. Potential developers will also be interested to note that each page here is derived from a separate class allowing easy parallel development and integration.
- *Object List.* The bottom-left panel holds a *wxTreeCtrl* that we will refer to as the *Object List*. It displays its items in a tree like structure similar to many operating systems' file browsing dialogs. An item may be either collapsed (meaning that its children are not visible) or expanded (meaning that its children are shown). Whenever a new object is loaded into the program

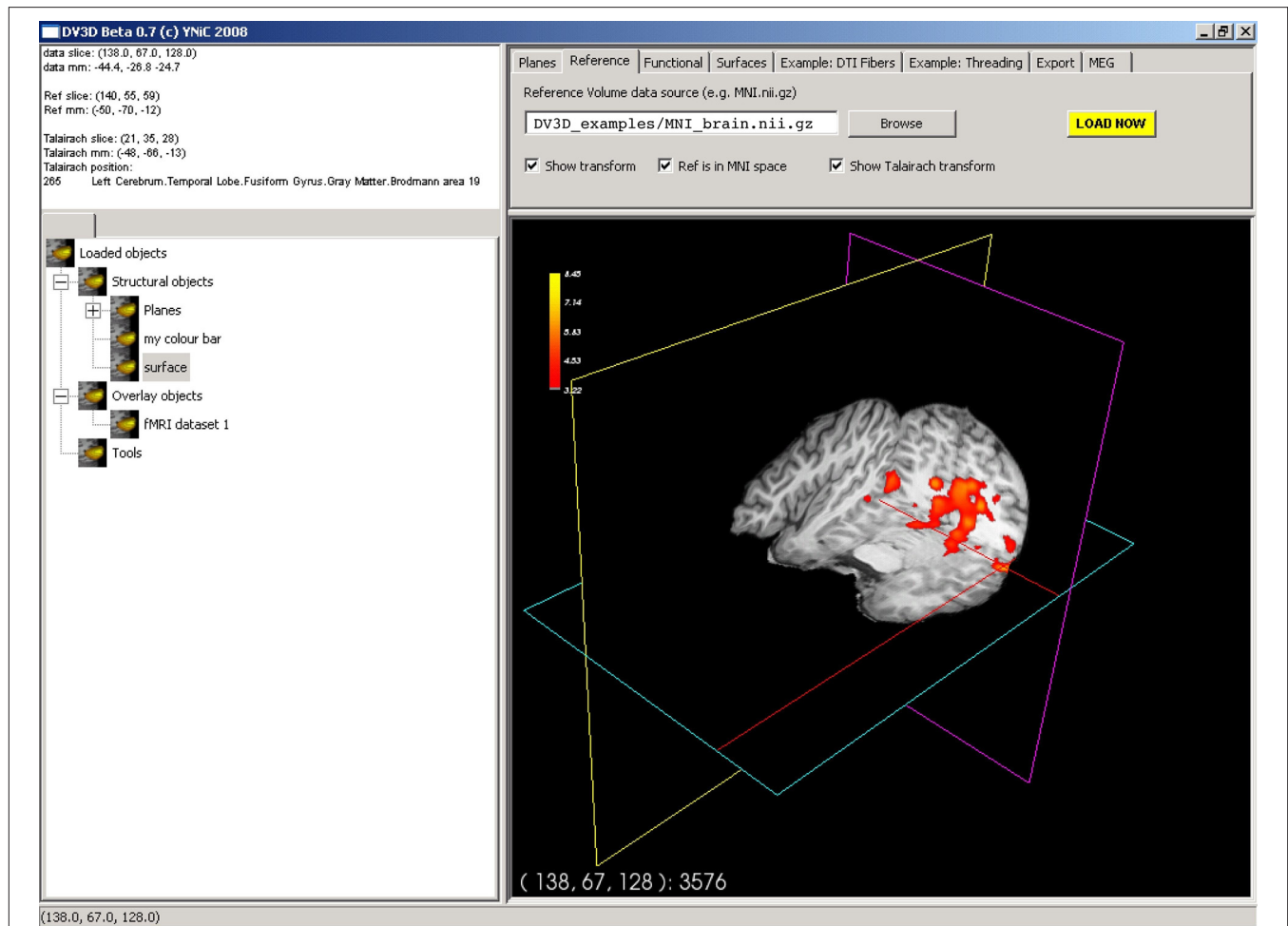


FIGURE 4 | DV3D's main application window. The main window for data interaction in DV3D. The bottom-right quadrant holds the *VTK window* where all 3D rendering takes place. The top-right holds the *Button Panel*, which consists of multiple sub-pages allowing a large array of user interaction functions. The top-left quadrant holds the *Message dialog* which displays the current

coordinates of the interaction cross hair in the *VTK window*. The bottom-left quadrant holds the *Objects List*: a list of all objects loaded in the the *VTK window*. Panels can be resized by clicking and dragging the vertical and horizontal dividers between each panel. Views in the *VTK window* are neurological by convention.

or generated by one of DV3D's routines, a tree item is added to this list. In addition to this, a property panel is created for each new object. This panel has a number of different buttons and tools used to manipulate the display properties of the objects in the *VTK window*. Since a unique item identifier identifies each item in the tree, it can be linked to the object in the *VTK window*. This allows us to manipulate some of the properties of the object in the *VTK window* associated with a specific item in the *Object List* simply by clicking on the object in the list. Each item has its own (optional) icon and a label. Users can simply rename the item in the tree to a more meaningful string without losing the interaction with the associated object in the *VTK window*. The *Object List* offers an intuitive and efficient tool for managing the content of the *VTK window*.

- *Message Dialog*. The top-left quadrant, which we will refer to as the *Message Dialog*, holds a *wxTextCtrl*. This object is effectively a text box that is updated with information for the user as the program is used. Interaction coordinates from the *VTK*

window (bottom right quadrant) are displayed in the *Message Dialog* if a base MRI volume is loaded.

- *Sizers*. A vertical and horizontal sizer bar define the boundaries of the quadrants. Clicking and dragging these sizers allows the user to alter the relative sizes of the quadrants of the *Main application window*.

The *Main application window's VTK window* allows us to display multi-modal data, whilst the *Button Panel*, *Object List* and object associated *Property Panels* allow us to manipulate the properties of the displayed objects.

In addition to the 3D viewing capabilities of the *VTK window*, DV3D provides traditional 2D orthogonal views of the 3D window via the *Orthogonal view window*. This window consists of three orthogonal projections of the *VTK window's* content. The options panel in this window allows the user to set the refresh frequency of the viewports, increasing program performance. Plane orientation and placement of the viewpoints is also fully customizable.

Viewing conventions

It is important to make the default visualization conventions of DV3D clear at this stage.

Radiological vs. neurological. Data viewed in the 3D VTK window of the *Main application window* is rendered according the neurological convention as described by FSL²⁷. Data viewed in the 2D *Orthogonal view window* also conforms to the neurological convention, but can be switched to the radiological convention.

Perspective vs. parallel projection. To make 3D visualization more natural, the VTK window utilizes a perspective projection algorithm during rendering to infer depth in the scene. Since the planes in the *Orthogonal view window* are effectively 2D we refrain from using this algorithm (since it carries some processing overhead) and revert to parallel projection.

Aligning different data sets

Transformations. DV3D allows the user to add different data sets of different types into the same coordinate space (the VTK window). Data is loaded into a millimeter coordinate frame defined by the data set's header description (e.g. the sform or qform matrices held in the header of NIfTI-1 files). By using header transformation matrices, DV3D can automatically align data. Alternatively, the user can provide additional affine transformations (4×4 matrices) to apply previously calculated alignment parameters (typical examples include affine transformations provided by FSL's FLIRT when coregistering an individual MRI to the MNI brain). This principle applies to any volumes or surfaces loaded. DV3D does not currently calculate new transformations, but rather handles those pre-calculated in external analysis packages.

Resolution and scaling. Unlike many other visualization packages (e.g. FSLView), DV3D does not require MRI data to be at the same resolution. DV3D uses a millimeter coordinate space. All data loaded into the VTK window are scaled according to the header information (e.g. the *pixdim* values in ANALYZE™ and NIfTI headers describe the voxel dimensions).

DESIGN OBJECTIVE: DEALING WITH DIFFERENT DATA TYPES

Viewing volume data in 2D and 3D

The *vtkImagePlaneWidget* is the core tool utilized by DV3D to display and interact with volumetric MRI data and associated overlay volumes. This widget works by creating a plane that can be interactively placed in an image volume. Readers may ask why a 2D tool is incorporated in a 3D data viewer. VTK allows the user to manipulate this plane in real time, using the third dimension to tilt, rotate, or translate the plane in virtually any orientation. Thus a 2D plane becomes a diverse data exploration tool. **Figure 5A** shows a set of planes created for an MRI data set. The functionality of the *vtkImagePlaneWidget* is described in detail in the tutorial examples and documentation. In short, it offers the following functionality:

- **Coordinate lookup.** DV3D captures the slice number data displayed by the *vtkImagePlaneWidget* and uses it to calculate

the equivalent millimeter coordinates in the underlying data set. The slice number and calculated millimeter coordinates are then displayed in the *Message Dialog* of the *Main application window*. **Figure 5B** shows the lookup cross-hair activated in the plane.

- **Interactive volume re-slicing.** The core functionality of the widget relies on the *vtkImageReslice* class that takes the image volume data as an input, re-slices (or 'reformats') it as required and then passes the output to the texture mapping pipeline. This tool allows real time slicing through volumetric data at virtually any angle. **Figures 5C–E** show this functionality in action.
- **Brightness and contrast.** In addition to rotation and translation of the planes, it is also possible to change the windowing and level of the data. This effectively adjusts the brightness and contrast of the data displayed in the window. Slider style controls are provided to control the absolute values of the window width and level for more precise user control. The default behavior allowing the mouse to control window width and level can be re-enabled in *User Preferences*.

Using multiple *vtkImagePlaneWidgets*, DV3D allows simultaneous overlaying of statistical data in 2D. Once a base volume has been loaded and its planes have been created, additional volumes can be loaded and overlaid onto this volume. The overlay load routine is accessed via the *Functional* tab on the *Button Panel*. Overlay volumes currently have to be transformed into the coordinate space of the base volume but do not need to be at the same resolution. For every overlay volume loaded, an additional set of planes is created; one for each axis in the VTK window and one for each axis in the *Orthogonal view window*. The overlay data is initially assigned a yellow (for its minimum value) to red (for its maximum) color lookup table before it is rendered. As with the base image planes, two additional objects are created: an *Object List* label and a *Property Panel*. Sliders control the window width and window level of the overlay layer only, i.e. the effective scalar range for the data that are visible in the overlay layer. This acts as a real time 2D and 3D statistical thresholding tool. The color map currently in use can also be altered using the color map selection dialog.

Viewing 3D surfaces

DV3D provides methods for loading and generating surfaces for display in the VTK window. Surfaces are created in memory as *vtkPolyData* objects, which have a number of native properties that the program is able to manipulate to increase user interactivity. Examples include access to the global transparency and color properties of the object. These properties can then be altered using the property panel automatically created for any surface loaded or generated.

Loading surfaces. Surface load routines are accessed via the *Surfaces* tab on the *Button Panel*. Clicking the Load button opens a file dialog offering the import of a number of different file formats. Surface inputs currently supported by DV3D include:

- FreeSurfer output surfaces (including inflated surfaces).
- SurfRelax output surfaces in the Geomview binary.off file format.

²⁷http://www.fmrib.ox.ac.uk/fslfaq/#general_radiologicaldef

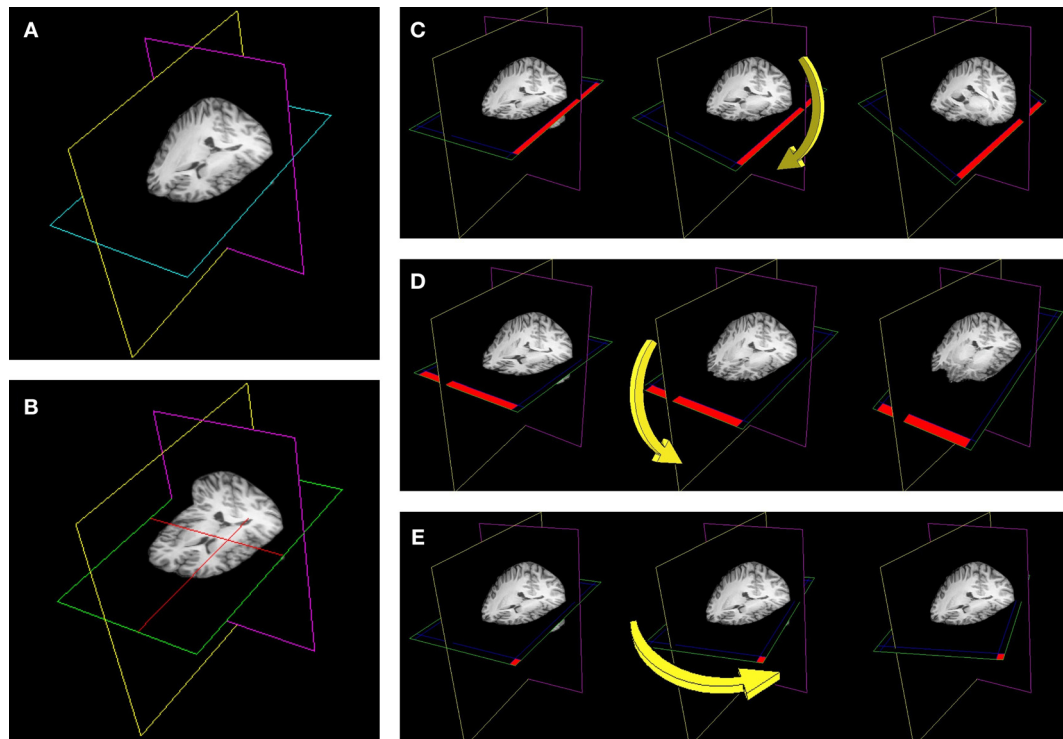


FIGURE 5 | The use of plane widgets to show 3D volume data. (A) A set of three orthogonal planes, each intersecting a single 3D MRI volume. **(B)** Left clicking on any one plane with a mouse will make a cross-hair visible (in red) allowing data from a specific coordinate in the data set to be displayed. Real-time reformatting of data (re-slicing it in any plane direction) is possible by tilting

the planes around their current origin. **(C)** The axial plane is rotated around the y-axis by clicking on the edge of the plane (shown in red) and moving the mouse. **(D)** The axial plane is rotated around the x-axis by clicking on another plane edge (shown in red). **(E)** The axial plane is rotated around the z-axis by clicking in the corner of the plane (shown in red).

- mrVista.mrm outputs.
- vtkPolyDataWriter output files (.vtk).
- Any surface exported to the GlTFI format.

Once the surface load dialog completes the object is loaded and automatically added to the *VTK window* and the *Orthogonal view window*. The automatically generated property panel will also be displayed.

Generating surfaces. VTK provides techniques for dynamically generating surfaces from volume data in memory. DV3D uses the *vtkContourFilter* to calculate and extract surfaces from underlying MRI data volumes. The *vtkContourFilter* interrogates the volume data set, finding points in the volume where the scalar value corresponds to a value stipulated by the user. It then scans through the data volume, connecting points of the same value and creating isocontour lines (in 2D) or isosurfaces (in 3D). Since the stipulated search value may occur several times in the data volume, multiple isolines or isocontours can be returned by the algorithm. An additional option offered by the algorithm is to retain only the largest connected surface, i.e., the surface with the largest number of vertices.

It may be interesting to generate surfaces from underlying data for a number of reasons. In **Figure 6** we show an example of a rough estimate of a scalp (**Figure 6A**) and rough cortical surface (**Figure 6B**) representative of the white-matter/gray-matter

boundary, extracted from the same individual's data. Isosurfaces extraction is highly sensitive to homogeneity inconsistencies in the MRI image volume and produces better results with intensity normalized volumes. In **Figure 6C** we show the same routine applied to the skull-stripped $1 \times 1 \times 1 \text{ mm}^3$ MNI brain distributed with FSL 4.0. It should be evident that this result is less noisy than that shown in **Figure 6B**, a result of the intensity normalization of the MNI brain. Surface generation for cortical surfaces using DV3D is meant to aid quick data exploration and is not nearly as informative or accurate as the algorithms utilized by programs like FreeSurfer, FSL's FAST²⁸ or SurfRelax. The speed with which an individual can extract a rough representation of this surface is however very useful. DV3D can give a user a quick insight into the cortical shape in just 30 s, where other packages take between 15 min and several hours to run.

Activation color mapping. In addition to offering access to the global transparency and color properties of the object, *vtkPolyData* objects allow access to the properties of individual vertices that define the shape of the surface. Each vertex can have a scalar value associated with it. VTK allows the user to create a color lookup table covering the range of all scalar values associated with the vertices of a surface. The color presented at each vertex on the surface can

²⁸<http://www.fmrib.ox.ac.uk/fsl/fast4/>

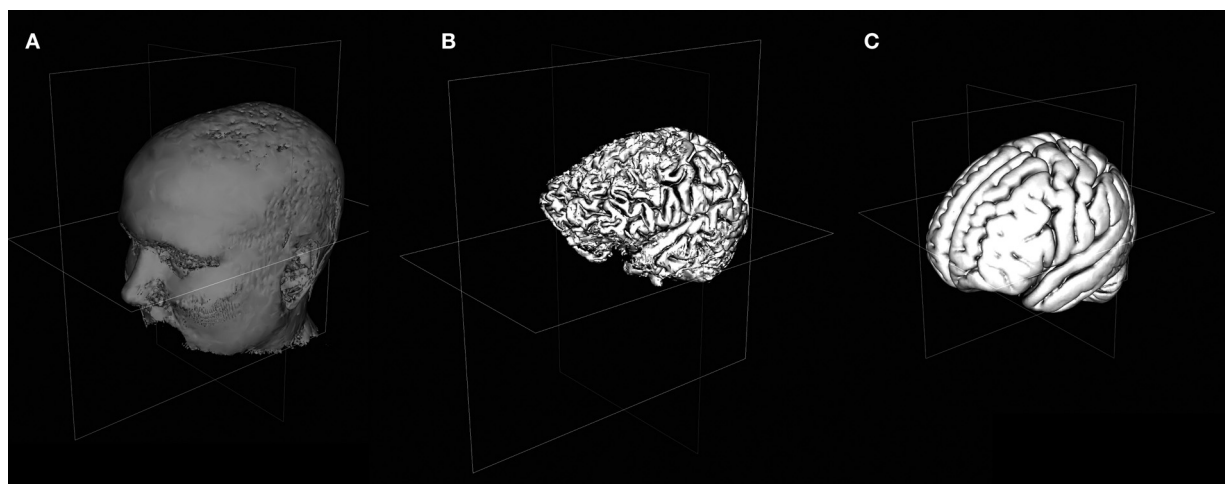


FIGURE 6 | Viewing 3D Surfaces in DV3D. (A) Example of a rough estimate of a scalp using the surface extraction technique. **(B)** Example of a rough estimate of a cortex using the same technique. Here the data set has been skull stripped first using FSL's Brain Extraction Tool. **(C)** A rough cortical extraction of the $1 \times 1 \times 1 \text{ mm}^3$ MNI brain distributed with FSL 4.0.

then be directly mapped through this lookup table to the scalar value at that point. This offers an easy way to map patterns of activation to a surface.

Viewing time-series data

Interactive time-series data visualization is another data exploration technique supported by DV3D. The ability to follow real time changes in signal amplitude at specified locations in data sets relies on VTK's aforementioned ability to map scalar data to individual vertices of loaded surfaces. DV3D extends the ability of VTK to map scalar data by allowing users to pass new values into surface objects' scalar arrays. By allowing users to update the scalar values mapped to surfaces with data from any time point in a time-series, DV3D allows dynamic viewing of time-series data in 2D and 3D by stepping through successive time points. DV3D also supports extraction of sensor time-series data for MEG and EEG data (e.g. Butterfly plots).

Numpy²⁹ is a mathematical methods module for Python that allows, amongst many other mathematical functions, the use and manipulation of arrays and matrix mathematics in Python. Python's automatic memory management, coupled with the power of Numpy matrix manipulations means that DV3D has access to efficient temporary data storage of large data arrays. VTK also offers techniques for data arrays to be passed directly into VTKArray classes, further increasing processing efficiency.

Two time-series objects are shown in **Figure 7**. A 3D contour plot and a minimum norm solution (techniques used for visualizing and analyzing MEG and EEG data) for two MEG data sets are shown in **Figures 7A,B**, respectively. The user first provides a coordinate file that describes the surface that is to be added to the *VTK window*. This file provides the coordinates for the vertices and edges of the surface to be generated. The user then provides a time-data file that holds an array of scalar values. This file holds multiple values for each vertex, arranged chronologically to represent the time-series at

each location or vertex in the coordinate file. Independently of the exact file formats, DV3D generates a surface from the coordinate file, and then loads the time-data file into memory, constructing a Numpy array to hold the time-series data. As the user interacts with the object, stepping to subsequent or previous time points, DV3D simply steps to the appropriate point in the array and extracts the relevant values. These values are then converted to a VTKArray and passed directly to the scalar value representation of the object. Although this process may seem rather complex, it is an extremely efficient technique for managing large data arrays without restricting rendering speed when visualizing time-series data.

Advanced interaction techniques

We have shown the way in which DV3D can load surfaces or generate them from underlying data, or re-slice volume data in real time using image planes. We will now briefly describe three of the more advanced features demonstrated in the user documentation and tutorials to show the data exploration potential of DV3D.

3D overlay data. This visualization technique relies on the previously described method for extracting isosurfaces from MRI volumes using the *vtkContourFilter*. We previously described extracting a rough representation of the cortex by passing a base sMRI volume to the *vtkContourFilter*. Following the same principle, we can pass an overlay volume to the *vtkContourFilter* in the place of the structural volume. This volume could, for example, be a statistical z-score map of the activation resulting from a contrast analysis of fMRI data. This is illustrated with a visual motion fMRI data set in **Figure 8**. The 2D overlay data is shown in **Figure 8A**.

Isocontouring with depth-dependent transparency mapping is a technique that can be applied to a variety of neuroimaging data types or result files. **Figure 8E** shows how this technique can be applied to probabilistic DTI visualization (e.g. FSL's Probtrack³⁰

²⁹<http://numpy.scipy.org/>

³⁰http://www.fmrib.ox.ac.uk/fsl/fdt/fdt_probtrackx.html

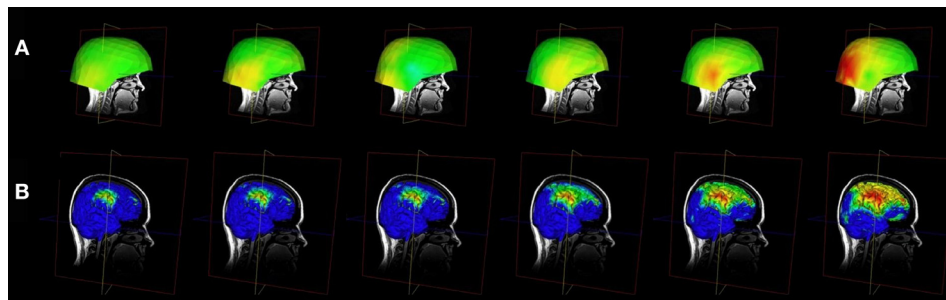


FIGURE 7 | Viewing time-series data in DV3D. (A) Evolution of an MEG field displayed via 3D-contour plot. **(B)** Evolution of a minimum norm projection via surface scalar lookup table. In both instances frames can be automatically generated by cycling data and exported for movie creation.

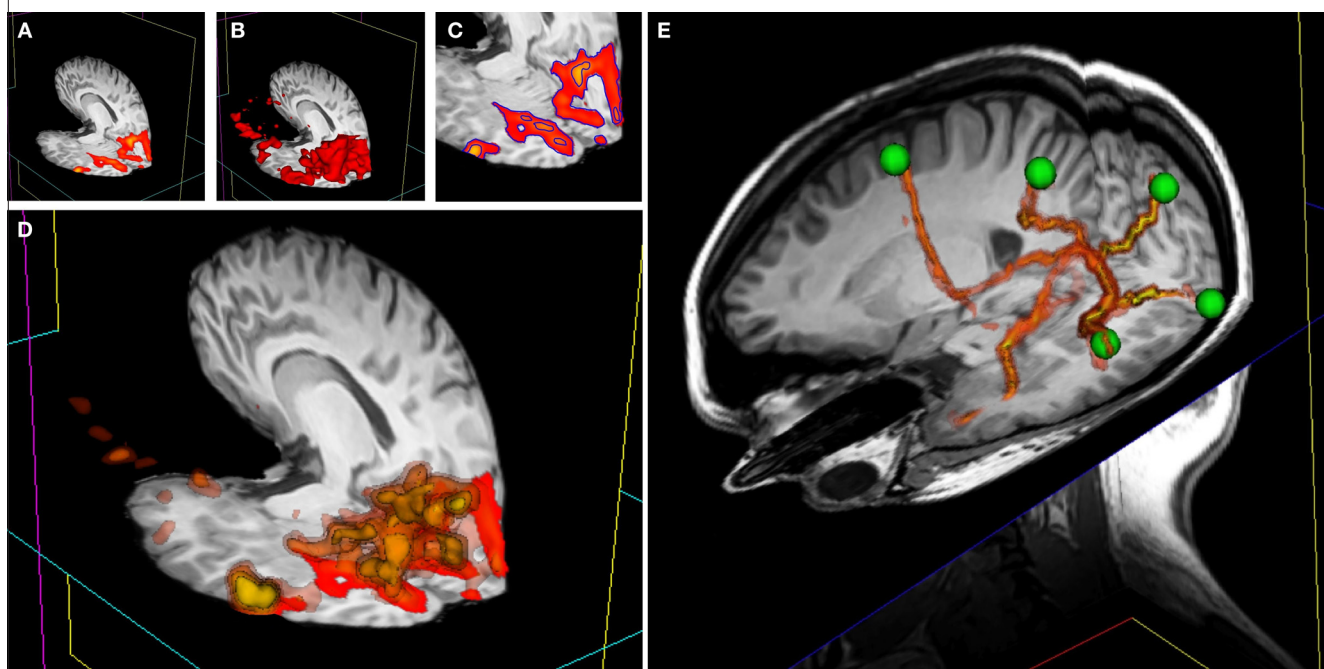


FIGURE 8 | 3D overlay data using isosurface transparency. (A) 2D overlay data from an fMRI experiment overlaid onto a structural MRI volume. **(B)** The vtkContourFilter can be applied to create an isosurface through the data at a specific threshold value, say $z = 2.3$. The returned 3D surfaces will encompass all areas in the data set that have a z-score of $z = 2.3$ or above. We could repeat the process, asking the vtkContourFilter to return smaller surfaces as we increase the threshold. **(C)** A 2D representation (using isocontours shown in blue) of 2 separate isovalues used to extract surfaces. **(D)** If we simultaneously render five sets of surfaces, at z-scores of $z = 2.3, 3.3, 4.3, 5.3$, and 6.3 , for example, the only set of surfaces visible would be that at $z = 2.3$, since all other surfaces are inside this surface. We can manipulate the transparency and color of the vtkPolyData class to make the distribution of

activation visible and overcome this problem. By making the outermost surface (at the lowest threshold value) 80% transparent, the second outermost 60% transparent, the third 40% transparent, the fourth 20% transparent, and the highest threshold surface completely opaque, we make all surfaces simultaneously visible. To emphasize this effect, we can also apply a color gradient (yellow to red) across the surface threshold range. Interacting with this mode of visualization in 3D gives an instantaneous percept of the entire distribution of the activation in 3D. **(E)** This image shows a number of tracts output from FSL's Probtrack toolbox rendered using the 3D overlay technique. The tracts are seen as yellow to red isosurfaces. The green spheres indicate the positions of seed and target points as defined in Probtrack.

output) to give a clear representation of the entire extent of probable connectivity between regions. In addition to being a tool for producing interesting 3D images of the connectivity probability distribution of the DTI data set, this technique has another potential benefit for DTI. Standard DTI fiber tracking techniques tend to represent 3D results at streamlines or stream-tubes in 3D space. With this technique, the colors mapped to each surface have actual

probabilistic value and can be mapped along the length of the tract or network path with a visible color bar.

Surface interrogation of overlay volume data. The vtkContourFilter interrogates data volumes, finding specific scalar values and then extracting the 3D coordinates with corresponding scalar values, constructing isolines or isosurfaces by effectively ‘connecting the

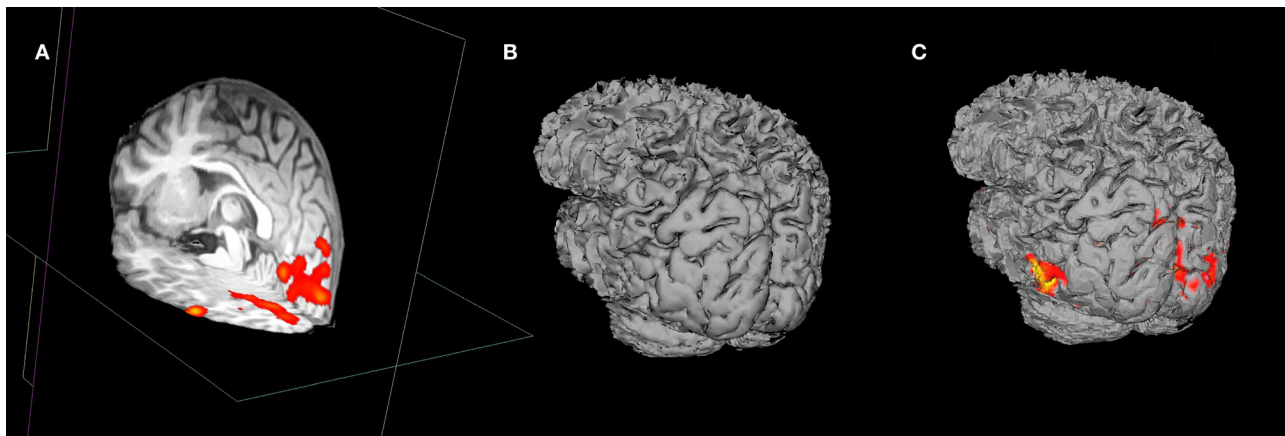


FIGURE 9 | A demonstration of surface interrogation of overlay volume data. (A) Structural MRI space with fMRI data overlay. **(B)** Rough cortical extraction from underlying structural MRI data. **(C)** Rough cortex with overlay intersection data rendered onto the surface at the user defined thresholds.

dots'. VTK also offers techniques to do the reverse: having a surface in the same coordinate space as a data volume, we can find where each vertex of the surface intercepts with the data volume and extract the volume's scalar value at this point. We have already shown (in Figure 7) that when a scalar values are provided for each vertex of a surface, we can use a color lookup table to overlay a color map of the distribution of the scalar value amplitudes across the surface.

Figure 9 demonstrates the usefulness of this technique. An overlay volume can be loaded into sMRI space (Figure 9A). The user can then create or load a surface (Figure 9B) into the same space. From the property panel of this surface the user can choose to map statistical data to the surface (at the current threshold and color map defined by the overlay plane's property set). This gives the user a very quick way to visualize activation distributions in 3D (Figure 9C).

DESIGN OBJECTIVE: COREGISTRATION TO ATLASES

Automatic atlas lookup

DV3D provides methods for real-time cross referencing with brain atlases. Atlas lookups are currently only possible on MRI-overlay mode. Once the user has loaded a base MRI volume, they can load a second volume into memory. On the *Reference* tab of the *Button Panel*, the user can select a file to load as the reference volume to compare to the base volume. Once the user selects a volume, they are prompted to supply a transformation matrix describing the mapping of the base volume (e.g. an individual's brain) to the reference volume (e.g. the MNI brain). DV3D is currently optimized for use with FSL output data, allowing referencing with the MNI and Talairach brains. If a user supplies the MNI brain as a reference, the user can select to automatically lookup the equivalent Talairach coordinates and brain label. DV3D uses the MTT-pooled transform for the MNI brain to the Talairach brain (Lancaster et al., 2007). Coordinates and slice numbers of the current and reference data set are displayed in the *Message Dialog* of the *Main application window*. The Talairach label, slice number and coordinate is displayed in the *Message dialog* if the supplied reference volume is the MNI brain and the user has checked the *Ref is MNI* and *Show*

Talairach Transform check boxes on the *Button Panel*. Interaction with a base MRI volume, with cross referencing to the MNI and Talairach atlas is demonstrated in Figure 4.

DESIGN OBJECTIVE: EXPORT ROUTINES FOR SHARING AND PUBLICATION

Surfaces

Any surface currently displayed in DV3D's VTK window can be written out to a file for sharing or reloading at a later time. Export routines for surfaces can be called by selecting the required surface's label in the *Object List*, clicking on the list item with the right mouse button and selecting the *Export surface* option. This will launch the operating system's native 'Save file as' dialog. The file can then simply be saved and re-loaded where required.

Images

DV3D offers a number of different options for saving out images, capturing the content of the *VTK window* and the *Orthogonal view window* as required. The user has full control over the resolution of the image output and is given the option of multiple output formats (including JPEG, TIFF, BMP and PNG). Controls enable the user to export the current view to single image, or export a sequence of views as separate frames (e.g. 360° rotation of the viewport to multiple, sequential images).

Movies

DV3D offers options for saving and creating movies from of the *VTK window*. The user has full control over the resolution of the image output since the frames of the movie are simply captured at the dimensions of the *VTK window* as it is displayed on the computer monitor. On the *Export tab* of the *Button Panel* the user can select:

- *Export 360°* directly to AVI movie. VTK provides a *vtkAVIWriter* class that is capable of writing renderer contents directly to AVI format video files. Currently this export routine does the same as the *Export 360° to multiple images* routine, rotating the camera through 360° around the object over 180 frames

and creating the output as a movie. Depending on the build options used at VTK installation time or the installer that the user has chosen to use, the `vtkAVIWriter` class is not always automatically compiled. The Enthought Python distribution, for example, builds this class on Windows by default, but not on OSX. Users wanting access to this functionality should consider manual installation of the VTK modules, or see the more advanced functionality of the streaming routine described in *Start interactive streaming*.

- *Start interactive streaming*. This is the most advanced interaction capture technique currently available with DV3D. It has the capability to capture user interactions in real time, periodically capturing frames from the *VTK window* as the user changes objects in it. Clicking the start interactive streaming button launches the operating system's 'Choose folder dialog', allowing the user to specify a folder for the output to be saved in. With this routine, frames are saved to memory as they are captured rather than being written out immediately. The user will notice very little jittering during interaction due to the decreased processing load. The individual frames are then written out when the Stop stream button is pressed. Individual frames can then be combined into a movie format by external software programs such as Apple's QuickTime Pro.

Examples of all export routines are provided at the software website references in the Supplementary Material section of this paper.

DESIGN OBJECTIVE: AN EFFICIENT WORKING ENVIRONMENT

A number of features of DV3D are designed to aid users to optimize the working environment of the package.

User preferences

A user preferences file can be accessed via the Preferences panel. This allows users access to environmental variables including:

- Automatic property panel display: users can choose whether the property panels generated for each loaded object are automatically displayed or not.
- Orthogonal window orientations: these settings allow the user finer control over the layout of the orientations of the Orthogonal view window panels.
- Automatically render orthogonal window: this setting toggles whether the program default is to automatically render the Orthogonal window when the VTK window changes, or whether the user calls this manually.

Parallel processing

Python offers access to parallel processing via a number of different modules. While there is little need for this at present, we have included a sample of how Python can manage separate threads with this release as a demonstration of how easy it is to implement, and how much potential there is for speeding up user interaction. The demonstration can be run from the *Threading* tab on the *Button Panel*. This function runs the load routine for a surface file with over one million vertices. The routine is run in the background while the user continues to interact with the program. Loading the same surface without threading requires the user to wait between 20 and

45 s for the process to complete. An example of the simplicity of the code required to access this functionality is shown in **Figure 10**.

Workspace saving

At any point during use of DV3D, users can choose to save the current status of the workspace to a file. This file holds metadata that can be loaded at the start of a later session to load the current working environment, with many of the current settings in use by the user, including all loaded objects and color / transparency settings. This file hard-codes the paths of input files and will fail if files are moved between sessions.

Surface decimation

Upon loading surfaces into memory, DV3D can be set to run a decimation routine to down-sample the number of vertices of each surface by between 10 and 90%. This surface is not shown automatically (the high-resolution surface is visible by default), but the user can choose to toggle between the decimated and original surface during interaction to help increase the speed of rendering.

Command line access for scripting

In addition to handling workspace files, DV3D offers the ability to handle explicit arguments passed to the program on the command line. This allows users access to advanced scripting options for automation of processing streams.

```
A def OnButtonClick(self, event):
    my_file = ChooseFile(self, '*.off')
    Load_surface_file(my_file, parent_frame)
    print 'file loaded ... continue'

B import threading
    class RunFunctionInThread( threading.Thread):
        def run(self):
            my_file = ChooseFile(self, '*.off')
            Load_surface_file(my_file, parent_frame)

    self.my_thread = RunFunctionInThread()

    def OnButtonClick(self, event):
        self.my_thread.start()
        print 'file loading in background ... continue'
```

FIGURE 10 | A demonstration of code simplicity in Python: enabling threading. (A) This code example demonstrates how a function may be linked to a button press in a standard Python script using the thread running the main program. On the button click, the program asks the user to choose a file to load. The program then passes the file to the subroutine (`Load_surface_file`) and runs the subroutine. While the subroutine is running the user has to wait for the object to be loaded and returned to the main program before continuing. (B) This second code example shows that we can produce the same result using Python's threading module. First the threading module is imported. The functionality of code example in (A) is then added as a function (`RunFunctionInThread`). The button click in this instance calls a thread (`my_thread.start()`) and runs the load routine will run in the background allowing the user to continue working while it is prepared. Note that threading only requires a few extra lines of simple code.

DESIGN OBJECTIVE: A FLEXIBLE, SCALABLE AND ACCESSIBLE OPEN-SOURCE FRAMEWORK

Our implementation of a flexible, scalable and accessible open-source framework is described largely in the Section ‘Methods: Implementing a Python Framework’ of this paper. We show that the combination of Python, wxWidgets and VTK gives us the ability to produce a code base that is freely distributable and platform independent. This implementation has all the functionality required to process a number of different file types and formats, is highly modularized for ease of understanding and promotes future user development due to the relative simplicity of Python as a programming language (for an example, see **Figure 10**).

DISCUSSION

The ‘Results’ Section of this paper shows that DV3D satisfies each of the key design objectives identified as important for a multi-modal neuroimaging data visualization package. In summary,

DV3D allows users to view data from many different imaging modalities and analysis streams in a single coordinate space. Data can be cross-referenced with standard spaces in real-time, from 2D or 3D objects. DV3D supports the display of a large number of input data formats, and allows the user to export data in a number of different formats. The user workspace can be customized to allow optimum productivity and allows access for both casual and power users (command line scripting and parallelization). DV3D’s platform independence (due to Python) makes it flexible, and the modularity and simplicity of the code base makes it both accessible and scalable.

Readers may ask about the novelty of DV3D. While we (to the best of our knowledge) are unaware of any other software package that utilizes isocontouring with depth-dependent transparency mapping to display 3D statistical overlays (see Advanced Interaction Techniques), we do not claim that any other techniques utilized by DV3D are novel. **Table 1** summarizes the features of DV3D,

Table 1 | Feature summary and comparison of imaging data visualization packages. This table summarizes some of DV3D’s key features and compares DV3D’s functionality with three commonly used imaging data visualization tools, FSLView, MRIcron and 3D Slicer. Features are accurate as at the time of initial development of DV3D.

Software feature	FSLView	MRIcron	3D Slicer	DV3D
NEUROIMAGING DATA SUPPORT				
Optimised for neuroimaging	✓	✓	–	✓
Structural MRI	✓	✓	✓	✓
Functional MRI	✓	✓	✓	✓
DTI – probabilistic	✓	–	–	✓
DTI – tractography	–	–	Calculated online	Loaded from memory
DTI – 2d vectors	✓	–	–	✓
DTI – 3d vectors	–	–	✓	✓
MEG/EEG contour plots (2D and/or 3D)	–	–	–	✓
MEG/EEG 3d time-series on surface	–	Single instant	–	Full dynamic
MEG/EEG dipoles	–	–	–	✓
MEG/EEG butterfly plots	–	–	–	✓
DATA EXPLORATION				
2D statistical map overlay	✓	✓	✓	✓
3D statistical map overlay	✓	–	–	✓
Interactive surface extraction	–	–	Complex watershed	Simple isosurfaces
Real-time atlas cross-referencing	If data in MNI space	–	–	4 × 4 Transform required
COMPLEX VISUALIZATION FUNCTIONS				
Real-time reformatting	–	–	Single plane	Multiple planes
Interactive data intersection	–	–	–	✓
Interactive time-series interrogation	2d fMRI only	–	–	2D and 3D fMRI, EEG and MEG
Batch processing from command line	✓	–	–	✓
EXPORT				
Static images	–	✓	✓	✓
Movies	–	–	✓	✓
Real-time streaming	–	–	–	✓
TECHNICAL				
Main code base language	C,C++,Tcl/Tk	Pascal	C++,Tcl/Tk	Python
Platform independent code base	–	–	–	✓
Access to parallel processing	–	–	–	✓

comparing the resulting functionality achieved by DV3D with similar packages already available. We show that, while DV3D is not an entirely comprehensive solution for visualizing neuroimaging data, it does represent a utility that can offer a single solution to users of a variety of neuroimaging analysis packages. Being optimized for neuroimaging data, this single package offers more options to researchers interested in multi-modal neuroimaging data analysis than any alternative stand-alone visualization package.

While visualization packages are primarily used to display the results output by analysis packages, many visualization tools have developed to include techniques to physically manipulate loaded results files with complex analytical algorithms. 3D Slicer, for example, utilizes complex segmentation algorithms to allow tissue segmentation from any MRI volume acquired at any part of the body. This allows 3D Slicer to be regarded as a tool that is suited to generalized medical imaging analysis and visualization rather than being neuroscience specific. When handling neuroimaging data, 3D Slicer is also more analytically driven than MRICron or DV3D. 3D Slicer does not load fiber-tracking results from external analysis packages. Rather it analyzes diffusion-weighted MRI data to calculate fiber tracts³¹. This move away from being a pure visualization tool, specific for neuroimaging data, does mean that 3D Slicer has more demanding development and maintenance overhead and can take longer to become familiar with, compared to MRICron or DV3D.

DV3D was designed to be a tool optimized for the visualization of neuroimaging data and not an analysis tool *per se*. Although many algorithms and calculations underlie the functionality of DV3D, they are primarily image processing functions allowing VTK to display results of analyses conducted in other software packages. If DV3D were solely a data visualization tool, it would simply take user input and display it in its raw format. We have shown however that DV3D offers routines for manipulating loaded data to add value to the visualization environment: DV3D can average raw MEG time series data by epoch and display this average as a contour plot; DV3D can manipulate volume grid data and extract and interpolate 3D surfaces from this data to display isosurfaces and isovolumes; DV3D offers the ability to decimate large surface data sets to increase rendering speed. DV3D has thus already begun to evolve from a pure visualization tool to a tool that allows users to interact with their data. DV3D does not, however, lose focus of its optimization for neuroimaging data processing.

Since DV3D has the potential to be more than a visualization tool, we have considered extending its functionality. Including more functions in DV3D will allow a more extensive range of tools for users to interrogate data. The modularity of the framework and platform independence of the code base allows access for rapid development and extension to include additional file format support and processing routine extension. Many functions have already been requested by interested parties and are under current consideration for inclusion in subsequent releases. Python offers modules for handling pipes on operating systems, allowing the potential for system calls and data exchange between system processes. We are currently exploring the capability to include calls to DV3D

to/from a number of packages. Other examples of user requests currently under development include the ability to align volumes and/or surfaces manually or with automated error-minimization routines, and functions to measure distances, areas, and volume size between/on displayed objects. Future development of DV3D will focus on support for additional formats, increased automation of processing streams, extended local settings customization, and more extensive data sharing options. We will also consider including the GIFTI format as a surface export option due to the significant increases in performance reported when handling these files relative to the *vtk* format (Harwell et al., 2008).

Python has a large and diverse international user base, and promotes the development of increasingly accessible and comprehensive solutions for current computing and analysis requirements. The use of Python as the base for DV3D allows a cross-platform, transparent, and extendible code base for user development. By using Python to wrap existing toolkits, including tools for visualization, rendering, parallelization and GUI generation, DV3D development has required minimal new code to be written to solve complex computations. In addition to the functionality DV3D currently offers, DV3D can also be easily expanded to meet users' changing needs because of its modular, open-source design. DV3D's framework is intentionally modularized to provide concise working examples, illustrating the power of VTK and how easily this power can be harnessed by Python. While the authors are keen to extend the package, provision of an open-source package is intended to stimulate and facilitate further development of the software by the user community. Example code illustrating the extension of the functionality of the package is provided for users interested in contributing code or developing the package for their own purposes. DV3D's code base currently consists of circa 12,000 lines of Python code. 3D Slicer has over 550,000 lines of C++ code, although this includes a large amount of additional analytical functionality that DV3D does not have. We suggest that the simplicity of Python relative to C++, and the vastly smaller code base, make DV3D more accessible in terms of community extension and development prospects.

DV3D's primary function is to allow easy, interactive display of multi-modal neuroimaging data. DV3D has been successfully implemented on many platforms and is currently used by local users from a variety of disciplines. DV3D is provided as a free, open-source package built on Python's platform independent model. DV3D can thus be used and, more importantly, developed by the wider neuroimaging community.

ACKNOWLEDGMENTS

The authors would like to acknowledge the developers of Python, VTK and wxWidgets for their ongoing support of open-source software provision. The reviewers are to be thanked for their insightful comments, some of which have already resulted in additional functionality being incorporated into the package.

SUPPLEMENTARY MATERIAL

DOWNLOADING THE SOFTWARE, EXAMPLES AND EDUCATIONAL RESOURCES

DV3D, examples output and input files and interactive user tutorials can be freely downloaded from <http://www.ynic.york.ac.uk/software/dv3d>.

³¹<http://www.slicer.org/slicerWiki/index.php/Slicer3:DTMRI>

REFERENCES

- Ashburner, J., Andersson, J., and Friston, K. J. (1999). High-dimensional nonlinear image registration using symmetric priors. *NeuroImage* 9, 619–628.
- Coltheart, M. (2006). What has functional neuroimaging told us about the mind (so far)? *Cortex* 42, 323–331.
- Cox, R. W. (1996). AFNI: software for analysis and visualization of functional magnetic resonance neuroimages. *Comput. Biomed. Res.* 29, 162–173.
- Delorme, A., and Makeig, A. (2004). EEGLAB: an open-source toolbox for analysis of single-trial EEG dynamics. *J. Neurosci. Methods* 134, 9–21.
- Frackowiak, R. S. J., Friston, K. J., Frith, C. D., Dolan, R. J., and Mazziotta, J. C. (1997). *Human Brain Function*. San Diego, Academic Press.
- Harwell, J., Bremen, H., Coulon, O., Dierker, D., Reynolds, R. C., Silva, C., Teich, K., Van Essen, D. C., Warfield, S. K., and Saad, Z. S. (2008). GIFTI: Geometry Data Format for Exchange of Surface-Based Brain Mapping Data. *OHBM – Poster Presentation*
- Jenkinson, M., Bannister, P. R., Brady, J. M., and Smith, S. M. (2002). Improved optimisation for the robust and accurate linear registration and motion correction of brain images. *NeuroImage* 17, 825–841.
- Lancaster, J. L., Tordesillas-Gutiérrez, D., Martinez, M., Salinas, F., Evans, A., Zilles, K., Mazziotta, J., and Fox, P. T. (2007). Bias between MNI and Talairach coordinates analyzed using the ICBM-152 brain template. *Hum. Brain Mapp.* 28, 1194–1205.
- Larsson, J. (2001). *Imaging Vision: Functional Mapping of Intermediate Visual Processes in Man*. Ph.D. thesis, Karolinska Institute, Stockholm.
- Liu, Z., Kecman, F., and Bin, H. (2006). Effects of fMRI–EEG mismatches in cortical current density estimation integrating fMRI and EEG: A simulation study. *Clin. Neurophysiol.* 117, 1610–1622.
- Mackenzie-Graham, A. J., Van Horn, J. D., Woods, R. P., Crawford, K. L., and Toga, A. W. (2008). Provenance in neuroimaging. *NeuroImage* 42, 178–195.
- Mazziotta, J., Toga, A., Evans, A., Fox, P., Lancaster, J., Zilles, K., Simpson, G., Woods, R., Paus, T., Pike, B. et al. (2001). A four-dimensional atlas of the human brain. *J. Am. Med. Inform. Assoc.* 8, 401–430.
- McDonald, C. R. (2008). The use of neuroimaging to study behavior in patients with epilepsy. *Epilepsy Behav.* 12, 600–611.
- Stufflebeam, S. M., and Rosen, B. R. (2007). Mapping cognitive function. *Neuroimaging Clin. N. Am.* 17, 469–484.
- Talairach, J., and Tournoux, P. (1988). *Coplanar Stereotaxic Atlas of the Human Brain: 3-Dimensional Proportional System – An Approach to Cerebral Imaging*. New York, Thieme Medical Publishers.
- Teo, P. C., Sapiro, G., and Wandell, B. A. (1997). Creating connected representations of cortical gray matter for functional MRI visualization. *IEEE Trans. Med. Imaging* 16, 852–863.
- Wakana, S., Jiang, H., Nagae-Poetscher, M., van Zijl, P. C. M., and Mori, S. (2004). A fiber-tract based atlas of Human white matter anatomy. *Radiology* 230, 77–87.
- Wandell, B. A., Chial S., and Backus, B. (2000). Visualization and measurement of the cortical surface. *J. Cogn. Neurosci.* 12, 739–752.

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

Received: 12 September 2008; paper pending published: 25 October 2008; accepted: 05 March 2009; published online: 27 March 2009.

Citation: Gouws A, Woods W, Millman R, Morland A and Green G (2009) DataViewer3D: an open-source, cross-platform multi-modal neuroimaging data visualization tool. *Front. Neuroinform.* (2009) 3:9. doi: 10.3389/neuro.11.009.2009
Copyright © 2009 Gouws, Woods, Millman, Morland and Green. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components

James A. Bednar*

Institute for Adaptive and Neural Computation, University of Edinburgh, Edinburgh, UK

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Stephen Eglan, University of
Cambridge, UK
Marc-Oliver Gewaltig, Honda Research
Institute Europe GmbH, Germany

***Correspondence:**

James A. Bednar, Institute for Adaptive
and Neural Computation, University of
Edinburgh, 10 Crichton Street,
Edinburgh, EH8 9AB, UK.
e-mail: jbednar@inf.ed.ac.uk

Many neural regions are arranged into two-dimensional topographic maps, such as the retinotopic maps in mammalian visual cortex. Computational simulations have led to valuable insights about how cortical topography develops and functions, but further progress has been hindered by the lack of appropriate tools. It has been particularly difficult to bridge across levels of detail, because simulators are typically geared to a specific level, while interfacing between simulators has been a major technical challenge. In this paper, we show that the Python-based Topographica simulator makes it straightforward to build systems that cross levels of analysis, as well as providing a common framework for evaluating and comparing models implemented in other simulators. These results rely on the general-purpose abstractions around which Topographica is designed, along with the Python interfaces becoming available for many simulators. In particular, we present a detailed, general-purpose example of how to wrap an external spiking PyNN/NEST simulation as a Topographica component using only a dozen lines of Python code, making it possible to use any of the extensive input presentation, analysis, and plotting tools of Topographica. Additional examples show how to interface easily with models in other types of simulators. Researchers simulating topographic maps externally should consider using Topographica's analysis tools (such as preference map, receptive field, or tuning curve measurement) to compare results consistently, and for connecting models at different levels. This seamless interoperability will help neuroscientists and computational scientists to work together to understand how neurons in topographic maps organize and operate.

Keywords: Python, simulators, interoperability, interfacing, topographic maps, large-scale, cortex, visual

INTRODUCTION

In mammals, much of the cortical surface (and many subcortical structures) can be partitioned into topographic maps (Kaas, 1997; Van Essen et al., 2001). These maps contain systematic two-dimensional representations of features relevant to sensory and motor processing, such as retinal position, sound frequency, line orientation, and motion direction (Blasdel, 1992; Merzenich et al., 1975; Ohki et al., 2005; Weliky et al., 1996; Xu et al., 2007). **Figure 1** shows an example retinotopic and orientation map from the primary visual cortex (V1). Understanding the development and function of topographic maps is crucial for understanding brain function, and will require integrating large-scale experimental imaging results with single-unit studies of the individual neurons and their connections that make up these maps. In principle, computational modeling can help make these links explicit, in order to explain how topographic maps can emerge from the behavior of single neurons.

However, existing simulators typically address only a small range of levels of analysis. For instance, NEURON (Hines and Carnevale, 1997) and GENESIS (Bower and Beeman, 1998) primarily focus on detailed studies of individual neurons or very small networks of them, rather than enough neurons to form a meaningful topographic map. Topographica (Bednar, 2008) and NEST (Diesmann and Gewaltig, 2002) allow much larger scale simulations of simpler neurons, but Topographica provides only limited support for

spiking neurons, while NEST provides only limited support for firing-rate neurons (necessary for the largest scale models) or for more detailed individual neuron models, and does not provide a GUI for large-scale visualizations. Combining multiple simulators to bridge between these levels of analysis could provide a complete, biologically grounded explanation of how single-neuron properties lead to large-scale topographic maps. Even for models at the same level, interfacing multiple simulators into a coherent framework can also help provide a uniform means for comparing and evaluating them. However, interconnecting simulators has previously been a significant technical challenge (Cannon et al., 2007; Djurfeldt and Lansner, 2007).

This paper describes how the Topographica map-level simulator can be used to achieve important types of interoperability between a very wide range of simulators with surprisingly little coding or development effort. One reason that interoperability is practical in Topographica is that Topographica is implemented in the Python scripting language, and many neural simulators now include Python interfaces. Another reason is that Python is a very high level language, known as a *glue language* (Ousterhout, 1998), that makes it easy to connect different interfaces for rapid software development. Even more important, however, is that Topographica is built around a high-level abstraction of the properties of topographic maps, which is relatively simple to adapt to components implemented in any particular simulator yet provides access to a

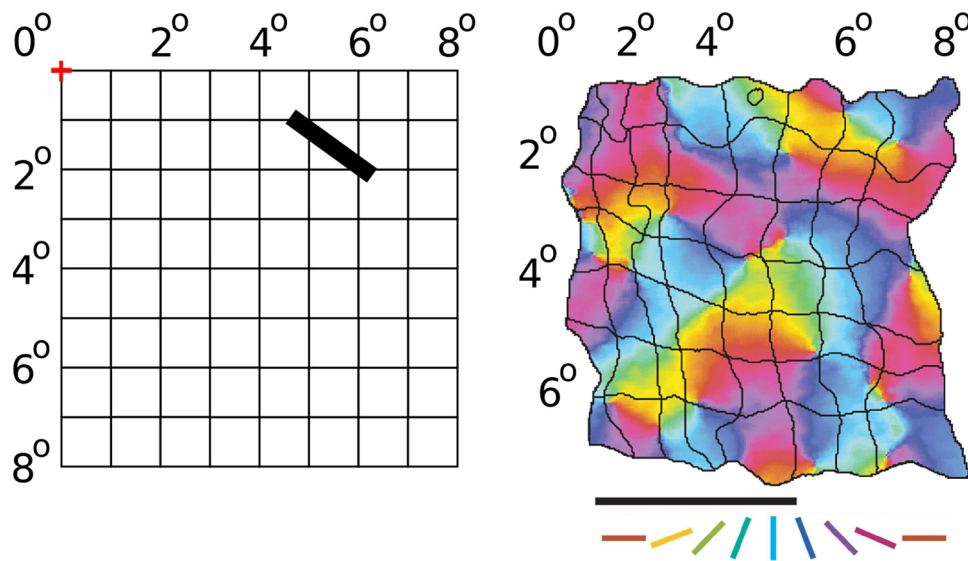


FIGURE 1 | Retinotopic and orientation map in V1. Given a particular fixation point (marked with a red + symbol above), the visual field seen by an animal can be divided into a regular grid, with each square representing a $1^\circ \times 1^\circ$ area of visual space. In cortical area V1 of mammals, neurons are arranged into a retinotopic map, with nearby neurons responding to nearby areas of the retina. As an example, the image on the right shows the retinotopic map on the surface of V1 of a tree shrew for an $8^\circ \times 7^\circ$ area of visual space (adapted from Bosking et al., 2002 with permission; scale bar is 1 mm). A stimulus presented in a particular location in visual space (such as the thick black bar shown) evokes a response centered around the corresponding grid square in V1 ($6^\circ, 2^\circ$). Which

specific neurons respond within that general area, however, depends on the orientation of the stimulus. The V1 map is color coded with the preferred orientation of neurons in each location; e.g. the black bar shown at left will primarily activate neurons colored in purple in the corresponding V1 grid squares. Similar maps could be plotted for this same area showing preference for other visual features, such as motion direction, spatial frequency, color, disparity, and eye preference (depending on species). Other cortical areas are arranged into topographic maps for other sensory modalities, such as touch and audition, and for motor outputs. Topographica is designed to simulate any of these cortical or subcortical areas.

large range of useful tools. Simply put, if a simulation in any other simulator or language contains a large number of neurons (at any level of complexity) arranged into a two-dimensional sheet or array (or a three-dimensional stack of such two-dimensional arrays), then it will be practical to use that simulation or parts of it within Topographica.

In turn, integrating such a simulation into Topographica will be *useful* if it can make use of analyses that rely primarily on an average (firing rate) activation level for each neuron, particularly if they are based on measuring responses to an input pattern. Many such routines are already implemented in Topographica, such as measuring receptive fields, tuning curves, or feature preference maps of any type, decoding activity values, and 1D, 2D, or 3D plotting of these and other measurements. Other simulators implement some of these functions, but rarely in a fully general form that can be applied to any neural area and any type of input feature. To make the most use of these components, it is helpful if each sheet of neurons in the underlying model can be separated from the others with well-defined interfaces, but even relatively monolithic models can be analyzed if they include at least one sheet of neurons that can accept an external input, and at least one neuron or set of neurons whose firing-rate activity patterns are of interest. Any such model can then be compared and tested against any similar model, using a consistent analysis and visualization framework. Similar considerations apply to using small parts of external models, such as a model retinal or cortical area, as part of a larger hierarchical or network model of a neural system connected in Topographica.

These features make it surprisingly straightforward to use Topographica for simulating and analyzing large-scale, detailed models of topographic maps, using either native or externally implemented components. Topographica is an open source project, and binaries and source code are freely available through the internet at topographica.org for interfacing to external code on Linux, Microsoft Windows, and Macintosh OS X platforms. In the sections below, we describe the main assumptions and abstractions used by Topographica, provide a detailed example of interfacing to an external spiking simulator, show how to interface to a wide variety of other external systems and simulators, and discuss in more detail which types of models are most suitable for interfacing with Topographica.

SOFTWARE DESCRIPTION AND METHODS

Models supported natively by Topographica typically consist of a collection of topographic maps in cortical or subcortical regions, such as an auditory or visual processing pathway. **Figure 2** shows an example simulation along with various types of analysis and plotting. This simple model consists of four separate populations of neurons, called *Sheets*: one sheet of retinal photoreceptors (labeled *Retina*), a sheet of ON retinal ganglion cell (RGC)/lateral geniculate nucleus (LGN) cells labeled *LGNON*, a sheet of OFF cells labeled *LGNOFF*, and a sheet of V1 pyramidal cells labeled *V1*. Neurons in each sheet are arranged topographically, with similar properties but at different spatial locations.

Topographica is a general-purpose discrete-event simulator, simulating a set of *EventProcessors* (any object in a *Simulation*

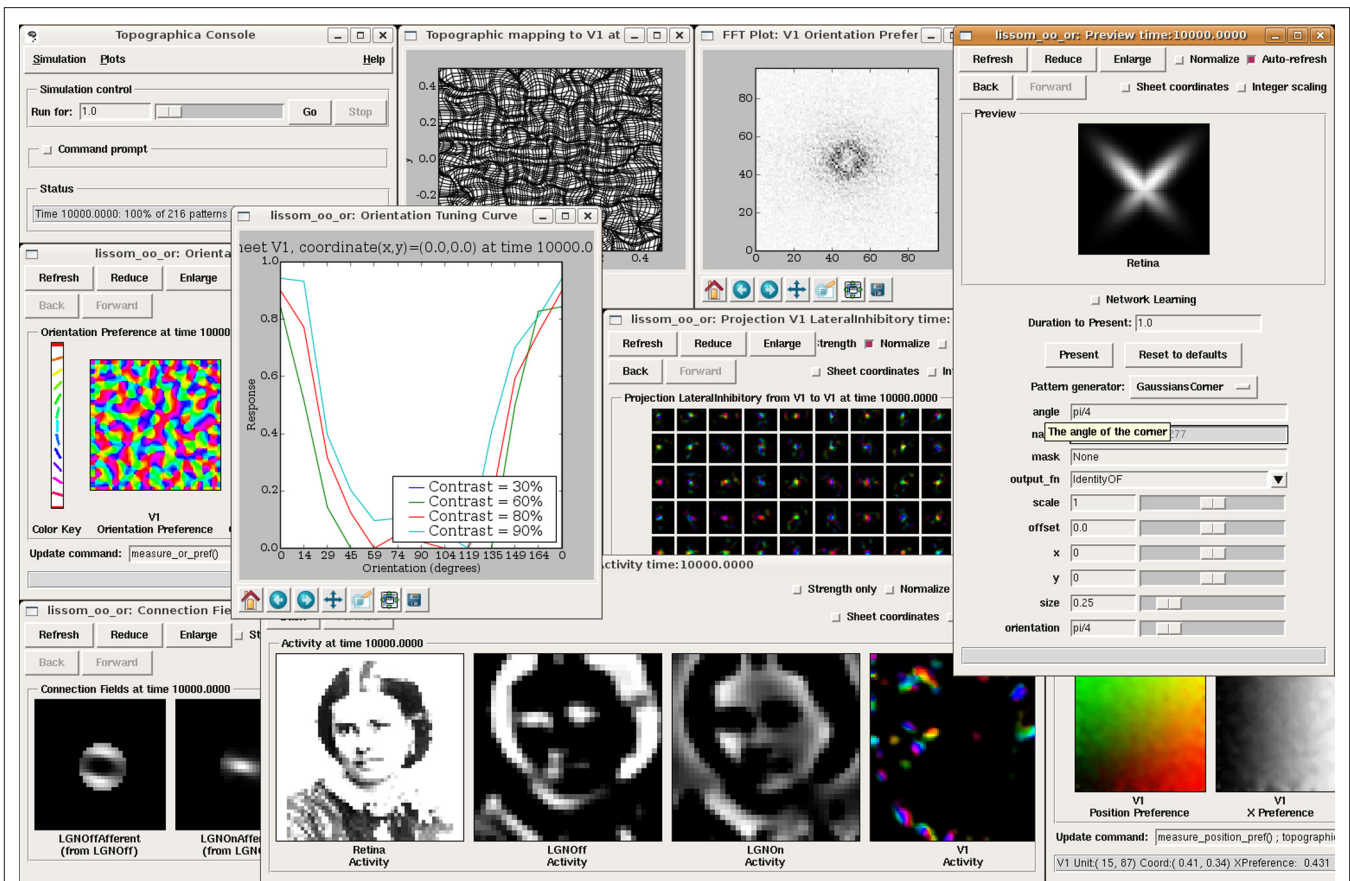


FIGURE 2 | Topographica software screenshot. This image shows a sample session from Topographica version 0.9.3, available freely at topographica.org. Here the user is studying the behavior of an orientation map in the primary visual cortex (V1), using a model of photoreceptors as the input to the Retina, ON and OFF RGC/LGN cells, and a simple V1 model. The window at the left labeled "Orientation Preference" shows a self-organized orientation map in V1. The window labeled "Activity" shows (from left to right) a sample visual image input to the retina, the ON and OFF channel responses to that input, and (on the right) an orientation-color-coded representation of activity in the V1 Sheet of neurons. The input patterns were generated using the Test Pattern "Preview"

dialog at the right. The window labeled "Connection Fields" shows the strengths of the connections to one neuron in V1. The lateral weights for a 9×9 sampling of the V1 neurons are shown in the "Weights Array" window in the center; neurons tend to connect to their immediate neighbors and to distant neurons of the same orientation. The "Topographic Mapping" window shows how retinotopy has been distorted by the orientation map, and the "FFT Plot" shows that the orientation map repeats regularly in all dimensions, as in animals. This type of large-scale analysis is difficult with other simulators, but typically requires no new coding or software development once a network simulation has a basic connection to Topographica.

capable of receiving and sending Events) connected into a graph by EPConnections. An EPConnection ensures that Events are delivered to the appropriate target after a specified delay. The pattern of connections and delays in a certain network determines how a simulation will progress, with events being generated at a certain EventProcessor, processed by the target EventProcessor, and potentially leading to additional Events delivered to other EventProcessors. Of course, any pattern of connection is allowed, including lateral and feedback connections. This approach is general enough to simulate any physical system as a collection of interconnected entities that can interact and change over time.

To make it practical to model large-scale topographic maps, the most common type of EventProcessor in Topographica is a two-dimensional Sheet of neurons as in the example above, rather than a neuron or a part of a neuron. Each Sheet is typically a population of similar neurons, and multiple Sheets can be used for each neural area, e.g. to represent different laminae or qualitatively different cell

classes. Conceptually, a sheet is a continuous, two-dimensional area (as in Amari, 1980; Roque Da Silva Filho, 1992), which is typically approximated by a finite array of neurons. This approach is crucial to the simulator design, because it allows user parameters, model specifications, and interfaces to be independent of the details of how each Sheet is implemented.

Apart from accepting and generating Events, all a Sheet is required to do is to have a fixed area and density of neurons, and to be able to generate a floating-point array of the appropriate size when asked for its current pattern of activity. Once this activity matrix is available for a new Sheet type, then nearly all of Topographica's analysis and plotting code can be used with the new Sheet type, e.g. to decode neural responses from the firing rate, or to measure a topographic map. This general-purpose interface is what makes it practical to wrap around a wide variety of external simulations, as long as they can be interpreted as a two-dimensional array whose elements can have some average firing-rate activity value.

Topographica comes with a variety of Sheet types, plus a large library of other simulation objects, such as projections (EPConnections between Sheets), activation functions, learning rules, analysis routines, and visualizations. The most extensive support is for models of the visual system, and Topographica includes flexible components for generating visual inputs (based on geometric patterns, mathematical functions, and photographic images), plus general-purpose mechanisms for measuring maps of visual stimulus preference, such as orientation, ocular dominance, motion direction, and spatial frequency maps. But many of the primitives are usable for any topographically organized system, and there are already Topographica models of somatosensory areas (e.g. monkey skin and rat whisker barrel areas), auditory inputs, and motor areas (e.g. for driving visual saccades). Moreover, additional components can be added easily to make external simulations visible from within Topographica, or to implement new functionality in general.

INTEROPERABILITY

To demonstrate concretely the procedure for connecting external simulations to Topographica, in this section we present a detailed example of wrapping an external NEST simulation using the Topographica Sheet interface. Shorter examples of how to interface with a variety of other simulators follow.

INTERFACING TO PERRINET RETINAL MODEL IN PyNN

For this example, we wrapped a spiking retinal ganglion cell model that is being developed by Laurent Perrinet (INCM/CNRS) as part of the FACETS project¹ and being used in a large-scale spiking model of cortical columns in V1 (Kremkow et al., 2007). Writing this interface was surprisingly simple, taking about 2 h to adapt one of the example Topographica simulations to send output to an external simulator and retrieve input from it, and we expect interfacing to other models to be similarly straightforward if they meet the assumptions laid out in the “Discussion” section.

The Perrinet retina model is specified in PyNN (Davison et al., 2007)², a Python wrapper that sets up and runs simulations of neural models relatively independently of the underlying simulation engine. This particular script calls the NEST simulator, which is well adapted for large-scale spiking neural networks (Diesmann and Gewaltig, 2002), but it could also be run under NEURON by changing one line of declaration.

The model contains two populations of spiking retinal ganglion cells, a 32×32 array of ON cells and a 32×32 array of OFF cells, receiving input from a 32×32 array of photoreceptors whose activation level can be controlled externally. The code can be obtained and run by downloading Topographica release 0.9.6 (or SVN version 9857 or later) of Topographica, and installing PyNN, NEST, and PyNEST using Topographica’s copy of Python (as described in examples/perrinet_retina.py in the distribution).

Figure 3 shows the Python code for wrapping this network as a Photoreceptor Sheet (Photoreceptors), a connection to PyNN (PyNNR), and two ganglion cell Sheets (ON_RGC and OFF_RGC), and **Figure 4** shows the resulting simulation running in Topographica. The example code would be nearly the same for interfacing to any

other external simulation that consists of two-dimensional arrays of neurons, and so we will step through each part of this code to show how the interface is achieved. In each case, the relevant line of code is marked with a circled number, which can be found on the code listing. Note that this code constitutes the complete, runnable model specification for Topographica; it is not a code excerpt or a high-level interface to some underlying, complicated interfacing code, but instead it is all that was required to connect to and run the external simulation within Topographica.

- ① First, the external simulation is imported, making anything available to Python from that simulation also available to Topographica. For this import to succeed, PyNN, NEST, and PyNEST need to be installed, and each need to have been given Topographica’s copy of Python during installation so that they will be available to Topographica.
- ② Next, we define a new type of Topographica EventProcessor PyNNRetina to handle communication between Topographica and the external simulator. This class simply accepts an incoming event from Topographica that contains a matrix of photoreceptor activity, passes the matrix to the external spiking simulator, collects the firing-rate-averaged results, and sends them out to any Topographica sheets that may be connected.
- ③ More specifically, the class first declares that it can accept an incoming event on a port labeled Activity, and that it will generate two separate types of output data to be made available on the ONActivity and OFFActivity dest_ports. It also declares that it has two user-controlled parameters, N (size of array of neurons) and simtime (duration to run the simulation for each input). (Additional parameters from the underlying simulator can be declared similarly, or all of the underlying parameters could be exposed as a batch using suitable gluing code.)
- ④ The constructor (__init__) does any initialization that should be done once per run, here consisting only of defining some parameters, but potentially including launching an external simulator, making a connection to a remote simulator already running, etc.
- ⑤ The input_event method is called by Topographica whenever an Event delivers data to this object’s src_port (Activity). In this case, the method adds the incoming activity matrix into its parameters data structure (ps), and then calls the external function run_retina to run the underlying simulation. When the external simulator completes, two lists of spikes are returned, one for ON and one for OFF, and these are processed using the helper function process_spike_list. For each list, process_spike_list computes the firing rate of each neuron and sends the resulting floating-point arrays out the appropriate port.
- ⑥ The remainder of the code instantiates a model network to display the results from this class, defining one PyNNR object, a Photoreceptors Sheet to generate input patterns, two RGC Sheets to display the resulting activity patterns, and connections between them.

Running this model (or other Python-based simulations) within Topographica adds only a tiny amount of computational cost. For this example running on a 3GHz Intel Core 2 Duo machine,

¹<http://facets.kip.uni-heidelberg.de>.

²<http://neuralensemble.org/trac/PyNN>.

```

import numpy
from topo import sheet, numbergen, pattern, param, projection
from topo.base.simulation import EventProcessor
① import perrinet_retina_pynest as pynr

② class PyNNRetina(EventProcessor):
③     dest_ports=["Activity"]
        src_ports=["ONActivity","OFFActivity"]
        N = param.Number(default=8,bounds=(0,None), doc="Network width")
        simtime = param.Number(default=4000*0.1,bounds=(0,None),
                                doc="Duration to simulate for each input")

        def __init__(self,**params):
            super(PyNNRetina,self).__init__(**params)
④         self.ps=pynr.retina_default()
            self.ps.update("N":self.N)
            self.dt=self.ps["dt"]

⑤         def input_event(self, conn, data):
                self.ps.update("simtime":self.simtime)
                self.ps.update("amplitude":.10*data)
                on_list,off_list=pynr.run_retina(self.ps)
                self.process_spikelist(on_list,"ONActivity")
                self.process_spikelist(off_list,"OFFActivity")

                def process_spikelist(self,spikelist,port):
                    spikes=numpy.array(spikelist)
                    spike_time=numpy.cumsum(spikes[:,0]) * self.dt
                    spike_out=pynr.spikelist2spikematrix(
                        spikes,self.N,self.simtime/self.dt,self.dt)
                    self.send_output(src_port=port,data=spike_out)

⑥ N=32
    topo.sim["PyNNR"]=PyNNRetina(N=N)

    topo.sim["Photoreceptors"]=sheet.GeneratorSheet(
        nominal_density=N, period=1.0, phase=0.05,
        input_generator=pattern.Gaussian(
            orientation=numbergen.UniformRandom(lbound=-pi,ubound=pi,seed=1)))

    topo.sim["ON_RGC"] =sheet.ActivityCopy(nominal_density=N, precedence=0.7)
    topo.sim["OFF_RGC"]=sheet.ActivityCopy(nominal_density=N, precedence=0.7)

    topo.sim.connect("Photoreceptors","PyNNR",name='.',
        delay=0.05,src_port="Activity",dest_port="Activity")
    topo.sim.connect("PyNNR","ON_RGC",name='...',
        delay=0.05,src_port="ONActivity",dest_port="Activity")
    topo.sim.connect("PyNNR","OFF_RGC",name='...',
        delay=0.05,src_port="OFFActivity",dest_port="Activity")

```

FIGURE 3 | Sample Topographica interface code. This Python code shows a complete, runnable Topographica 0.9.6 simulation interfacing with an external PyNN/PyNEST spiking simulation of ON and OFF retinal ganglion cells. The text in bold starts the PyNN simulation and retrieves the results, and would need to

be changed for interfacing to a new external simulation. The other text sets up an appropriate Topographica simulation framework, and only needs changing to e.g. match the number and type of sheets that you want to expose from the underlying external simulation.

simulating in batch mode with $N=8$ and $\text{simtime}=4$ s takes 16.07 s in Topographica, versus 15.88 s using the native PyNN version (averages of 5 trials; variance negligible). This 0.2-s time difference

consists mainly of libraries that Topographica imports when it starts up, and the ongoing cost is normally negligible for a non-trivial external Python model.

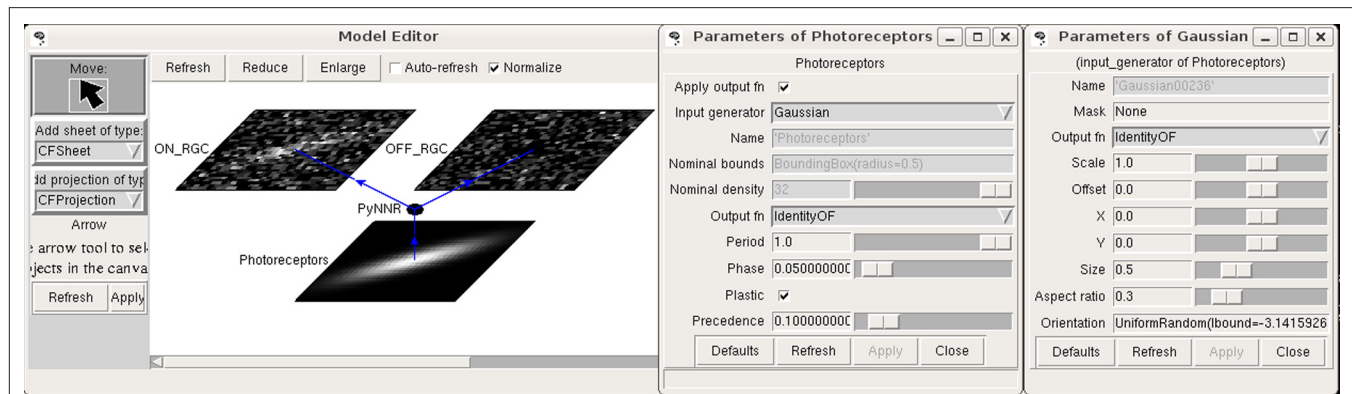


FIGURE 4 | Example architecture. This figure shows the simulation from **Figure 3** running in Topographica. On the input sheet is a 2D Gaussian pattern generated by Topographica and presented to the underlying spiking network,

with the resulting spike count responses shown on the ON and OFF RGC sheets. The type of input pattern and its parameters can be manipulated as shown.

With this interface in place, the external simulation can be used with nearly all of Topographica's features. For instance, **Figure 4** shows one example input pattern and the resulting pattern of ON and OFF RGC activity. For this example, the main benefit to having the Topographica wrapper is to be able to present any of the types of input patterns in Topographica's large library of input patterns, using either the GUI so that the results can be seen interactively, or systematically using Python code. For other simulations, e.g. those including cortical areas such as V1, Topographica can compute tuning curves, receptive fields, many types of preference maps, and other analyses and plots for any of the neurons and Sheets available to Topographica, with no coding required. As long as the computation only requires average firing rates, no special-purpose code or additional interface will be needed beyond what is shown in this example. Thus Topographica can be used to provide a consistent set of analyses and plots for a wide variety of underlying simulations.

INTERFACING TO OTHER PYTHON CODE (E.G., PyNEST, NEURON)

The general approach outlined in the section "Interfacing to Perrinet Retinal Model in PyNN" can be used for any other model running in an external simulator that has a Python interface or is written directly in Python. In each case, a new Topographica EventProcessor class can be created to accept incoming events, process them somehow, and generate appropriate output. For instance, similar steps would have been used if the retina model had been written in PyNEST directly rather than PyNN, or in NEURON's own Python interface. As long as the external simulator can be told to use Topographica's copy of Python, then Topographica can import the required functions, execute them as part of such a class, and thus control its input and output. As a result, the main issues with interfacing to other Python-based simulators are not so much technical as conceptual; these conceptual issues will be reviewed in the "Discussion" section.

INTERFACING TO MATLAB

Topographica can also connect easily to external simulations running in Matlab, using the Python ↔ Matlab interface package `mlabwrap`³ that is supplied with Topographica.

³<http://mlabwrap.sourceforge.net>.

For instance, the following complete, runnable Topographica script defines a Python/numpy array `a` and then calls a Matlab function "nestedsum" on it:

```
from mlabwrap import mlab
import numpy
len=100000
a=numpy.array(range(len))
print mlab.nestedsum(a, len)
```

Here `nestedsum.m` is an arbitrary example of a Matlab function placed somewhere in Matlab's path, containing:

```
function s = nestedsum(a,len)
s=0.0;
for i=1:len
    s=s+sum(a);
end
```

(This code prints `5.0000e+14` when run from Matlab, and `4.99995000e+14` when run from Topographica/Python.) Any built-in or user-supplied Matlab function can be called similarly (including plotting code like `mlab.plot(a)`), with nearly seamless interchange of scalar and array data between the two systems. This capability makes it simple to develop interfaces like that in the section "Interfacing to Perrinet Retinal Model in PyNN", or just to use small bits of Matlab code or visualizations when appropriate.

The `mlabwrap` package performs some data conversion behind the scenes, but the overhead is still usually negligible. The example above run on the same machine as for PyNN takes 12.27 s in Topographica, versus 11.57 s for a pure Matlab version. Again, this 0.7 s difference includes the entire startup time, and increases little with simulation size (e.g. 0.8 s out of 44 for `len=200000`).

The main technical limitation of the `mlabwrap` Matlab interface is that at present it only supports 1D and 2D arrays, because the `mlabwrap` author has not yet added n-dimensional array support. More importantly, interfacing to external Matlab models can be difficult because of the monolithic (as opposed to object-oriented) programming style typically used for Matlab programming. For instance, the Olshausen and Field (1996) model

available from⁴ is a good match to Topographica conceptually, but running it within Topographica in a useful way requires splitting up the Matlab code into three components to handle the input pattern generation, response to the input, and the weights update separately. These functions were originally controlled by a single Matlab script. Thus in practice how difficult it would be to interface to Matlab code depends on the programming style and complexity, with simple functions being simple to access but complicated models potentially requiring prior reorganization on the Matlab side.

INTERFACING TO C/C++

Python offers a wide variety of methods for interfacing to C or C++ code, any of which could be used with Topographica. The specific interface currently used for the performance-critical portions of Topographica is Weave⁵, which allows snippets of C or C++ code to be called easily from within Python code. A sample complete, runnable Topographica/Python script with C code is:

```
import weave, numpy
len=100000
sum=0.0
a=numpy.array(range(len))
code = """
    int i,j;
    for (i=0; i<len; i++)
        for (j=0; j<len; j++)
            sum+=a[i];
    return_val=sum;
"""
print weave.inline(code,["a","len","sum"])
```

Here the C code in the string named `code` is computing the same function as the Matlab code above; it will print 4.99995e+14 when run. The first time it is run the C compiler will be called automatically to compile that code fragment, and then the saved object file will be reused in subsequent calls and on subsequent runs, unless the C code string is changed. This approach makes it simple to include bits of existing C code to optimize specific functions, or to make calls to C libraries.

The C interface adds very little overhead, in part because it uses numpy arrays in place. The example above takes 10.34 s in Topographica, versus 10.07 for a pure C equivalent. This 0.3-s difference is primarily due to the Topographica startup time, because it does not increase with simulation size or length. Also note that the full C version must be recompiled for any change, even trivial ones, while the Topographica/Python version only recompiles when the code string changes (which is typically rare if C is used only for performance-critical sections; recompilation adds about 1 s to the runtime in this example).

Using weave in this way makes it simple to add small bits of C code, but other approaches such as `ctypes` (included in Python 2.5) can be more suitable for interfacing to large external C packages. Again, how difficult the interface will be depends on whether the external code is arranged into entities that can be called directly from Topographica; as discussed below,

reorganizing the code in this way is usually straightforward but can take some effort.

DISCUSSION

As the examples above show, very little coding is required to wrap even complex simulations into the basic Sheet and EventProcessor components used in Topographica. A large class of models across different modelling and analysis levels (e.g., firing-rate, integrate-and-fire, and compartmental neuron models) can fit into this structure, allowing all of them to be analyzed and compared consistently, interconnected where appropriate, and explored visually even if the underlying simulator has no graphical interface (as for NEST). Although the general problem of simulator interoperability is difficult to address, in this specific case it is relatively easy to get practical benefits from combining simulators.

Although the approach outlined above is general purpose, it does require coding a new Topographica component to match each specific model implemented externally. A useful but more complex alternative would be to provide a detailed mapping between object types in an external simulator. For instance, one could provide a Topographica Sheet object that instantiates a corresponding NEST layer object, and similarly for a Topographica Projection object and a NEST connection object. In this way NEST or other simulators could be used to provide specific functionality missing from Topographica, rather than to implement complete models. However, developing such interfaces is much more involved than the simple wrapping described here.

Even though the Topographica Sheet interface is general enough to fit a wide range of current models, there are some models that do not fit within its assumptions. In particular, a Sheet usually needs to have an underlying grid shape to the population of neurons, though individual neurons can be absent or at jittered spatial locations, as long as no more than one neuron is present in any grid cell. (Strictly speaking, it need only be possible to visualize the model in this way; the actual organization is arbitrary.) Also, only Cartesian grids are currently supported, though hexagonal grids could be added in the future. Arbitrary 3D locations will be difficult to support, except by imposing a 3D grid. Note that nonlinear spacings are supported, using arbitrary coordinate mapping between Sheets, e.g. for foveated retinotopic mappings, as long as there is still an underlying grid of neurons.

Apart from operating loosely on a grid, Topographica assumes that models will have regions that are separable from each other, communicating only over well defined channels, and usually incrementally processing some sort of external stimuli that change over time. Although these assumptions are extremely general, and can apply to any physical system, many models do not satisfy them fully. For instance, models that represent inputs not as individual patterns but as correlation functions (e.g. Miller, 1994) are difficult to connect to Topographica, because most of the functionality of Topographica requires testing the response to specific external stimuli (e.g. for measuring maps, tuning curves, and receptive fields). Other types of models that operate in a “batch” mode rather than one pattern at a time (e.g. Olshausen and Field, 1996) can usually be adapted to work in incremental mode as required by Topographica, but they may then run much more slowly.

⁴<https://redwood.berkeley.edu/bruno/sparsenet/>

⁵www.scipy.org/Weave.

Given the ease with which many models can be wrapped, an intermediate-term goal will be to provide example code for wrapping as many current V1 models as possible into Topographica, to establish for the first time a platform for evaluating their behavior and functionality consistently. At present, each model is implemented independently, with different analysis routines and types of visualization, and thus it is extremely difficult to determine if apparent differences in behavior are significant. As long as runnable code is available for each model, wrapping it into Topographica should be straightforward and should provide immediate benefits.

In addition to interfacing with external model components, any of the mechanisms outlined above can be used to call externally defined general-purpose analysis or visualization functions. For instance, the NeuroTools package⁶ defines an object-based Python representation of spike trains, such as those used in the spiking retina model above. A native spiking Topographica model can then use these functions rather than reimplementing them within Topographica.

This paper focuses on making external simulations available within Topographica, to allow simulations at the topographic map level or at lower levels to be brought into a common analysis and testing framework. It is also straightforward to interface in the opposite direction, running a Topographica simulation from within an external system or simulators. The Topographica User Guide⁷ provides detailed examples of running models from the Python command line or Python scripts, and the same interface can be used from within any simulator that has Python bindings. Moreover, Topographica has a highly modular design with few dependencies between components, and there are many Topographica objects that are useful on their own and can be

used just as any other Python object from within an external program.

At present, Topographica is primarily useful for doing analyses based on firing rates, because of its extensive firing-rate based libraries. Spiking simulations are also possible in Topographica, but they are currently quite limited, and will require additional work to establish general-purpose abstractions that can be used to integrate data across models and simulators. In the long run, we intend Topographica to be useful as a high-level platform for analyzing spiking output as well as firing-rate output, and would welcome collaborations with people interested in that topic or in other aspects of Topographica or interoperability development.

In summary, working at the topographic map level makes it practical to provide interconnections between models and simulators working at the same or different levels of detail. As long as the neurons are grouped into two-dimensional sheets of related units, they will be able to interface easily with Topographica's tools and components. The result provides a shared platform for evaluating models from different sources, allowing consistent analysis and testing even for very different implementations. We believe this shared, extensible tool will be highly useful for the community of researchers working to understand the large-scale structure and function of the nervous system.

ACKNOWLEDGMENTS

Supported in part by the National Institutes of Mental Health under Human Brain Project grant 1R01-MH66991, by the National Science Foundation under grant IIS-9811478, and by the EPSRC/MRC Doctoral Training Centre in Neuroinformatics at the University of Edinburgh. Thanks to Laurent Perrinet (INCM/CNRS) for contributing his retina model as an example, for assisting with the process of interfacing it to Topographica, and for making comments on an earlier draft of this manuscript. Thanks also to all of the Topographica developers, particularly Christopher Ball, without whom this work would not have been possible.

REFERENCES

- Amari, S. (1980). Topographic organization of nerve fields. *Bull. Math. Biol.* 42, 339–364.
- Bednar, J. A. (2008). Understanding neural maps with topographica. In *Interactive Educational Media for the Neural and Cognitive Sciences*. Brains, Minds and Media, Vol. 3, bmm # 1402, S. Lorenz and M. Egelhaaf (eds). <http://www.brains-minds-media.org/archive/1402>.
- Blasdel, G. G. (1992). Orientation selectivity, preference, and continuity in monkey striate cortex. *J. Neurosci.* 12, 3139–3161.
- Bosking, W. H., Crowley, J. C., and Fitzpatrick, D. (2002). Spatial coding of position and orientation in primary visual cortex. *Nat. Neurosci.* 5, 874–882.
- Bower, J. M., and Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the GENeral Neural Simulation System*, 2nd Edn. Santa Clara, Telos.
- Cannon, R. C., Gewaltig, M.-O., Gleason, 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.
- Davison, A., Yger, P., Kremkow, J., Perrinet, L., and Muller, E. (2007). PyNN: towards a universal neural simulator API in Python. *BMC Neurosci.* 8(Suppl. 2), P2 (Toronto, Proceedings of the Sixteenth Annual Computational Neuroscience Meeting (CNS*2007)).
- Diesmann, M., and Gewaltig, M. (2002). NEST: an environment for neural systems simulations. In *Forschung und wissenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, Vol. 58, T. Plesser and V. Macho, eds (Göttingen, Ges. für Wiss. Datenverarbeitung), pp. 43–70.
- Djurfeldt, M., and Lansner, A. (2007). Workshop report: 1st INCf workshop on Large-scale Modeling of the nervous system. Available from Nature Precedings. doi:10.1038/npre.2007.262.1.
- Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209.
- Kaas, J. H. (1997). Theories of visual cortex organization in primates. *Cereb. Cortex* 12, 91–125.
- Kremkow, J., Perrinet, L., Kumar, A., Aertsen, A., and Masson, G. (2007). Synchrony in thalamic inputs enhances propagation of activity through cortical layers. *BMC Neurosci.* 8(Suppl. 2), P206. (Toronto, Proceedings of the Sixteenth Annual Computational Neuroscience Meeting (CNS*2007)).
- Merzenich, M. M., Knight, P. L., and Roth, G. L. (1975). Representation of cochlea within primary auditory cortex in the cat. *J. Neurophysiol.* 38, 231–249.
- Miller, K. D. (1994). A model for the development of simple cell receptive fields and the ordered arrangement of orientation columns through activity-dependent competition between ON- and OFF-center inputs. *J. Neurosci.* 14, 409–441.
- Ohki, K., Chung, S., Ch'ng, Y. H., Kara, P., and Reid, R. C. (2005). Functional imaging with cellular resolution reveals precise micro-architecture in visual cortex. *Nature* 433, 597–603.
- Olshausen, B. A., and Field, D. J. (1996). Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* 381, 607–609.
- Ousterhout, J. K. (1998). Scripting: higher level programming for the 21st century. *Computer* 31, 23–30.

- Roque Da Silva Filho, A. C. (1992). Investigation of a generalized version of Amari's continuous model for neural networks. PhD Thesis. Brighton, School of Cognitive and Computing Sciences, University of Sussex.
- Van Essen, D. C., Lewis, J. W., Drury, H. A., Hadjikhani, N., Tootell, R. B. H., Bakircioglu, M., and Miller, M. I. (2001). Mapping visual cortex in monkeys and humans using surface-based atlases. *Vision Res.* 41, 1359–1378.
- Weliky, M., Bosking, W. H., and Fitzpatrick, D. (1996). A systematic map of direction preference in primary visual cortex. *Nature* 379, 725–728.
- Xu, X., Anderson, T. J., and Casagrande, V. A. (2007). How do functional maps in primary visual cortex vary with eccentricity? *J. Comp. Neurol.* 501, 741–755.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 15 September 2008; paper pending published: 20 November 2008; accepted: 26 February 2009; published online: 24 March 2009.*
- Citation: Bednar JA (2009) Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. Front. Neuroinform. (2009) 3:8. doi: 10.3389/neuro.11.008.2009*
- Copyright © 2009 Bednar. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Python scripting in the Nengo simulator

Terrence C. Stewart*, Bryan Tripp and Chris Eliasmith

Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, ON, Canada

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Andrew P. Davison, CNRS, France
Jochen M. Eppler, Honda Research
Institute Europe GmbH, Germany;
Albert Ludwigs University, Germany

*Correspondence:

Terrence C. Stewart, Centre for
Theoretical Neuroscience, University of
Waterloo, 200 University Avenue West,
Waterloo, ON, Canada N2L 3G1.
e-mail: tcstewart@uwaterloo.ca

Nengo (<http://nengo.ca>) is an open-source neural simulator that has been greatly enhanced by the recent addition of a Python script interface. Nengo provides a wide range of features that are useful for physiological simulations, including unique features that facilitate development of population-coding models using the neural engineering framework (NEF). This framework uses information theory, signal processing, and control theory to formalize the development of large-scale neural circuit models. Notably, it can also be used to determine the synaptic weights that underlie observed network dynamics and transformations of represented variables. Nengo provides rich NEF support, and includes customizable models of spike generation, muscle dynamics, synaptic plasticity, and synaptic integration, as well as an intuitive graphical user interface. All aspects of Nengo models are accessible via the Python interface, allowing for programmatic creation of models, inspection and modification of neural parameters, and automation of model evaluation. Since Nengo combines Python and Java, it can also be integrated with any existing Java or 100% Python code libraries. Current work includes connecting neural models in Nengo with existing symbolic cognitive models, creating hybrid systems that combine detailed neural models of specific brain regions with higher-level models of remaining brain areas. Such hybrid models can provide (1) more realistic boundary conditions for the neural components, and (2) more realistic sub-components for the larger cognitive models.

Keywords: Python, neural models, neural engineering framework, theoretical neuroscience, neural dynamics, control theory, representation, hybrid models

INTRODUCTION

Large-scale neural modeling requires software tools that not only support efficient simulation of hundreds of thousands of neurons, but also provide researchers with high-level organizational tools. Such neural models involve heterogeneous components with complex interconnections that may be either speculative in nature or constrained by existing neurobiological evidence. To effectively construct, modify, and investigate the behaviour of these models, researchers need to be able to specify the collective behavior of large groups of neurons as well as the low-level physiological details.

In order to support this style of research, we have developed a neural simulator package called Nengo. For high-level organization, Nengo makes use of the neural engineering framework (NEF; Eliasmith and Anderson, 2003), which provides methods for abstractly describing the representations and transformations involved in a neural model and how they relate to spiking behavior. To provide access to the broad range of functionality we require (from neural groups to individual synapses), we integrated a Python language scripting system into the simulator. This enables a variety of novel features, including the inspection and modification of running models, the ability to script common experimental tasks, and the integration of non-neural cognitive models. In this paper, we describe this system (see Introduction), discuss the features related to its use of Python (see Python and Nengo), and provide an extended example of ongoing research that has directly benefited from these abilities (see Integration with Other Libraries).

NENGO

Nengo is an open-source cross-platform software package for modeling neuronal circuits¹, and tested on Macintosh OS X, Linux, and Microsoft Windows. It is implemented in Java, and provides both a detailed Application Programming Interface and a Graphical User Interface (**Figure 1**), so that it is suitable for both novice and expert modelers. As will be discussed, the Python scripting system forms a bridge between the easy-to-use graphical environment and the full power of the underlying programmatic interface. This ensures a smooth transition from novice to expert, as all aspects of the simulation are accessible at all times.

A variety of spiking point-neuron models are provided with Nengo. This includes the standard LIF neuron and the Hodgkin-Huxley model, as well as an adapting LIF (La Camera et al., 2004) and the Izhikevich model (Izhikevich, 2003). Integration is performed with a variable-timestep integrator, using the Dormand-Prince 4th and 5th order Runge-Kutta formulae (Dormand and Prince, 1980). At the network level, interaction between neurons treats spikes as discrete events; Nengo is not meant for neural models where the detailed voltage profile of a specific spike affects the post-synaptic neurons.

These neuron models can be connected directly to form simple networks, and input can consist of current injection or voltage clamp. Spike times, membrane voltages, and current can be recorded from the neurons. This approach is suitable for situations where connectivity information is known, or where the dynamics of a

¹<http://nengo.ca>

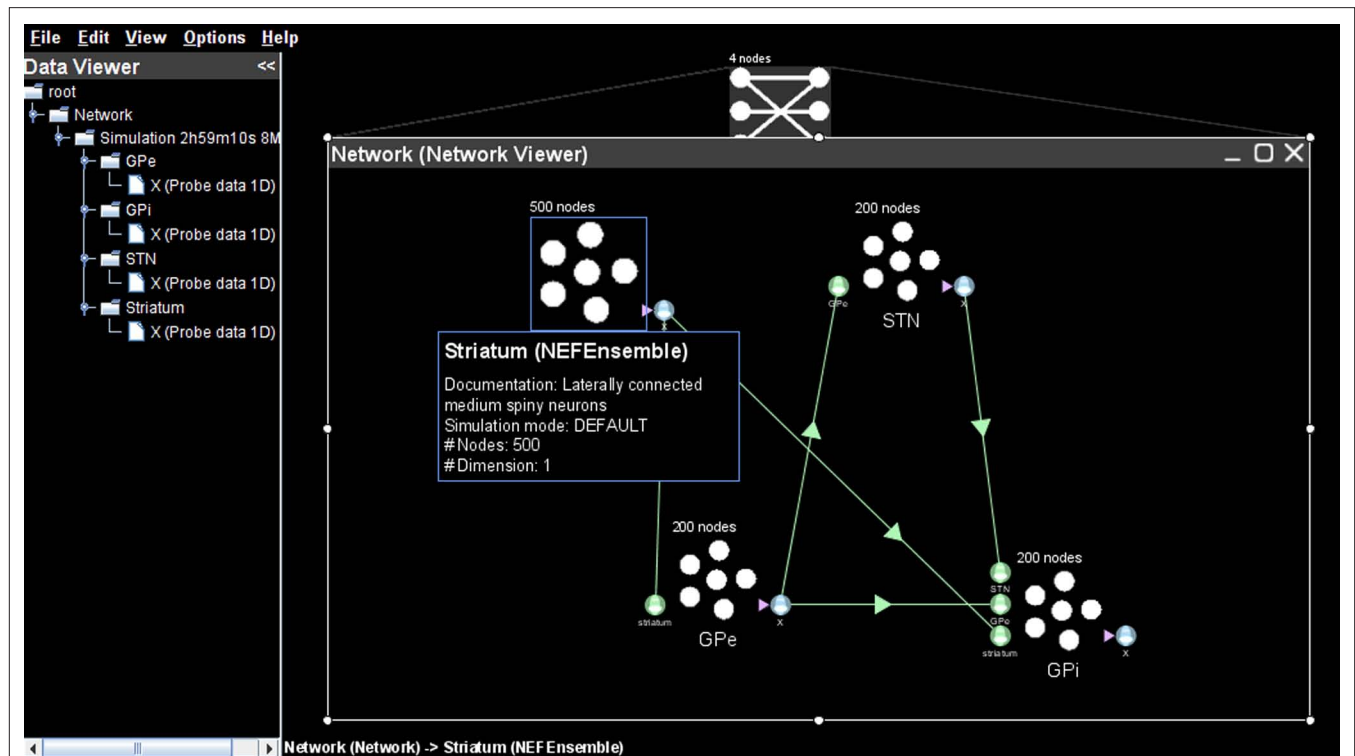


FIGURE 1 | A neural model of the basal ganglia developed in Nengo.

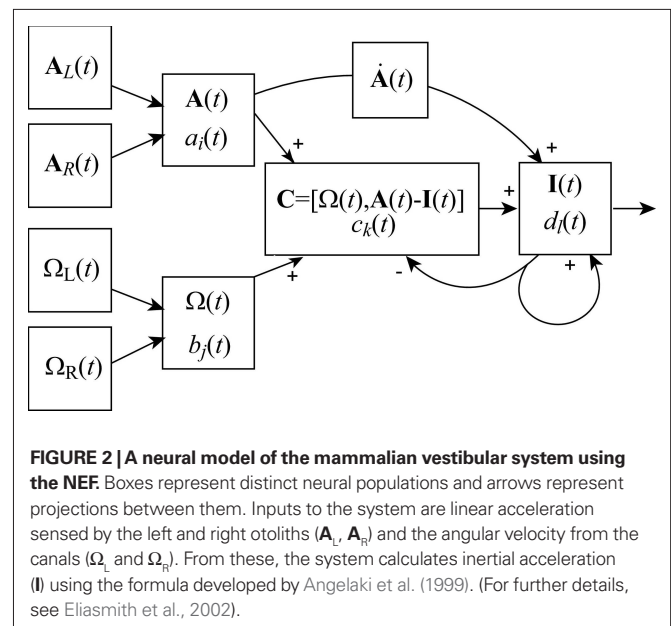
particular configuration are being investigated. However, modeling of more sophisticated population-coding networks is greatly facilitated by using the NEF-related features of the simulator.

NEURAL ENGINEERING FRAMEWORK

For complex neural models, it is often useful to describe the system of interest at a higher level of abstraction, such as that shown in Figure 2. For this reason, we define heterogeneous groups of neurons (where individual neurons vary in terms of their neural properties such as bias current and gain) and projections between these groups. We can then use the NEF (Eliasmith and Anderson, 2003) as a method for realizing this high-level description using neural models with adjustable degrees of accuracy. The NEF provides not only a method for encoding and decoding time-varying representations using spike trains, but also a method for deriving linearly optimal synaptic connection weights to transform and combine these representations. This approach combines work from a variety of researchers, most notably Georgopoulos et al. (1986), Rieke et al. (1999), Salinas and Abbott (1994), and Seung (1996).

The NEF has been used to model the barn owl auditory system (Fischer, 2005), rodent navigation (Conklin and Eliasmith, 2005), escape and swimming control in zebrafish (Kuo and Eliasmith, 2005), working memory systems (Singh and Eliasmith, 2006), the translational vestibular ocular reflex in monkeys (Eliasmith et al., 2002), and the manipulation of symbolic representations to support high-level cognitive systems (Stewart and Eliasmith, 2009).

Within the NEF, a neural group forms a distributed representation of a time-varying vector $\mathbf{x}(t)$ of arbitrary length. Each neuron



i has an associated preferred direction vector $\tilde{\Phi}$ (the stimulus for which it most strongly fires), bias current J^{bias} , and scaling factor α . For a given neuron, α and J^{bias} can be experimentally determined from its maximum firing rate and the minimum value of \mathbf{x} for which it responds. If the nonlinearities of any given neural model (LIF, ALIF, etc.) are written as $G[\cdot]$ and the neural noise of variance

$\sigma^2 = \eta(\sigma)$, then the encoding of any given $\mathbf{x}(t)$ as the temporal spike pattern across the neural group is given as Eq. 1.

$$\sum_n \delta(t - t_{in}) = G_i [\alpha_i \tilde{\Phi} \cdot \mathbf{x}(t) + J_i^{bias} + \eta_i(\sigma)] \quad (1)$$

Given this spiking pattern, we can in turn estimate the original vector as $\hat{\mathbf{x}}(t)$. In some approaches (e.g. Georgopoulos et al., 1986), this is done by weighting each encoding vector $\tilde{\Phi}$ by the average firing rate of the corresponding neuron. In the NEF, however, we derive the linearly optimal decoding vectors Φ for each neuron (see Eliasmith and Anderson, 2003 for details). This method has been shown to uniquely combine accuracy and neurobiological plausibility (e.g. Salinas and Abbott, 1994).

$$\begin{aligned} \Phi &= \Gamma^{-1} \Upsilon \\ \Gamma_{ij} &= \int a_i a_j dx \\ \Upsilon_j &= \int a_j x dx \end{aligned} \quad (2)$$

Since $\mathbf{x}(t)$ varies over time, we do not weight these decoding vectors by the average firing rate. Instead, we weight them with the post-synaptic current $h(t)$ induced by each spike. The shape and time-constant of this current are determined from the physiological properties of the neural group:

$$\hat{\mathbf{x}}(t) = \sum_{in} \delta(t - t_{in}) * h_i(t) \Phi_i = \sum_{in} h(t - t_{in}) \Phi_i \quad (3)$$

The representational error between $\mathbf{x}(t)$ and $\hat{\mathbf{x}}(t)$ is dependent on the particular neural parameters and encoding vectors, but in general is inversely proportional to the number of neurons in the group. Given a sufficient number of neurons, an arbitrary level of accuracy can be reached. For a known number of neurons with known physiological properties, we can determine how well the values can be represented.

The derivation of the optimal decoding vector also allows us to determine the optimal connection weights to perform arbitrary transformations of these representations. For linear functions, consider two neural populations, X representing $\mathbf{x}(t)$ and Y representing $\mathbf{y}(t)$. If we want $\mathbf{y}(t) = \mathbf{M} \mathbf{x}(t)$, we can derive the following for the neurons in population Y:

$$\begin{aligned} \sum_m \delta(t - t_{jm}) &= G_j [\alpha_j \tilde{\Phi} \mathbf{y}(t) + J_j^{bias}] \\ \text{substitute: } \mathbf{y}(t) &= \mathbf{M} \mathbf{x}(t) \quad \mathbf{x}(t) \approx \hat{\mathbf{x}}(t) = \sum_{in} h(t - t_{in}) \Phi_i \\ &= G_j [\alpha_j \tilde{\Phi} \mathbf{M} h(t - t_{in}) \Phi_i + J_j^{bias}] \\ &= G_j [(\alpha_j \tilde{\Phi} \mathbf{M} \Phi_i) h(t - t_{in}) + J_j^{bias}] \end{aligned} \quad (4)$$

This manipulation converts weighted post-synaptic currents caused by the spikes in neural group X into a spiking pattern for group Y that would cause Y to represent the value in X transformed by the linear operation M. Crucially, if we set the synaptic connection weights between the i th neuron in X and the j th neuron in Y to be $\omega_{ij} = \alpha_j \tilde{\Phi} \mathbf{M} \Phi_i$, then the post-synaptic neurons will encode $\mathbf{M} \mathbf{x}(t)$. This allows us to develop a model by defining the hypothesized computations and directly solving for the corresponding

connection weights, rather than relying on a learning rule or manually setting the weights.

For nonlinear transformations, we can generalize the derivation of the decoding vector to estimate the desired function $f(\mathbf{x})$. This provides a new set of decoding vectors $\Phi^{f(\mathbf{x})}$ which can be used in place of the previous Φ to provide an optimal linear estimate of this function. This allows arbitrary nonlinear functions to be computed, although more complex nonlinearities across multiple dimensions of \mathbf{x} will require more neurons with $\tilde{\Phi}$ values that lie in those dimensions.

$$\begin{aligned} \Phi^{f(\mathbf{x})} &= \Gamma^{-1} \Upsilon^{f(\mathbf{x})} \\ \Gamma_{ij} &= \int a_i a_j dx \\ \Upsilon_j^{f(\mathbf{x})} &= \int a_j f(\mathbf{x}) dx \end{aligned} \quad (5)$$

Treating neural groups as representing time-varying vectors and synaptic connections as performing arbitrary transformations allows us to organize a neural system using the powerful framework of control theory. Eliasmith and Anderson (2003) have shown how to translate any state-space model from modern control theory into an equivalent neural circuit. For example, an ideal integrator is shown in **Figure 3A**, and its NEF counterpart, a neural integrator implemented with 300 LIF neurons, is shown in **Figure 3B**. Importantly, the idealized version can be seen as an approximation of the actual neural behaviour. As is discussed in the next section, this feature can be used to create large-scale models where every component can potentially be simulated at the level of neurons, even though it may be too computationally expensive to do so for the whole system.

The NEF provides a generic method for modeling any neural system where groups of neurons are taken to represent scalars, vectors, and functions, and where synaptic connections implement transformations on these representations. The system generalizes to higher dimensional vectors and has also been used as the basis of models of path integration (Conklin and Eliasmith, 2005) and working memory (Singh and Eliasmith, 2006). Arbitrary nonlinear encodings are supported by adjusting G to be the output of any neural model. While the above derivation assumes linear dendrites, the approach generalizes to nonlinear dendritic behavior as well (see Eliasmith and Anderson, 2003).

PROGRAMMING INTERFACE

Nengo is a highly modular object-oriented Java program, making the underlying simulation system extensible and adaptable to novel modeling situations. The following features are directly exposed to the developer by the architecture:

Neuron models

Specialized neuron models can be written in Python or Java. These can extend existing models and/or use generic components, such as the built-in dynamical system solver. For example, a Nengo implementation of a dopamine-sensitive bistable striatal neuron (Gruber et al., 2003) was recently developed. The core of its implementation is shown later in this paper.

Neural plasticity

Arbitrary functions can be added for adjusting synaptic weights based on spike timing and modulatory signals.

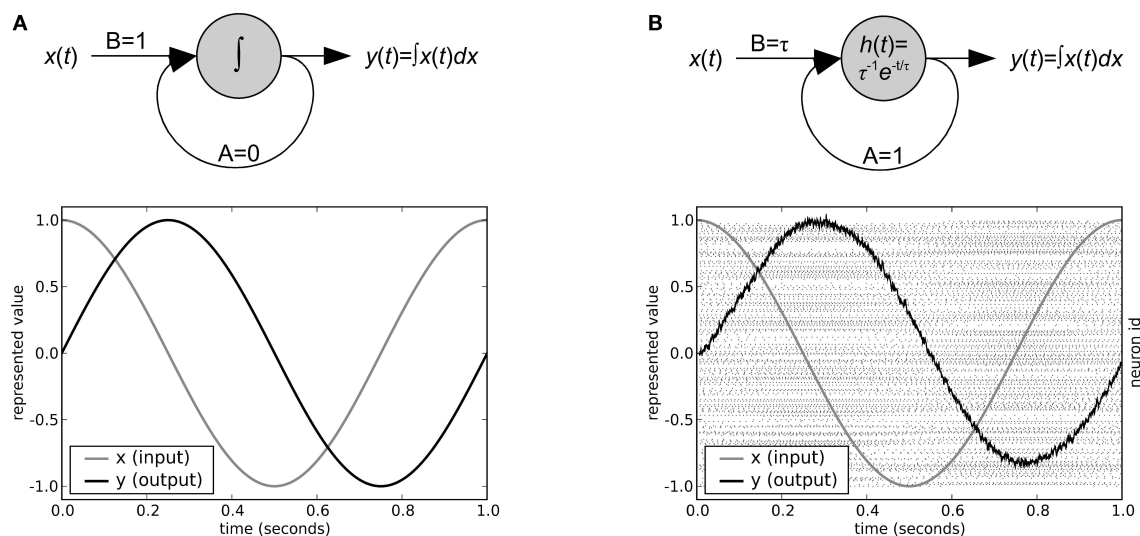


FIGURE 3 | A classic control-theory integrator (A) and an Nengo integrator (B). Both integrators are provided with the same sine wave input $x(t)$. The Nengo integrator uses 300 LIF neurons with maximum firing rates distributed uniformly between 100 and 200 Hz, post-synaptic current time constants of

20 ms, and refractory periods of 2 ms. The output value for the Nengo integrator is determined from the individual spike times of each neuron using Eq. 3. Neuron spikes are shown as dots in panel (B), with neurons arranged along the y-axis.

Muscle models

For any neural models involving motor neurons, the dynamic behavior of the muscles form an important part of the model as well. Nengo supports multiple approaches to muscle modeling (e.g. Keener and Sneyd, 1998; Winter, 1990).

All of these components, along with other useful tools for modeling such as external inputs and probability distribution functions for various neural properties, can be implemented in either Java or Python. As discussed in further detail below, all of the features of Nengo are exposed in both languages, allowing developers the flexibility to choose the approach which is most suitable to them.

Since the software was developed for large-scale modeling, each component within a Nengo model has an adjustable simulation mode. For neural groups defined using the Nengo approach, three modes are provided: spiking neurons, rate neurons, and a direct high-level abstraction of the overall neural behavior. This direct mode allows for fast approximate simulations where the individual neurons within the group are not simulated; instead the behavior is approximated in terms of the underlying represented values $x(t)$. When neural groups simulated at a low level connect with groups simulated at a high level, Eqs 1 and 3 (above) are used to determine the corresponding spike trains and $\hat{x}(t)$ values. If sufficient time and computational resources are available, all parts of the model can be simulated in terms of spiking neurons. However, this capability of mixing levels of simulation means that a detailed neural model involving tens of thousands of neurons can be embedded within a high-level approximation of the millions of other neurons with which this system must interact. By switching modes of particular neural groups, the effect of different degrees of accuracy can be easily determined. Changing simulation models is also a useful exploratory tool, since approximate behavior can be determined quickly.

USER INTERFACE

Nengo also provides a graphical user interface for constructing and simulating models. Neural groups can be created and configured, projections and synaptic connection weights can be defined, and simulations can be run and analyzed, all through a point-and-click interface. This provides a direct method for visualizing the overall organization of a complex neural circuit at multiple levels of abstraction.

This interface is intended to be equally suitable for novice and expert users. In particular, we wanted to ensure that while common tasks are made easier by the interface, more experienced users have simultaneous access to the full capabilities of the programmatic interface. To achieve this, a Python scripting interface is embedded in the graphical user interface, complete with a full history and object-inspection based code completion tools. Usage examples of this combined graphical and scripting system are given in the next section.

Python AND NENGO

To blend the graphical interface with the full power of the underlying programmatic interface, we embedded a Python scripting engine. This allows Python code and scripts to run in concert with the user interface. In this way, users can follow a graphical point-and-click approach for common modeling tasks, and turn to Python scripting for more complex or specialized tasks.

Since Nengo is implemented in Java, the scripting interface was implemented with Jython². This is a Java implementation of Python, which allows Python code to be compiled to the Java Virtual Machine, and provides seamless interaction between languages, including inheritance between languages and full access to the Java API using Python syntax. Importantly, no extra development effort (beyond embedding Jython within the Nengo graphical user

²<http://www.jython.org>

interface) was required to allow Python access to the Nengo code; Jython automatically provides the Python syntax and interactive capabilities described here.

As an example, **Figure 4** shows the Python scripting interface being used to duplicate an existing group of neurons (groupA, created using the point-and-click interface). This duplication is performed using the standard Java `clone()` method. The name of this new neural group is then changed to groupB and it is added to the existing network. These tasks can also be performed via the graphical interface; this example is meant to show the direct relationship between the underlying Java entities, the graphically displayed objects, and the Python scripting.

RUN-TIME INSPECTION AND MODIFICATION

The simplest use of the scripting system is to display and edit the values of variables within the simulation. The most recent object selected in the graphical display is always bound to the variable `that` in the scripting system. This allows us to quickly inspect and change objects. For example, to display the bias current (I^{bias} in Eq. 1) of a given neuron, we can click on it in the interface and type the following, with the output from Nengo shown in bold:

```
print that.bias
1.9371659755706787
```

The command `that.bias` is automatically converted by Jython into the Java method invocation `getBias()` on the currently

selected object, and the result is printed to the screen. This convenience functionality is built in to Jython and works with any Java code that conforms to the JavaBean properties standards.

For more complex situations, we use Python to extract relevant information and analyze and record it in the desired manner. For example, we can display all of the I^{bias} values across a group of neurons, find their average, and save the values in a comma-separated values (CSV) file.

```
bias=[n.bias for n in groupA.nodes]
print bias
[1.9371659755706787, 0.5016773343086243,
0.40018099546432495, 2.8485255241394043,...
print sum(bias)/len(bias)
-17.20441970984141
import csv
csv.writer(file('output.csv','w')).writerow(bias)
```

This approach can also be used to set values within the simulation; the command that `.bias = 0.3` is converted into the Java method `setBias(0.3)` by Jython. This allows model parameters to be set in a flexible manner. For example, to cause the RC time constant for a group of neurons that use an LIF spike generator to be uniformly distributed between 200 and 300 ms, we can do the following:

```
for n in groupA.nodes: n.generator.tauRC=random.
uniform(0.2,0.3)
```

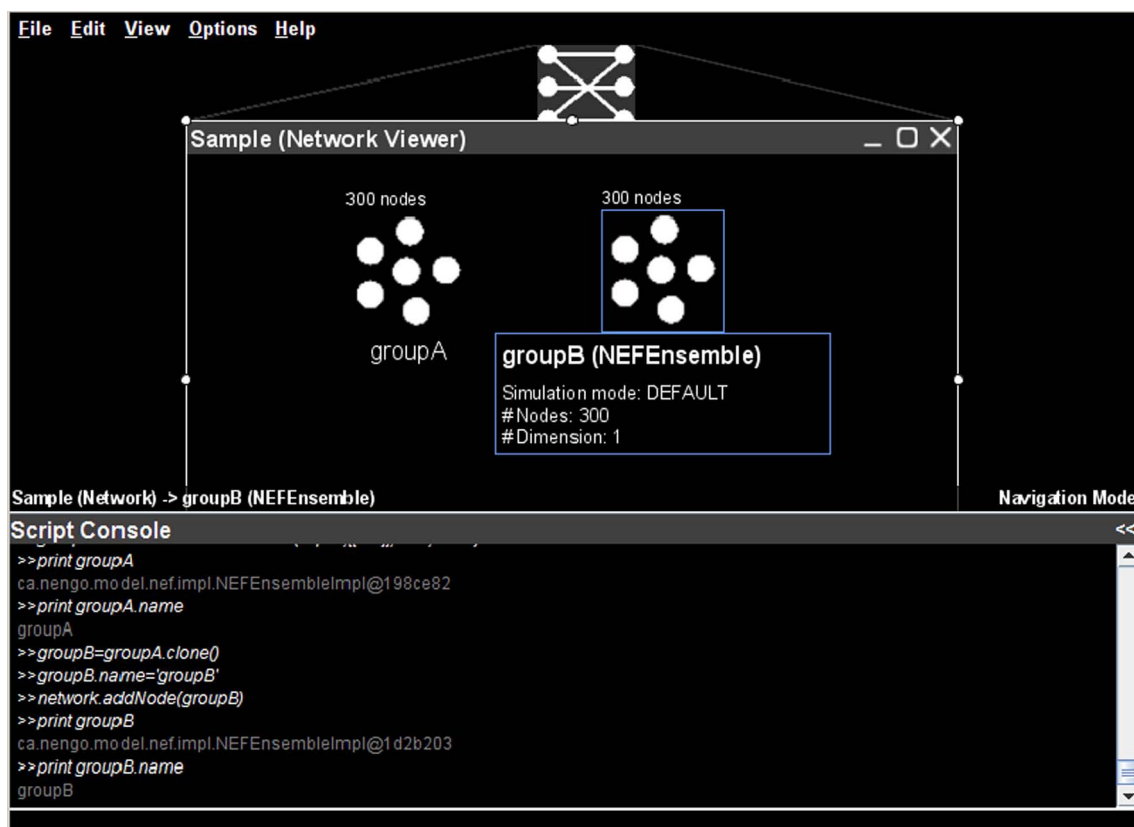


FIGURE 4 | Basic usage of the Python scripting interface to interact programmatically with a neural model.

PROGRAMMATIC MODEL CREATION

Python can also be used to directly create models. This involves defining the various neural groups and specifying the projections between them. As this is done, Nengo automatically solves for the required synaptic weight matrices, based on the neural properties, preferred direction vectors, and the desired transformation.

To configure a NEF neural group, we define the various parameters based on the neurobiological properties of the particular types of neurons being modeled. This can include specifying probability distributions for those aspects that are heterogeneous across the group.

```
ef=ca.nengo.model.nef.impl.NEFEnsembleFactoryImpl()
ef.nodeFactory.tauRC=0.02
ef.nodeFactory.tauRef=0.002
ef.nodeFactory.maxRate=GaussianPDF(200,50)
ef.nodeFactory.intercept=IndicatorPDF(-1,1)
```

Given this definition, we can now create neural groups of the desired size, encoding vectors of a given length. Terminations are defined by providing the linear transformation matrix (M in Eq. 4) and the post-synaptic time constant. Nonlinear functions are computed by creating a separate origin and providing the desired function. This separate origin does not imply a separate source of action potentials; it is implemented internally using the same spike timing as the standard projection origin (i.e. the neural group's axons), but with a different set of decoding vectors, as per Eq. 5. For example, the following script will create a neural group which accepts five inputs and outputs the maximum value encoded by those five inputs, using the neural properties defined above.

```
group=ef.make('group',neurons=1000,dimensions=5)
for i in range(5):
    M=[0,0,0,0,0]
    M[i]=1
    group.addDecodedTermination('in'+i,[M],tauPSC=0.007
                                .modulatory=False)
group.addDecodedOrigin('max',[PostfixFunction
                              ('max(x)',5)], 'AXON')
```

We have found this approach to be flexible and highly useful for our ongoing research. In particular, this has allowed us to quickly explore the behaviors of complex cognitive models, including our ongoing work on neural implementation of Kalman filters for sensorimotor integration, language based reasoning, the role of basal ganglia in motor control, and other projects. While much of Nengo is devoted to supporting NEF-style models, similar commands are used for models that directly specify neural connections and plasticity, or that merge the two approaches.

SCRIPTING OF COMMON TASKS

Besides directly creating or modifying models, Python is also useful for defining stimuli, controlling simulations, and analyzing or recording results. Inputs to neural groups can be defined using arbitrary Python code, allowing for anything from simply adding white noise to a baseline input value to providing dynamic inputs based on the current motor outputs of the model.

More generally, we can use the scripting system to evaluate neural models. That is, we can easily run multiple simulations, adjusting parameters, and recording the data. For example, the following code

runs an existing simulation 10 times, adjusts the refractory period each time, and records the model output to a MATLAB® file. This allows us to quickly explore the behavioral effects of physiological parameters.

```
result=ca.nengo.io.MatlabExporter()
for i in range(10):
    for n in groupA.nodes: n.generator.tauRef=0.001*i
    simulator.run(start=0,end=1)
    result.add('data'+i,probe.data)
result.write(file('result.m','w'))
```

DEFINING NEURON TYPES

Given the wide range of existing neuron models, and the continual development of new ones, Nengo needs to allow the user to easily define and use new neuron models throughout the system. This is facilitated by a general-purpose dynamical system solver which creates spiking neuron models based on their dynamical description. Given the simplicity of the Python syntax, existing published neural models can be easily translated from their mathematical description into code.

For example, the following Python code defines the membrane dynamics for a dopamine-sensitive bistable striatal neuron developed by Gruber et al. (2003). This model's behaviour is affected by levels of dopamine, which are set using a separate modulatory input within Nengo, allowing it to be controlled by other neural groups.

```
Cm=1; E_K=-90; g_L=.008; VKir2_h=-111; VKir2_c=-11;
gbar_Kir2=1.2
VKsi_h=-13.5; VKsi_c=11.8; gbar_Ksi=.45; R=8.315;
F=96480; T=293
VLCa_h=-35; VLCa_c=6.1; Pbar_LCa=4.2; Ca_o=.002;
Ca_i=0.0000001

class GruberDynamics(ca.nengo.dynamics.
AbstractDynamicalSystem):
    def f(self,time,input):
        I_s,mu=input
        Vm=self.state[0]

        L_Kir2=1.0/(1+exp(-(Vm-VKir2_h)/VKir2_c))
        L_Ksi=1.0/(1+exp(-(Vm-VKsi_h)/VKsi_c))
        L_LCa=1.0/(1+exp(-(Vm-VLCa_h)/VLCa_c))
        P_LCa=Pbar_LCa*L_LCa

        x=exp(-2*Vm/1000*F/(R*T))
        I_Kir2=gbar_Kir2*L_Kir2*(Vm-E_K)
        I_Ksi=gbar_Ksi*L_Ksi*(Vm-E_K)
        I_LCa=P_LCa*(4*Vm/1000*F*F/(R*T))*
            ((Ca_i-Ca_o*x)/(1-x))
        I_L = g_L*(Vm-E_K)

        return [-1000/Cm*(mu*(I_Kir2+I_LCa)+I_Ksi+I_L-I_s)]
```

Using this approach, any component of a neural system expressed in terms of its internal dynamics can be integrated into a Nengo model.

INTEGRATION WITH OTHER LIBRARIES

Since Nengo integrates a Python scripting system via Jython, Nengo models can also make use of other code libraries. This not only includes the standard built-in Python libraries for string processing,

random number generation, asynchronous communication, and other common tasks, but also any other library written in Java or 100% Python. Unfortunately, Jython currently does not support direct integration with Python extension modules, such as NumPy or SciPy. To make use of such tools for data analysis, the output from Nengo can be exported to a file. However, for modules which can be directly integrated, Nengo allows for seamless communication between systems from within the graphical user interface.

ACT-R

As an example of this model integration, we have combined Nengo with a Python implementation of ACT-R, a high-level model of human cognition (Anderson and Lebiere, 1998). ACT-R divides human cognitive function into a variety of separate modules, which map on to particular brain areas (Anderson et al., 2008). Although no neural implementation of these modules exists as of yet, the underlying theory provides millisecond-level timing information for the behaviour of these modules which accords well with timing of overt behavior and of fMRI BOLD responses. ACT-R distills decades of cognitive science research into a form that provides a high-level model of many brain regions that can, in theory, interact with a lower-level neural model. In order to bring about this possibility, we connected the Python implementation of ACT-R (Stewart and West, 2007) to Nengo. This is freely available as part of CCMSuite³.

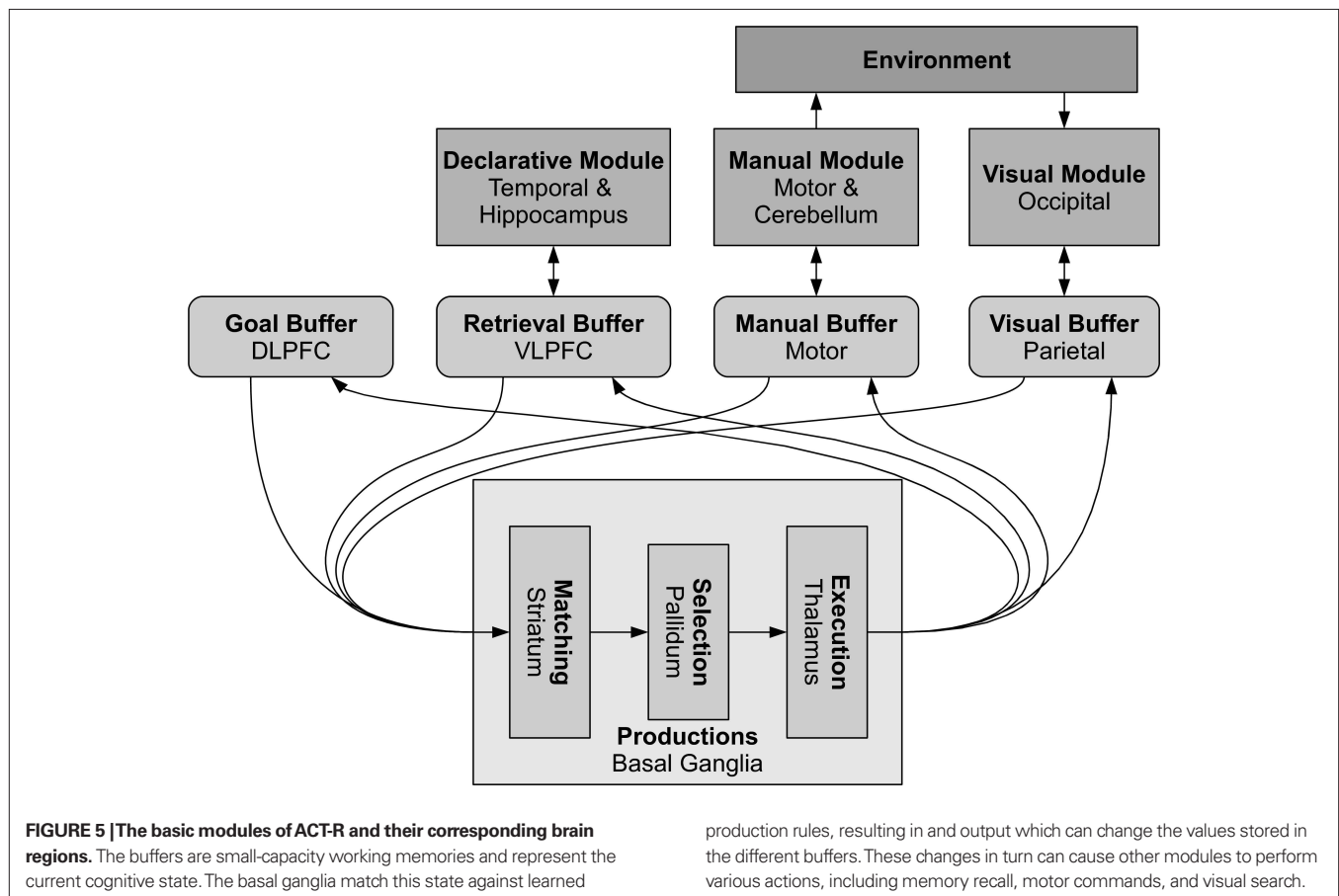
³<http://ccmlab.ca/ccmsuite.html>

The modules in ACT-R (see **Figure 5**) were developed to explain human cognitive performance across a wide variety of tasks, including serial recall, visual search, mental arithmetic, task switching, and the use of graphical interfaces. Each cortical module maintains a *buffer* which contains one *chunk* of information. This chunk is a symbolic representation of the current working memory associated with that module. For example, the declarative memory module may retrieve the fact that *two plus two is four*, storing that in its buffer as the chunk 'value1:two value2:two operation:plus result:four'. The symbolic values within a chunk are organized into *slots*, and a chunk of a given type always has the same set of slots.

Communication between modules is controlled by a generalized action selection system associated with the basal ganglia. This contains a set of production rules: IF-THEN statements which identify which values should be placed in which buffers based on the current values in other buffers. To fit a wide range of behavioral data, a cycle of determining which productions match the current situation, selecting one of them, and sending its associated values is assumed to take the brain approximately 50 ms.

REPRESENTATION MAPPING

To integrate ACT-R and Nengo, we need to define a system of communication between them. That is, if we construct a neural model of a given brain region, we need to remove the corresponding component from the ACT-R model and connect the Nengo model in its place. This connection requires translating the symbolic



representations used in ACT-R into spiking patterns and vice-versa, since communication in ACT-R is via chunks and communication in Nengo is via spikes.

Since Nengo provides access to the NEF, this mapping from symbols to population spike trains is facilitated by Eqs 1 and 3 described above for mapping vectors to population spike trains. We simply need to map the symbolic representation of a chunk into a vector and back again. In theory, this could be as simple as having a separate dimension in the vector for every possible chunk, or as sophisticated as using Vector Symbolic Architectures (Gayler, 2006). For example, the following code maps the chunk 'state:A' to [1,0,0], 'state:B' to [0,1,0], and 'state:C' to [0,0,1] and vice-versa. Note that the mapping from vector to chunk must take into account the representational noise introduced by the spiking neurons.

```
class Translator:
    def convertToVector(self,model):
        chunk=str(model.input)
        if chunk=='state:A': return [1,0,0]
        elif chunk=='state:B': return [0,1,0]
        elif chunk=='state:C': return [0,0,1]
        else: return [0,0,0]
    def applyVector(self,model,vector):
        mx=max(vector)
        if mx<0.3: model.output=None
        elif mx==vector[0]: model.output=Chunk('state:A')
        elif mx==vector[1]: model.output=Chunk('state:B')
        elif mx==vector[2]: model.output=Chunk('state:C')
```

INTEGRATED SIMULATION

To demonstrate this integration, we can create a Nengo implementation of an ACT-R buffer and connect it to an ACT-R model. For simplicity, the ACT-R model is of a set of three production rules which causes the goal buffer to cycle through three possible values (from state:A to state:B to state:C and back to state:A and so on). This simplistic model is sufficient to demonstrate communication from the ACT-R portion of the model to the Nengo portion and back again.

```
from ccm.lib.actr import *
class Model(ACTR):
    goal=Buffer()

    def production1(goal='state:A'):
        goal.set('state:B')
    def production2(goal='state:B'):
        goal.set('state:C')
    def production3(goal='state:C'):
        goal.set('state:A')
```

Once this model is defined, it can be created within Nengo. This involves the helper function `nengo.create` which is provided by CCMSuite and ensures that time in the ACT-R model is synchronized with time in the Nengo simulation. Once the model is created, a Nengo origin and termination are defined that use the defined mapping between ACT-R symbols and Nengo spike trains given above. Once these origins and terminations are defined, they are treated exactly as any other in Nengo, allowing neural models to be built and connected to them via either the Nengo graphical user interface or through the scripting system.

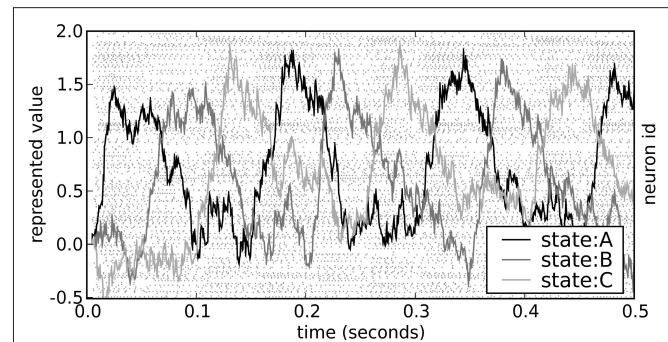


FIGURE 6 | Spike pattern and vector decoding of a neural population implementing an ACT-R goal buffer. Dots indicate spike times for each neuron in the goal buffer, arranged along the y-axis. The three lines show the three-dimensional value decoded from the spikes using Eq. 3. The three dimensions correspond to the three possible values for the buffer, showing that the represented value cycles through the three states.

```
import ccm
model = ccm.nengo.create(Model)
goal = model.getNode('goal')
goal.createOrigin('output',Translator())
goal.createTermination('input',Translator())
```

For this case, we implement the buffer using a three-dimensional integrator of the same type as that shown in Figure 3. This consists of 300 LIF neurons in a single neural group which integrates the value provided by ACT-R and outputs the current stored value back to ACT-R. These neurons are configured as per section “Programmatic Model Creation”

```
goalBuffer=ef.make("GoalBuffer",neurons=300,
                    dimensions=3)

M=[[1,0,0],[0,1,0],[0,0,1]]
goalBuffer.addDecodedTermination("input",M,tauPSC=0.007,
                                modulatory=False)
goalBuffer.addDecodedTermination("feedback",
                                M,tauPSC=0.007,
                                modulator=False)

model.addProjection(goalBuffer.getOrigin('X'),
                    goalMemory.getTermination('feedback'))
model.addProjection(goalBuffer.getOrigin('X'),
                    goal.getTermination('input'))
model.addProjection(goal.getOrigin('output'),
                    goalMemory.getTermination('input'))
```

The behavior of this model is shown in Figure 6. The neural group maintains the stored value over time, and then quickly changes this value when requested by the ACT-R production system. Importantly, the behavior of the model is robust over the time frame expected by ACT-R.

DISCUSSION

Nengo greatly facilitates the creation of complex neural circuits. The use of the NEF provides a general-purpose framework for representing information in spiking neurons that is flexible enough to support a wide variety of neuron models. The way in which the NEF systematically relates high-level information processing

to electro-physiology facilitates modeling of complex circuits and validation against both behavioral and electro-physiological data. Finally, the integrated Python scripting language, with its emphasis on readability and rapid development, makes it ideal for quickly creating models and exploring model variations.

This system is also supported by a rich graphical user interface suitable for introducing new users in, for example, classroom situations. Common tasks are supported directly by the user interface, and Python scripting offers a highly readable syntax for more complex situations without extensive language-specific training. Nengo is currently being used in a graduate-level course on the NEF, and students without previous Python exposure are able to make use of it and the user interface to create complex models, including modeling sensorimotor control using Kalman filters and sequence recognition in birdsong. Importantly, having the Python scripting available means that both experienced researchers and new students can use Nengo effectively.

REFERENCES

- Anderson, J. R., Fincham, J. M., Qin, Y., and Stocco, A. (2008). A central circuit of the mind. *Trends Cogn. Sci.* 12, 136–143.
- Anderson, J. R., and Lebiere, C. (1998). *The Atomic Components of Thought*. Mahwah, Erlbaum.
- Angelaki, D. E., McHenry, M. Q., Dickman, J. D., Newlands, S. D., and Hess, B. J. M. (1999). Computation of inertial motion: neural strategies to resolve ambiguous otolith information. *J. Neurosci.* 19, 316–327.
- Conklin, J., and Eliasmith, C. (2005). An attractor network model of path integration in the rat. *J. Comput. Neurosci.* 18, 183–203.
- Dormand, J. R., and Prince, P. J. (1980). A family of embedded Runge–Kutta formulae. *J. Comput. Appl. Math.* 6, 19–26.
- Eliasmith, C., and Anderson, C. (2003). *Neural Engineering: Computation, Representation, and Dynamics in Neurobiological Systems*. Cambridge, MIT Press.
- Eliasmith, C., Westover, M. B., and Anderson, C. H. (2002). A general framework for neurobiological modeling: an application to the vestibular system. *Neurocomputing* 46, 1071–1076.
- Fischer, B. (2005). A model of the computations leading to a representation of auditory space in the midbrain of the barn owl. PhD thesis. St Louis, Washington University in St Louis.
- Gayler, R. W. (2006). Commentary: vector symbolic architectures are a viable alternative for Jackendoff's challenges. *Behav. Brain. Sci.* 29, 78–79.
- Georgopoulos, A. P., Schwartz, A. B., and Kettner, R. E. (1986). Neuronal population coding of movement direction. *Science* 233, 1416–1419.
- Gruber, A. J., Solla, S. A., Surmeier, D. J., and Houk, J. C. (2003). Modulation of striatal single units by expected reward: a spiny neuron model displaying dopamine-induced bistability. *J. Neurophysiol.* 90, 1095–1114.
- Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572.
- Keener, J., and Sneyd, J. (1998). *Mathematical Physiology*. New York, Springer.
- Kuo, D., and Eliasmith, C. (2005). Integrating behavioral and neural data in a model of zebrafish network interaction. *Biol. Cybern.* 93, 178–187.
- La Camera, G., Rauch, A., Lüscher, H.-R., Senn, W., and Fusi, S. (2004). Minimal models of adapted neuronal response to in vivo-like input currents. *Neural Comput.* 16, 2101–2124.
- Rieke, F., Warland, D., de Ruyter van Steveninck, R., and Bialek, W. (1999). *Spikes: Exploring the Neural Code*. Cambridge, MIT Press.
- Salinas, E., and Abbott, L. F. (1994). Vector reconstruction from firing rates. *J. Comput. Neurosci.* 1, 89–107.
- Seung, H. S. (1996). How the brain keeps the eyes still. *Proc. Natl. Acad. Sci. U.S.A.* 93, 13339–13344.
- Singh, R., and Eliasmith, C. (2006). Higher-dimensional neurons explain the tuning and dynamics of working memory cells. *J. Neurosci.* 26, 3667–3678.
- Stewart, T. C., and Eliasmith, C. (2009). Compositionality and biologically plausible models. In *Oxford Handbook of Compositionality*, W. Hinzen, E. Machery and M. Werning, eds (Oxford University Press).
- Stewart, T. C., and West, R. L. (2007). Deconstructing and reconstructing ACT-R: exploring the architectural space. *Cogn. Syst. Res.* 8, 227–236.
- Winter, D. A. (1990). *Biomechanics and Motor Control of Human Movement*. John Wiley & Sons, New Jersey.

ACKNOWLEDGMENTS

We thank Shu Wu for developing Nengo's graphical user interface.

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

Received: 14 September 2008; paper pending published: 10 October 2008; accepted: 20 February 2009; published online: 24 March 2009.

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

Copyright © 2009 Stewart, Tripp and Eliasmith. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Technical integration of hippocampus, basal ganglia and physical models for spatial navigation

Charles Fox^{1*}, Mark Humphries¹, Ben Mitchinson¹, Tamas Kiss², Zoltan Somogyvari², Tony Prescott¹

¹ Adaptive Behaviour Research Group, Department of Psychology, University of Sheffield, Sheffield, UK

² Department of Biophysics, KFKI Research Institute for Particle and Nuclear Physics, Hungarian Academy of Sciences, Budapest, Hungary

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Michael E. Hasselmo, Boston
University, USA
Eilif Müller, Brain Mind Institute, EPFL,
Switzerland

*Correspondence:

Charles Fox, Adaptive Behaviour
Research Group, Department of
Psychology, Faculty of Pure Science,
University of Sheffield, Sheffield,
South Yorkshire, Western Bank,
Sheffield S10 2TP, UK.
e-mail: charles.fox@sheffield.ac.uk

Computational neuroscience is increasingly moving beyond modeling individual neurons or neural systems to consider the integration of multiple models, often constructed by different research groups. We report on our preliminary technical integration of recent hippocampal formation, basal ganglia and physical environment models, together with visualisation tools, as a case study in the use of Python across the modelling tool-chain. We do not present new modeling results here. The architecture incorporates leaky-integrator and rate-coded neurons, a 3D environment with collision detection and tactile sensors, 3D graphics and 2D plots. We found Python to be a flexible platform, offering a significant reduction in development time, without a corresponding significant increase in execution time. We illustrate this by implementing a part of the model in various alternative languages and coding styles, and comparing their execution times. For very large-scale system integration, communication with other languages and parallel execution may be required, which we demonstrate using the BRAHMS framework's Python bindings.

Keywords: hippocampus, basal ganglia, spatial navigation, place cells, plus-maze, BRAHMS, Python

INTRODUCTION

As computational resources inexorably grow, computational neuroscience is increasingly moving beyond modeling individual neurons or neural systems to consider the integration of multiple models, often constructed by different research groups. At the software level there is a drive towards interoperability of simulators at both model specification (Goddard et al., 2001) and run-time stages (Cannon et al., 2007). However, these efforts have concentrated on creating small networks of different multi-compartment models (Gleeson et al., 2007), or large networks of different single-compartment spiking neuron models (Cannon et al., 2007).

Our focus here is on a third strand that can take advantage of growth in computing power: the integration of multiple neural models that form components of a brain-wide system, and the testing of that integrated model in an embodied form. Embodiment often takes the form of a robot and a test environment, whether simulated or real. Requiring the neural models to generate appropriate behavioural output using only inputs available in the environment is a strong test of the proposed computations of that neural system (Humphries et al., 2005; Prescott et al., 2006). In such large simulations, development time is as much an issue as computation time – to implement and test the models, construct simulated environments, implement realistic sensors, and so on. This paper shows how Python provides an excellent solution to both development and computation time problems; we also discuss how Python can work with platforms designed for such large-scale integration (Mitchinson et al., 2008).

As a case study, we report on our preliminary integration of recent hippocampal formation and basal ganglia models, both proposed components of the neural system for spatial navigation (Redish and Touretzky, 1997). The hippocampal formation's role in spatial navigation is not controversial: "place" cells within

CA1/CA3 encode position in space (O'Keefe and Conway, 1978; Wiener, 1996); "grid"-cells in entorhinal cortex (EC) provide metric information for path-integration via a tessellating rhomboid pattern (Hafting et al., 2005; McNaughton et al., 2006); and hippocampal lesions impair (but not necessarily abolish) rats' abilities to navigate in open environments (Whishaw, 1998). The basal ganglia's main input nucleus – the striatum – is a major target of hippocampal formation output, and also appears necessary for unimpaired spatial navigation: lesioning the connecting fibres impairs accurate navigation in open environments (see e.g. Devan et al., 1996; Gorny et al., 2002; Whishaw et al., 1995), and blocking plasticity in the region of striatum targeted by hippocampal fibres prevents acquisition of paths to targets (Sargolini et al., 2003; Smith-Roe et al., 1999).

A recurring theme in the basal ganglia literature is that they form a selection mechanism for motor programs (Hikosaka et al., 2000; Mink and Thach, 1993) or, more generally, for "actions" (Redgrave et al., 1999). Thus, the specific hypothesis underlying our integrated model is that the basal ganglia select movement direction based on current spatial position provided by the hippocampal formation input.

The system described below is a preliminary technical integration of the action-selecting basal ganglia model of Gurney et al. (2001a,b) with the hippocampal navigation model of Ujfalussy et al. (2008). The basal ganglia model may be used to select between any types of action, but simple predefined salencies between two target locations are currently used. The hippocampus model may run using any form of sensory input: at present we use visual input, but report on the implementation of physical simulation of tactile whisker-like sensors as an example of developing advanced sensors in Python, which could form a further input in future. Neither the inputs to the models or the placeholder function connecting them are intended

to be biologically realistic at this stage. The neural models control a mobile rat-like robot in a standard plus-maze environment with external landmarks, all implemented in a 3D simulator built using existing Python modules. The purpose of this paper is to illustrate a complete neural and physical simulation system, detailing the specific libraries and packages in the tool-chain that were found useful, and not to make any new claims about the biological models. We hope that it will provide a guide for others who wish to implement similar systems, as it can be difficult for newcomers to select the best tools from the plethora of open-source Python extensions.

COMPUTATIONAL MODELS

We are updating prior models of hippocampal formation-basal ganglia interactions (Arleo and Gerstner, 2000; Chavarriaga et al., 2005) by including the entire basal ganglia circuit and by using a grid-cell driven model of hippocampus. In addition, prior models assumed a direct, modifiable, projection from place cells to the striatum (Arleo and Gerstner, 2000; Chavarriaga et al., 2005). However, such a projection, if it exists, is minor compared to input from other regions of the hippocampal formation, particularly the subiculum, suggesting further stages of processing between the basic

representation of position and the striatum (see e.g. Groenewegen et al., 1999; van Groen and Wyss, 1990). In the current integrated system, we provide a simple spatial decoding scheme as a proxy for detailed models of the intervening structures to follow. We do not here present new results from the individual models (Gurney et al., 2001a,b; Ujfalussy et al., 2008), but report on systems integration at a technical level using Python and BRAHMS.

BASAL GANGLIA

The basal ganglia are a group of inter-connected subcortical nuclei, which receive massive convergent input from most regions of cortex, and output to targets in the thalamus and brainstem (Bolam et al., 2000). We have previously shown how this combination of inputs, outputs, and internal circuitry implements a neural substrate for a selection mechanism (Gurney et al., 2001a,b, 2004; Humphries and Gurney, 2002; Humphries et al., 2006; Prescott et al., 2006). **Figure 1** illustrates the macro- and micro-architecture of the basal ganglia, highlighting three key ideas underlying the selection hypothesis: that the projections between the neural populations form a series of parallel loops – *channels* – running through the basal ganglia from input to output stages (Alexander

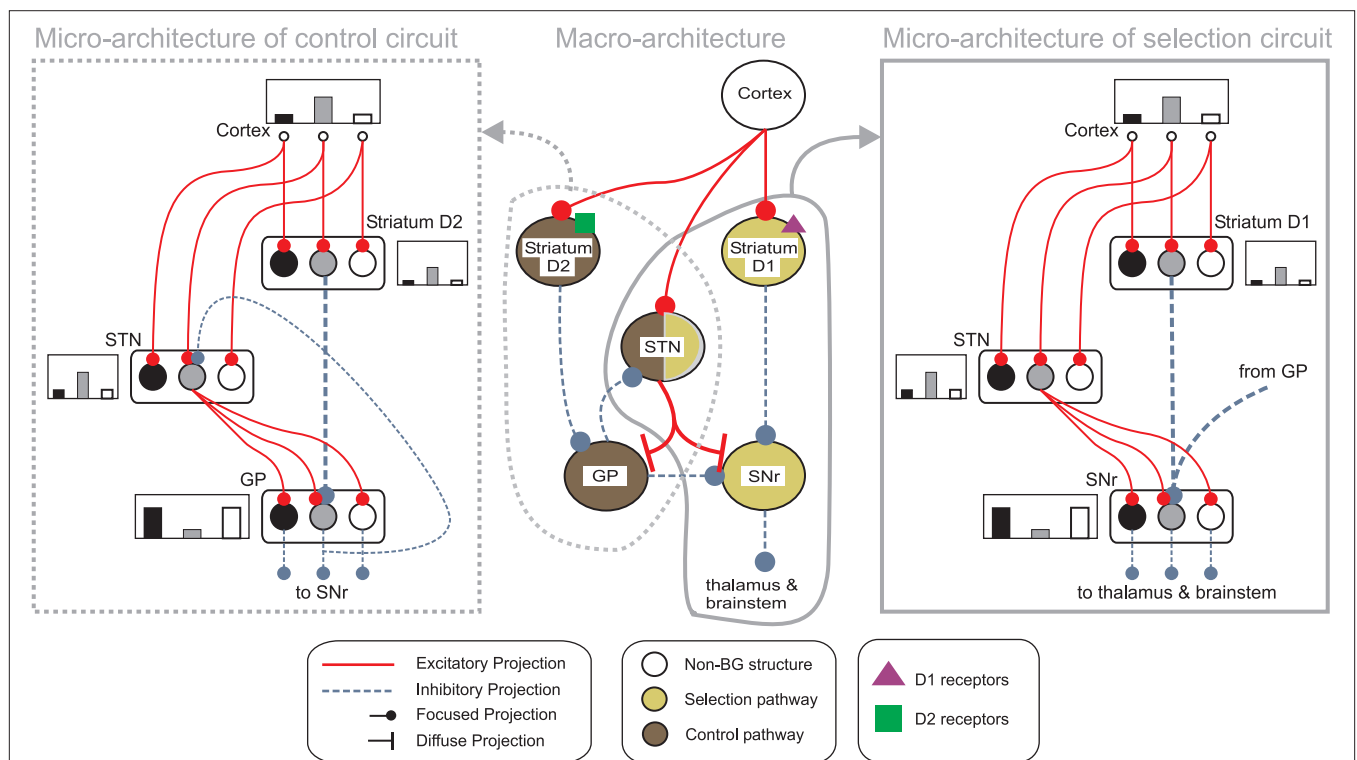


FIGURE 1 | Architecture of the basal ganglia model. The main circuit (centre) can be decomposed into two copies of an off-centre, on-surround network: a selection pathway (right) and a control pathway (left). Three parallel loops – channels – are shown in both pathways, with example activity levels in the bar charts to illustrate the relative contributions of the nuclei (the three channels are colour-coded black/grey/white, corresponding to the example bar charts). Note that, for clarity, full connectivity is only shown for the second channel. Briefly, the selection mechanism works as follows. Constant inhibitory output from substantia nigra pars reticulata (SNr) provides an “off” signal to its widespread targets in the thalamus and brainstem. Cortical inputs representing competing saliences are

organised in separate channels (groups of co-active cortical neurons), which project to corresponding populations in striatum and STN. In the selection circuit, the balance of focussed (one-to-one) inhibition from striatum and diffuse (one-to-many) excitation from STN results in the most salient input suppressing the inhibitory output from SNr on that channel, signalling “on” to that SNr channel’s targets. In the control circuit, a similar overlap of projections to GP exists, but the feedback from GP to the STN acts as a self-regulating mechanism for the activity in STN, which ensures that overall basal ganglia activity remains within operational limits as more and more channels become active. For quantitative demonstrations of this model, see Gurney et al. (2001b, 2004) and Humphries et al. (2006).

and Crutcher, 1990); that the total activity from cortical sources converging at each channel of the striatum encodes the salience of the action represented by that channel; and that the selection of an action is signalled by a process of *disinhibition* – the selective removal of tonic inhibition from cells in the basal ganglia's target regions that encode the action (Chevalier and Deniau, 1990).

We use here the population-level implementation of this model from Gurney et al. (2001b). The average activity of all neurons comprising a channel in a population is represented by a single unit that changes according to

$$\tau \dot{a} = -a + u \quad (1)$$

where τ is a time constant and u is summed, weighted input. We use $\tau = 40$ ms. The normalised firing rate y of the unit is given by a piecewise linear output function

$$y = F(a, \epsilon) = \begin{cases} 0 & a \leq \epsilon \\ a - \epsilon & \epsilon < a < 1 + \epsilon \\ 1 & a \geq 1 + \epsilon \end{cases} \quad (2)$$

The following describes net input u_i and output y_i for the i th channel of each structure, with n channels in total. Net input is computed from the outputs of the other structures, except cortical input c_i to channel i of striatum and subthalamic nucleus (STN). The striatum is divided into two populations, one of cells with the D1-type dopamine receptor, and one of cells with the D2-type dopamine receptor. Many converging lines of evidence from electrophysiology, mRNA transcription, and lesion studies suggest a functional split between D1- and D2-dominant projection neurons and, further, that the D1-dominant neurons project to SNr, and the D2-dominant neurons project to globus pallidus (GP; Gerfen and Wilson, 1996; Surmeier et al., 2007).

Activation of these receptors has opposite effects on striatal input: D1 activation increases the efficacy of the input; D2 activation decreases the efficacy of the input (see Gurney et al., 2001b, for full details). Let the level of tonic dopamine be λ : then the increase in synaptic efficacy due to D1 receptor activation is given by $(1 + \lambda)$; the decrease in synaptic efficacy due to D2 receptor activation is given by $(1 - \lambda)$. Normal dopamine levels were indicated by $\lambda = 0.2$, and dopamine-depletion by $\lambda = 0$, following previous work (Gurney et al., 2001b; Humphries and Gurney, 2002). The full model is thus given by:

$$\text{Striatum D1: } u_i^{d1} = c_i(1 + \lambda) \quad (3)$$

$$y_i^{d1} = F(a_i^{d1}, 0.2) \quad (4)$$

$$\text{Striatum D2: } u_i^{d2} = c_i(1 - \lambda) \quad (5)$$

$$y_i^{d2} = F(a_i^{d2}, 0.2) \quad (6)$$

$$\text{STN: } u_i^{stn} = c_i - y_i^{sp} \quad (7)$$

$$y_i^{stn} = F(a_i^{stn}, -0.25) \quad (8)$$

$$\text{Globus pallidus: } u_i^{gp} = 0.9 \sum_i y_i^{stn} - y_i^{d2} \quad (9)$$

$$y_i^{gp} = F(a_i^{gp}, -0.2) \quad (10)$$

$$\text{SNr: } u_i^{snr} = 0.9 \sum_i y_i^{stn} - y_i^{d1} - 0.3 y_i^{gp} \quad (11)$$

$$y_i^{snr} = F(a_i^{snr}, -0.2) \quad (12)$$

Full details for the chosen constants can be found in (Gurney et al., 2001a), and are summarised here. Thresholds for striatal output were set $\epsilon > 0$ so that a large positive input would be required for any output from these neurons, modelling the large input required to push the striatal projection neuron into its firing-ready “up-state” (Gerfen and Wilson, 1996). The STN, SNr, and GP all had $\epsilon < 0$, as each of these has tonic output at rest (Bolam et al., 2000). Non-unity weights (0.3, 0.9) on inputs were set to be within analytically-derived bounds for stable operation of the model (Gurney et al., 2001a).

We used forward Euler to simulate this system for a two-channel model, with the same time-step of 10 ms as was used for the discrete equations of the hippocampus model (see below).

The model was implemented in Python using an object-oriented hierarchy. Neuron objects contain Dendrite objects, which store modulated and unmodulated weights, and references to parent neurons. Neurons also store their parameters (ϵ, τ, s) (where s is the sign of dopamine action) and state (u, a, y). The neuron class contains methods to apply dopamine modulation and determine the unit's output. A Population class groups units together, and contains methods to instantiate sets of one-to-one (e.g. GP→SNr) or diffuse (e.g. STN→SNr) links to other Populations. These methods automatically construct Dendrite objects and update references.

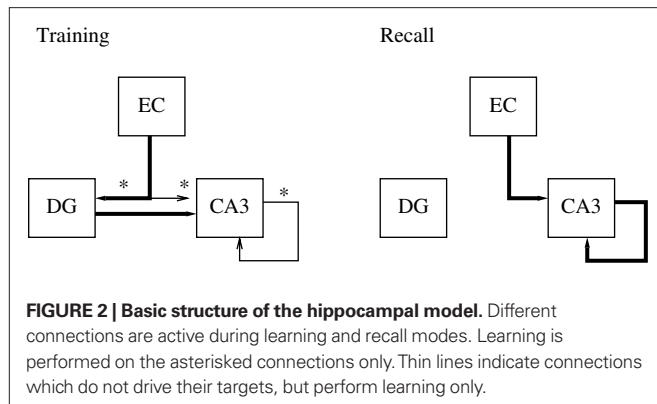
We have found Python's default and named arguments to be especially useful in this type of modeling. Neurons may be given many default parameter values which remain invisible in the user-level code unless specifically overridden. For example, the sign s of dopamine action is assumed to be zero (meaning no effect) unless an easy-to-read named parameter is passed:

```
STN = Population(n, epsilon = -0.25)
STN.addParPopOneToOne(Cx, w_Cx_STN)
D2 = Population(n, dopamineAction = -1, epsilon = 0.2)
D2.addParDopamine(SNc)
D2.addParPopOneToOne(Cx, w_Cx_D2)
```

HIPPOCAMPAL FORMATION MODEL

The hippocampal formation comprises the EC, dentate gyrus (DG), fields CA3 and CA1 of the hippocampus proper, and the subiculum. These form a feed-forward loop of connections that starts and ends in the EC. Though all structures are thought to contribute, the hippocampal model of Ujfalussy et al. (2008) instantiates just the minimum putatively required for the hippocampal formation to act as a memory store (following Treeves and Rolls, 1994); for spatial navigation, the memory formed is considered the place code created by the place cells. **Figure 2** shows the basic structure, formed by just the EC, DG, and CA3.

Following previous models (e.g. Treeves and Rolls, 1994), the model of Ujfalussy et al. (2008) makes three key assumptions. First, the DG region is a preprocessing stage for CA3, acting as a competitive network that creates a sparse and clustered code of the pre-synaptic EC input, which – similarly to other neocortical regions – realises a denser representation. This sparse, orthogonal code is in turn used as a teaching signal for the CA3 region. Second, the CA3 region acts as an auto-association memory, which stores memory traces in its extensive recurrent local collaterals for later retrieval. Third, many previous hippocampal models (Arleo and Gerstner, 2000; Rolls, 1995; Treeves and Rolls, 1994) assume that the hippocampus operates in two distinct modes during learning



and retrieval, which are also incorporated into the present model. As in these models, switching between the two modes is performed manually (contrary to models such as Hasselmo et al. (1995, 1996) which explicitly address the separation between learning and recall). **Figure 2** shows the connections that change between the modes.

Entorhinal cortex

Grid cells in EC are modeled as having firing rates that are functions of the agent's actual physical position $\mathbf{r} = (x, y)$ in the simulated environment. Grid cells each have two parameters, determining the phase and scale of their receptive fields. The output of the i, j th grid cell is

$$g(\mathbf{r})_{i,j} = \sqrt{\frac{1}{3} \sum_k \cos^2[\mathbf{w}_k \cdot (s_i \mathbf{r} - \theta_j)]} \quad (13)$$

where s_i and θ_j are the i th scale factor and j th phase shift respectively, and $\{\mathbf{w}_k\}_{k=1:3}$ are unit vectors at 60° from each other. We used an ordered set of scale factors from 0.5 to 2.5 in steps of 0.5, and an ordered set of phases from 0 to π in steps of $\pi/5$; i and j are indices into these sets. **Figure 3** shows that Eq. 13 produces receptive fields with the characteristic rhomboid or “double triangle” tessellation of grid cells (Hafting et al., 2005).

The EC relays input from several cortical areas (Marr, 1971) to the hippocampal formation, and is thus often treated (e.g. Rolls, 1995; Rolls et al., 2006), as in the present model, as the input source for all sensory information. Thus, as well as comprising a large population of grid cells, EC is modelled with an additional population of sensory cells. We use 100 visual cells, whose activations are set by 10×10 grayscale images.

Dentate gyrus

The DG is thought to perform a principal-components-like dimensionality reduction of input from the EC (Lorincz, 1998). Writing w_{ij} for weights on inputs y_j , the j th DG unit's neural activation is given by

$$a_j = \sum_i w_{ij} y_i \quad (14)$$

where the sum is taken over all EC inputs. Output firing rates $\{y_j\}$ are given by a m -best function $\{y_j\} = F\{a_j\}$ which preserves the m largest activations, linearly re-maps them to the interval $[0, 1]$, and sets the others to zero.

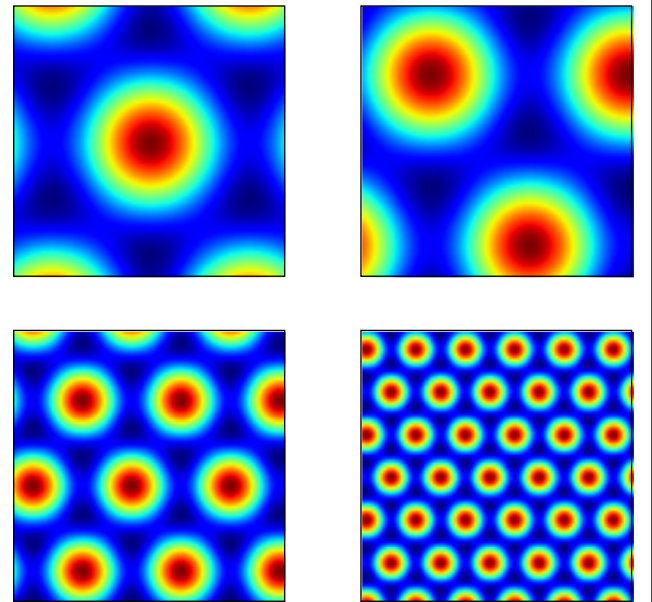


FIGURE 3 | Grid cell receptive fields from the model, over physical 2D space. These are plotted with Pylab's Matlab-style `imagesc` command.

During the training phase only, every $EC \rightarrow DG$ weight is updated at each time-step using the standard Hebbian learning rule,

$$\Delta w_{ij} = \alpha y_j (y_i - w_{ij}) \quad (15)$$

where α is the learning rate, and again j represents the DG cell population and i represents the afferent EC population.

CA3 place cells

CA3 functions differently during training and recall. During training, CA3 is driven only by input from DG; hence unit activity is updated according to Eq. 14 with j representing the CA3 cell population and i representing the afferent DG population. CA3 output is computed from these activations with the same m -best function used for the DG output.

Despite being driven by DG only, no learning is performed on this connection. Instead, learning is performed on the otherwise dormant $EC \rightarrow CA3$ and $CA3 \rightarrow CA3$ pathways. $EC \rightarrow CA3$ weights are altered by Eq. 15, where y_i is EC output, and y_j is CA3 output. Following Rolls (1995), each recurrent $CA3 \rightarrow CA3$ weight is altered by the gated Hebbian rule,

$$\Delta w_{ij} = \alpha y_i y_j (1 - w_{ij}) - \beta w_{ij} \quad (16)$$

where β sets the “forgetting rate”, and i and j now both refer to cells within the CA3 population.

During the recall phase, the EC input is used to initiate retrieval of a stored memory pattern. First, the activation of CA3 units is computed from the EC inputs only, using Eq. 14 with j representing the CA3 cell population and i representing the afferent EC population; their output is then computed using the m -best function. Second, this initial output vector was used as the cue to retrieve the memory trace in the CA3 autoassociative network. The activity

of the k th CA3 unit is then the weighted sum of total output from the EC and the recurrent connections

$$a_k = \sum_{i \in \text{EC}} w_{ik} y_i + \sum_{j \in \text{CA3}} w_{jk} y_j \quad (17)$$

with CA3 output y_k again computed by applying the m -best function. Activation (Eq. 17) and output calculations of the CA3 units were iterated I times to bring the CA3 close to an attractor state as in Hopfield-style networks (Hopfield, 1984).

We used eight DG cells and 30 CA3 cells with: learning rate $\alpha = 0.05$, forgetting rate $\beta = 0.00002$, sparsity $m = 20$ and $I = 5$ recurrent iterations.

Implementation in Python

The hippocampal model uses simple rate-coded units and linear weights, in contrast to the basal ganglia's leaky integrators. For this reason the population activations and firing rates are amenable to fast implementation as vectors rather than as attributes of individual objects. Multiplication of population firing rates by weight matrices may then be performed by matrix algebra. This style of programming is common in Matlab, and may be performed in Python using the Numpy library¹. Numpy emulates much of Matlab's matrix syntax, including notation for slicing matrices (e.g. $A = M[:, 1:5]$), addressing ($M[2, 3] = 4$) and performing operations such as element-wise addition ($B = A + 1$) as well as matrix algebra ($C = \text{dot}(A, B)$).

We have also made use of two further libraries: SciPy² provides a library of higher-level mathematical functions similar to Matlab's toolboxes; and Pylab³ provides interactive plotting commands. For example, **Figures 3 and 4** were plotted using Pylab. Pylab emulates many of Matlab's graphics commands including 2D and 3D graphs, and image viewers. The Matlab application programmer interfaces (APIs) are replicated almost literally, using the same function names and argument conventions where possible, such as `clf`, `plot` and `imagesc`.

Training

Training of the EC→DG, EC→CA3 and CA3→CA3 weights was performed over five epochs. Weights were initialised to random real values from a uniform distribution ranging from 0 to 1. Grid and visual cell input data was collected from a simulated robot moving to a sequence of pre-determined points in a plus-maze environment (see "Building and Using the 3D Simulator" for simulator details). The robot enters the maze from the open arm, then visits each of the other arms in turn and comes to rest at the center. About 1,500 data points were sampled during this motion. After training, Python's standard `cPickle` library provided a simple way to serialise and save the trained Hippocampus object, using only the following code:

```
file = open("myfile", "r")
cPickle.dump(myObject, file)
file.close()
```

The effect of training the hippocampus model with the grid cell and visual input was to generate place fields in CA3, such as those

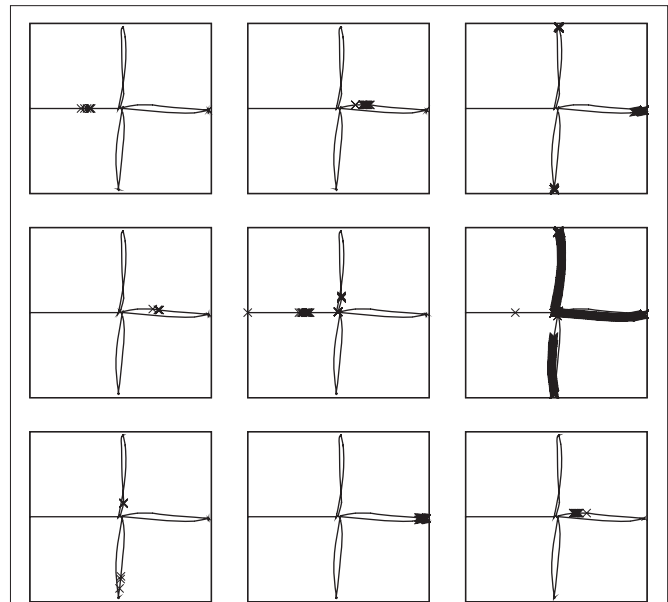


FIGURE 4 | Receptive fields for nine CA3 place cells, superimposed on the robot's path around the plus maze. Crosses show locations where cells firing rates are in the top 5% of their activity throughout the path. Plotting was performed with Pylab's `plot` command, which has similar syntax to Matlab.

shown in **Figure 4**, which shows the locations of strongest firing for nine of the 30 CA3 cells, superimposed on the robot's path. Of the 30 cells simulated, 11 responded to single places, 13 to two or more places, and 6 were silent at all places (where a "single" place is defined as a contiguous series of strong activations).

DECODING PLACE

We used a placeholder function for decoding hippocampal place representations into striatal input, as a proxy for detailed models of the intervening structures (e.g. CA1, subiculum) to follow. A simple linear regression was used to find a linear mapping from the vector of place cell activations to the Cartesian (x, y) spatial positions. SciPy provides such regression in its linear algebra sub-package (function `linalg.lstsq`).

BUILDING AND USING THE 3D SIMULATOR THE PLUS-MAZE ENVIRONMENT

We used Python to construct a plus-maze environment in which to test our current and future forms of the integrated basal ganglia-hippocampus model. The plus-maze environment was chosen as it is widely used for neural recording studies that probe the roles of striatum, hippocampus, and their interactions in spatial tasks (Albertin et al., 2000; Khamassi, 2007; Mulder et al., 2004; Tabuchi et al., 2000, 2003). Following these studies, the simulated plus-maze comprised a symmetric arrangement of walled arms, and two extra-mazecues (**Figure 5**).

The neural model was used to control the "ICEAsim" simulated robot, a differential wheels robot in a rat-like form, created by Cyberbotics (Lausanne) for the ICEA project⁴. ICEAsim was

¹www.numpy.scipy.org

²www.scipy.org

³www.matplotlib.sourceforge.net

⁴www.iceaproject.eu

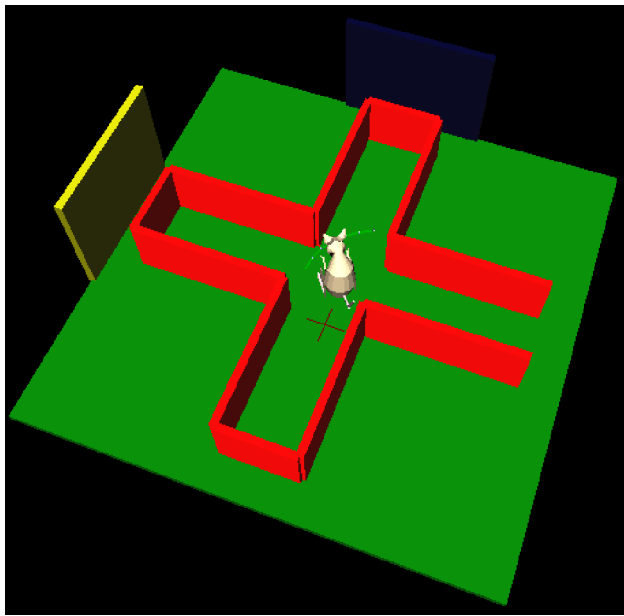


FIGURE 5 | The simulated plus-maze environment. The hippocampus reports the current estimated location, shown by the cross on the floor. When this estimate is close to the center of the plus-maze, the basal ganglia is consulted for an action to turn. 3D physical simulation and visualisation uses PyODE and Pivy.

initially created under Cyberbotics' Webots simulator (Michel, 2004), but was readily imported into a Python simulation via the standard VRML format. Wheel commands are sent via a higher-level (and non-biological) function which takes as input a requested target location to which to move. Our implementation of ICEAsim added two whisker sensors, which output the angle and curvature at their bases for use in tactile perception algorithms (see "Python Physics Implementation" and "Comparison to PyRobotics"). We added realistic whisker-like sensors as the basis for future studies: while rats can successfully navigate in the dark, and corresponding place fields are formed in the hippocampal formation, this has been attributed entirely to idiothetic (self-motion) cues (Quirk et al., 1990; Rossier et al., 2000); surprisingly little attention has been paid to the potential role of rats' whiskers in constructing spatial maps in the dark.

For the purposes of this paper, we used a simple task and a placeholder function to test that the models were correctly implemented and technically integrated. After hippocampal training (a separate task, not involving basal ganglia), the robot was simply required to successfully navigate to the end of a maze arm, starting from the entrance of the maze. The basal ganglia model received input saliences $\{c_1, c_2\}$ on two channels, corresponding to two actions ("go to left arm" and "go to right arm"), and which – in this preliminary system – were assigned predefined time series. The placeholder function monitored the hippocampal position estimate, and when this estimate was close to the center of the maze, the action corresponding to the basal ganglia output channel (in SNr) with minimum value was selected and executed to completion. A "go to left arm" or "go to right arm" routine is called, which uses hippocampal output to estimate the required path to follow and sets

the robot's differential wheel speeds accordingly. **Figure 5** illustrates the simulated robot's behaviour: video of the robot's movement, and corresponding activity in the integrated neural models, are available as Supplementary Material. Future biological models of basal ganglia-hippocampus interactions may of course replace the predefined time series and placeholder function with more complex and ongoing interactions between the models, using the technical integration framework presented here.

PYTHON PHYSICS IMPLEMENTATION

To simulate tactile whisker sensors requires realistic physics modeling, as the precise bending (Birdwell et al., 2007), vibration (Ueno and Kaneko, 1994) and other dynamics (Ritt et al., 2006) of whiskers are crucial in making inferences from touch. The Open Dynamics Engine (ODE) is an excellent open-source (BSD license) physics engine, and we use the PyODE wrapper (pyode.sourceforge.net) to use it from Python. ODE provides primitive objects such as cubes, spheres and cylinders, which may be combined and transformed to produce objects such as the walls of the plus-maze and the parts of the robot. PyODE wraps all the major ODE functions for shapes, kinematics and collision handling, and provides access to ODE's standard set of flexible joints. We use the latter to construct rotating wheels, and whiskers. The whiskers are modelled as a series of spherical or cylindrical segments, connected by joints with rotational Hooke's law springs. ODE handles the constraint forces required to keep joints together automatically; however very small time steps (and hence long simulation times) are needed when the number of segments is above three. For example with three segments per whisker the simulation requires about 3 min to run stably on a 1.6-GHz machine; with four segments it requires about 10 min.

VISUALISATION

3D visualisation is important in robotics simulation, both to ensure that the simulation is behaving as intended, and also to provide realistic visual input to robot sensors, for processing by neural models.

OpenGL is a standard 3D graphics API⁵, and is implemented by the free software Mesa and by many hardware-specific graphics drivers. OpenGL provides low-level graphics commands to draw lines, triangles and polygons, and position lights and cameras. The OpenGL API is wrapped in Python by pyOpenGL (pyopengl.sourceforge.net).

Higher-level graphics commands – such as drawing cubes, cylinders and cones using scene graphs – are provided by the OpenInventor API, implemented by the free software Coin⁶. Coin has been wrapped for Python by the Pivy binding⁷ (Fahmy, 2006) which we use here. Pivy allows raw pyOpenGL commands to be mixed into its higher-level structures where necessary.

To simulate vision (for input to the hippocampus model) we read back images from simulated cameras attached to the robot. Pivy wraps Coin's `SoOffScreenRenderer` function to perform this task. (Modelers are advised that use of this function may be

⁵www.OpenGL.org

⁶www.coin3d.org

⁷www.pivy.coin3d.org

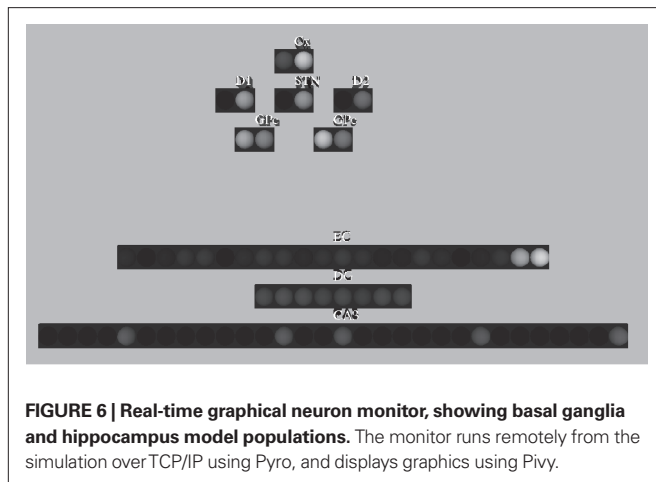


FIGURE 6 | Real-time graphical neuron monitor, showing basal ganglia and hippocampus model populations. The monitor runs remotely from the simulation over TCP/IP using Pyro, and displays graphics using Pivy.

incompatible with the use of direct rendering on some graphics hardware. Disabling direct rendering solves this problem but reduces execution speed.) If graphical output is required in video form only – such as for presentation but not as data for neural models – the free program Yukon⁸ is able to export OpenGL graphics to .avi movies, whose frame-rate and resolution may be edited with the free Aviflix program⁹.

In addition to the main 3D representation of the physical world, it is often useful to attach additional graphical monitors to show the internal state of the neural models in real-time. Pivy – like Coin – takes control of program flow, calling back user functions to draw and update the world. When multiple displays are required – or when handing over control is too intrusive – it is useful to instantiate several processes running Pivy. We use the Python Remote Objects package (PyRo¹⁰) to handle communication between such processes. PyRo allows an object from one process to appear on another as if it was resident there, allowing function to be called and data to be passed easily. PyRo processes communicate via TCP/IP so the monitors may run on different machines to the main simulation. **Figure 6** shows a screen-shot of the state visualisation tool we built for the integrated basal ganglia-hippocampus model.

COMPARISON TO PYROBOTICS

Our simulation is constructed using Python wrappers for ODE and OpenGL. An alternative approach to simulation would be to use the higher-level Player interface and Gazebo simulator¹¹ which are available through the PyRobotics¹² integrated robotics simulator. PyRobotics allows worlds and robots to be built from standard components using XML specification, and controllers written in Python. This approach is recommended for simulations requiring standard physics, but our use of the lower-level APIs was determined by the need to write custom physics code for the whisker sensors. Whiskers are difficult to model and coding with ODE directly allows finer

control over the contacts and forces that are simulated than would be available in a higher-level simulator. Our custom simulations are not intended to be an integrated robotics simulator, but they serve as an example of lower-level PyODE and Pivy simulation.

SPEED COMPARISONS

Python is often thought of as being a “slow language” and if this is the case then it would be a barrier to its use in large scale, computationally-intensive neural simulations. However, various libraries and programming styles exist that can improve performance. We investigated a variety of these to evaluate Python’s suitability for large simulations. We chose execution of the previous basal ganglia model alone as a benchmark representative of many neural simulation tasks, and used a model with 100 channels running over 1,000 time steps to provide a sizable task requiring time of the order of seconds. Neural models are commonly implemented in high-level Matlab code (and its open-source equivalent, Octave), or in low-level C code. C code allows and requires the user to perform their own memory management, leading to greater development time but often faster running times. We re-implemented the basal ganglia model in these languages, writing the fastest code our skills allowed.

In addition to the object-oriented Python model described earlier, we also re-implemented Python models using the Numpy, Pyrex and Weave libraries. As described above, Numpy provides Matlab-like data structures, operations and syntax, to the extent that the Matlab program can be ported to Numpy with only minor syntactic modifications. Pyrex¹³ is a Python-like language for writing Python extension modules, which provides C-like manual typing and data structures. As with C, Pyrex increases development time by adding work to the programmer’s load, but may increase execution time as a result. Programming Pyrex is conceptually similar to writing C programs, but using a Python-like syntax and allowing very simple integration into pure Python code. Weave (part of SciPy) allows inline C code to be embedded directly into Python files, and its “converters” library automates data type conversion between languages. We implemented inline Weave code within the body of the main Numpy simulation loop.

Another way to improve Python speed is to use more advanced compilers and virtual machines. There is much current research into such tools but a popular system is Psyco¹⁴. We used Psyco to run the pure Python, object-oriented model (it has negligible effect on Numpy code, in which most of the computation is performed by external numerical C libraries).

Table 1 shows the average execution times for the above implementations (and a BRAHMS version discussed below). Execution was performed on a 1.6 GHz, 1.5 GB Ubuntu system and time averages were taken over five runs. No calls were made to platform-specific BLAS or random-number generator libraries within the simulation loops (such calls are not required or useful in implementing the basal ganglia model’s equations). It can be seen that for the Matlab and C-like programming styles (i.e. Numpy and Pyrex respectively) Python is about four times slower than the non-Python alternative. Weave is only a fraction slower than raw C, the overhead being due

⁸www.dbsservice.com/projects/yukon

⁹www.transcoding.org

¹⁰www.pyro.sourceforge.net

¹¹www.playerstage.sourceforge.net

¹²www.pyrobotics.org

¹³www.cosc.canterbury.ac.nz/greg.ewing/

¹⁴www.psyco.sourceforge.net

Table 1 | Computation times for the basal ganglia model implemented in different languages and programming formats.

Language/format	Time (s)
Object-oriented Python	66.1
As above, with Psycho	48.6
Octave	1.31
Numpy Python + BRAHMS	0.89
Numpy Python	0.82
Pyrex	0.22
Matlab	0.21
Scipy.weave.inline	0.05
Raw C	0.04

to type conversions. The object-oriented version has a much larger run-time – as expected of this style of programming – and the time is reduced by about 25% using Psycho.

These results suggest that Python is not inherently “slow” – a factor of four is not large in such comparisons – though it can be used to write slow but conceptually meaningful, human-readable, object-oriented code if desired. Alternatively, if human comprehension is less important, then Matlab-like and C-like programming styles can be used to regain speed. In most cases it is desirable to work on an easily comprehensible “reference implementation” of a model at first, then develop a faster implementation once the research is complete. Python eases this often difficult transition as Numpy, Weave and Pyrex commands may be gradually mixed into and replace the research code: the more traditional replacement of Matlab by C programs requires a complete rewrite from scratch.

SUBJECTIVE EXPERIENCES WITH Python

The above has considered architectural and computational features of Python and its associated libraries that are useful in embodied neural modeling. However these are not the only criteria for choosing a language for development: at least as important are the more subjective aspects of the system during development and debugging. Here we offer our experiences of hands-on development of the neural and physical models.

We have found that Python supports a wide range of coding styles. In particular, it is possible to code almost literal line-by-line translations of Matlab programs by making heavy use of Numpy’s matrices and Pylab’s plotting facilities. A key feature is the ability to use interpreted Python from a command line, enabling Matlab-like exploitation of data, testing of functions, and calculator-style calculations. There is typically a little more keyboard typing than when using Matlab. Throughout we have drawn explicit parallels between using Python and Matlab, as Matlab (or Octave) is often the preferred choice for rapid model development and analysis.

Python’s class system allows Java-like object-oriented construction of dendrites, neurons and populations. Stylistically its use is similar to Java, or C++ with passing by reference. We have found it a natural but relatively slow-execution way to model neural systems.

The physical and neural simulation, with OpenGL interface, runs at comparable speed to commercial robotics simulators such

as Webots (Cyberbotics, Lausanne). We have found development time to be much improved over C++, and comparable with Matlab. However Python gives more versatility than Matlab, allowing easy integration with many open-source libraries and the underlying operating system. Our development has used Emacs with its Python mode. In particular, this integrates with the Python debugger, pdb, to allow visual stepping through code and command-line interaction as in Matlab. This type of interaction can be especially important in neural and AI programs, whose states and interactions can become very complex in unpredicted ways.

LARGE-SCALE INTEGRATION WITH BRAHMS

All of the components discussed above (basal ganglia, hippocampus, 3D simulator) were implemented as stateful functions in Python. Thus, integrating them into a computational system was straightforward, by writing a simple Python “main” function that called these objects in turn to progress them through time. Such an approach to integration is effective, so long as there is no requirement for integration across more complex boundaries. One example of a more complex boundary is cross-language: integrating between functions written in Python, C, Java, or Matlab, for instance, is not generally straightforward. Whilst Python might be a suitable language for large portions of a development, bottleneck computations may benefit from being recoded in a lower-level language such as C. Besides, contributing authors may not all share competence and/or enthusiasm for Python development.

Other obstacles to integration include different component authors, particularly in different groups. This can be problematic since different authors tend to design different interfaces for their components and, in the world of research, rarely have time to properly document these interfaces. Integrating through time – that is, using code written some years ago with code written today – can throw up the same problems as integrating across authors, particularly if documentation is lacking. Cross-platform integration is sometimes necessary, particularly as emphasis shifts to high-performance or embedded computing, and this is far from trivial.

As such multi-module eclectic models become prevalent, and with growing interest in widely varying use cases (high-performance, desktop, embedded), a general solution to the integration problem is urgently required. One such solution is the BRAHMS Modular Execution Framework (brahms.sourceforge.net; Mitchinson et al., 2008). BRAHMS consists of a supervisor, which is analogous to the simple Python “main” function mentioned above, a fixed supervisor interface against which software components can be developed (currently available in C, C++, Matlab and Python), and a user-extensible set of data types for passing data between software components (forming the inter-process interface). Components need not agree between themselves on implementation: they need only conform to these two interfaces provided and made public by the framework. A BRAHMS system, constructed from processes authored as described below, can be parallelised across computer cores sharing memory or connected by an MPI layer or LAN; alternatively, it can be run on an embedded system, since BRAHMS is lightweight. Here we describe the BRAHMS Python language binding.

A BRAHMS PROCESS IN Python

The current BRAHMS Python binding (called “1262”) requires that the process be implemented as a function; this function is rendered stateful by passing in and out a reference to a dictionary object, called *persist*. The function is a handler for framework events, so its body consists of a switch block on the event type. The 1262 template provided with BRAHMS handles four events.

The first, (EVENT_MODULE_INIT), returns information about the process to the framework, and is already implemented completely in the template. The developer can update the author information as appropriate and familiarise themselves with the two possible process flags (discussed below). The second, (EVENT_STATE_SET), passes the component its state, which is obtained by the framework from the system document. This “state” typically consists only of process parameters for initialisation. The third event, (EVENT_INIT_CONNECT), requires that the process validate its inputs and create its outputs (discussed below). The fourth, (EVENT_RUN_SERVICE), requires that the process service its inputs and outputs (read input data, write output data) at some time, t . This implies that the process must complete its computations at least up to time t (a process is free to progress its state beyond t for any reason). This last event (discussed below) is received multiple times during execution and is, effectively, the process *step* function.

Connectivity

In general, a system to be computed may include any number of processes, each of which has each of its outputs dependent on some subset of its inputs. A valid (fully specified) system may have arbitrary (including recursive) output structure dependencies. Since processes are responsible for instantiating their own outputs this requires, in general, multiple calls from the framework to each process in the system to request that it create outputs. The BRAHMS supervisor takes care of making these calls (by sending event EVENT_INIT_CONNECT), guarantees that more inputs will be available on each subsequent call (with zero to N available on the first call, and exactly N available on the last), and requires that each process follow a simple algorithm on receiving each call. The algorithm is: (a) observe (and validate, if necessary) the structure of any newly presented inputs; (b) create *as many outputs as possible*. This algorithm will successfully instantiate any valid system. When required dependencies are not met, the framework will raise a “deadlock” error.

EXAMPLE

We have constructed and executed successfully a second version of the integrated basal ganglia, hippocampus and physical world simulation in which these three components are implemented as separate BRAHMS modules. Such conversion is straightforward, with each module being pasted into the template and modified such that it expresses the interface described above. Wrapper code linking a Hippocampus Python object to BRAHMS is given in the Appendix. We tested the overhead introduced by the BRAHMS framework by running a BRAHMS-wrapped version of the Numpy basal ganglia model used in the previous speed comparisons. **Table 1** shows that the overhead of using BRAHMS is very small; yet using it will now allow extensions to the basic integrated model in any (currently supported) language or level of modelling detail.

CONCLUSIONS

For large-scale integration and testing of neural models, Python can achieve an excellent balance between development time and computational run-time. The flexibility offered by its modules allows programmers to adopt the style most comfortable to them, without a strong penalty in computation time. We have shown here how all these aspects have contributed to the construction of both an integrated basal ganglia-hippocampal formation model for spatial navigation and its embodiment. Moreover, Python either forms the basis for (PyNN; neuralensemble.org/trac/PyNN), or is compatible with (BRAHMS; Mitchinson et al., 2008), platforms that address larger-scale integration across modelling levels and hardware. Thus, Python is a crucial part of the neuroinformaticstoolbox: flexible, usable, readable, and scalable.

APPENDIX

The following shows the code used to link the Hippocampus model to the BRAHMS framework. The code implements four BRAHMS events. The persistent state consists of an instance of a pre-trained Hippocampus object, created in EVENT_STATE_SET. Servicing (EVENT_RUN_SERVICE) consists of reading the BRAHMS inputs, passing them in an appropriate format to the Hippocampus object, and passing its output back to BRAHMS. The other events are described in the Section “A BRAHMS Process in Python”.

```
import brahms
from hc import *
def brahms_process(persist, input):
    output = {'info': {}, 'operations': [], 'event':
              {'response': 0}}

    if input['event']['type'] == EVENT_MODULE_INIT:
        #these flags inform BRAHMS that this process
        #needs all inputs to be available before it can
        #initialise,
        #and that the process does not change the sample rate.
        output['info']['flags'] = F_NEEDS_ALL_INPUTS + F_NOT_
            RATE_CHANGER
        output['info']['component'] = (0, 1)
        output['info']['additional'] = "
        output['event']['response'] = C_OK

    elif input['event']['type'] == EVENT_STATE_SET:
        #create an instance of the Python Hippocampus object
        pars = persist['state']
        persist['ptHC'] = loadHippocampus()
        output['event']['response'] = C_OK

    elif input['event']['type'] == EVENT_INIT_CONNECT:
        #check the data types of the BRAHMS inputs
        p = input['iif']['default']['ports']
        if len(p) != 1:
            output['error'] = 'expects one input'
            return (persist, output)
        if p[0]['class'] != 'dev/std/data/numeric':
            output['error'] = 'expects data/numeric INPUT'
            return (persist, output)
        if p[0]['structure'] != 'DOUBLE/REAL/102':
            output['error'] = 'expects real double 2x1 input'
            return (persist, output)
        #create a BRAHMS output
```



```

persist['hOut'] = brahms.operation(
    persist['self'], OPERATION_ADD_PORT,
    ", 'dev/std/data/numeric',
    DOUBLE/REAL/' + str(persist['state']['n_out']),
    out')
output['event']['response'] = C_OK

elif input['event']['type'] == EVENT_RUN_SERVICE:
    ptHC = persist['ptHC'] #retrieve my persistent state
    #retrieve my current inputs from BRAHMS
    ins = input['iif']['default']['ports'][0]['data']
    x = ins[0]
    z = ins[1]
    img = ins[2:102]
    img = array(img.ravel())
    img.shape = (100,1)
    #call to the Python Hippocampus object
    x_hat, z_hat = ptHC.step(x,z,img)
    #create output and send it to BRAHMS
    myOutput = numpy.array([x, z, x_hat, z_hat], numpy.
double)

```

REFERENCES

- Albertin, S. V., Mulder, A. B., Tabuchi, E., Zugaro, M. B., and Wiener, S. I. (2000). Lesions of the medial shell of the nucleus accumbens impair rats in finding larger rewards, but spare reward-seeking behavior. *Behav. Brain Res.* 117, 173–183.
- Alexander, G. E., and Crutcher, M. D. (1990). Functional architecture of basal ganglia circuits: neural substrates of parallel processing. *Trends Neurosci.* 13, 266–272.
- Arleo, A., and Gerstner, W. (2000). Spatial cognition and neuro-mimetic navigation: a model of hippocampal place cell activity. *Biol. Cybern.* 83, 287–299.
- Birdwell, J., Solomon, J., Thajchayapong, M., Taylor, M., Cheely, M., Towal, R., Conradt, J., and Hartmann, M. (2007). Biomechanical models for radial distance determination by the rat vibrissal system. *J. Neurophysiol.* 98, 2439–2455.
- Bolam, J. P., Hanley, J. J., Booth, P. A., and Bevan, M. D. (2000). Synaptic organization of the basal ganglia. *J. Anat.* 196(Pt 4), 527–542.
- 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 Schutter, E. D. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.
- Chavarriaga, R., Strössl, T., Sheynikhovich, D., and Gerstner, W. (2005). A computational model of parallel navigation systems in rodents. *Neuroinformatics* 3, 223–241.
- Chevalier, G., and Deniau, J. M. (1990). Disinhibition as a basic process in the expression of striatal function. *Trends Neurosci.* 13, 277–280.
- Devan, B. D., Goad, E. H., and Petri, H. L. (1996). Dissociation of hippocampal and striatal contributions to spatial navigation in the water maze. *Neurobiol. Learn Mem.* 66, 305–323.
- Fahmy, T. (2006). Pivy – Embedding a Dynamic Scripting Language into a Scene Graph Library. Master's Thesis, Vienna, Vienna University of Technology.
- Gerfen, C., and Wilson, C. (1996). The basal ganglia. In *Handbook of Chemical Neuroanatomy*, Vol 12, Integrated Systems of the CNS, Part III, L. Swanson, A. Bjorklund, and T. Hokfelt, eds (Amsterdam, Elsevier), pp. 371–468.
- 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. H., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philos. Trans. R Soc. Lond., B, Biol. Sci.* 356, 1209–1228.
- Gorny, J. H., Gorny, B., Wallace, D. G., and Whishaw, I. Q. (2002). Fimbria-fornix lesions disrupt the dead reckoning (homing) component of exploratory behavior in mice. *Learn Mem.* 9, 387–394.
- Groenewegen, H. J., Mulder, A. B., Beijer, A. V. J., Wright, C. I., Lopes Da Silva, F. H., and Pennartz, C. M. A. (1999). Hippocampal and amygdaloid interactions in the nucleus accumbens. *Psychobiology* 27, 149–164.
- Gurney, K., Prescott, T. J., and Redgrave, P. (2001a). A computational model of action selection in the basal ganglia I: a new functional anatomy. *Biol. Cybern.* 85, 401–410.
- Gurney, K., Prescott, T. J., and Redgrave, P. (2001b). A computational model of action selection in the basal ganglia II: analysis and simulation of behaviour. *Biol. Cybern.* 85, 411–423.
- Gurney, K. N., Humphries, M., Wood, R., Prescott, T. J., and Redgrave, P. (2004). Testing computational hypotheses of brain systems function using high level models: a case study with the basal ganglia. *Network* 15, 263–290.
- Hafting, T., Fyhn, M., Molden, S., Moser, M.-B., and Moser, E. I. (2005). Microstructure of a spatial map in the entorhinal cortex. *Nature* 436, 801–806.
- Hasselmo, M., Schnell, E., and Barkai, E. (1995). Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region ca3. *J. Neurosci.* 15, 5249–5262.
- Hasselmo, M., Wyble, B., and Wallenstein, G. V. (1996). Encoding and retrieval of episodic memories: role of cholinergic and gabaergic modulation in the hippocampus. *Hippocampus* 6, 693–708.
- Hikosaka, O., Takikawa, Y., and Kawagoe, R. (2000). Role of the basal ganglia in the control of purposive saccadic eye movements. *Physiol. Rev.* 80, 953–978.
- Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci. U.S.A.* 81, 3088–3092.
- Humphries, M. D., Gurney, K., and Prescott, T. J. (2005). Is there an integrative center in the vertebrate brainstem? A robotic evaluation of a model of the reticular formation viewed as an action selection device. *Adapt. Behav.* 13, 97–113.
- Humphries, M. D., and Gurney, K. N. (2002). The role of intra-thalamic and thalamocortical circuits in action selection. *Network* 13, 131–156.
- Humphries, M. D., Stewart, R. D., and Gurney, K. N. (2006). A physiologically plausible model of action selection and oscillatory activity in the basal ganglia. *J. Neurosci.* 26, 12921–12942.
- Khamassi, M. (2007). Complementary Roles of the Rat Prefrontal Cortex and Striatum in Reward-Based Learning and Shifting Navigation Strategies. Ph.D. Thesis, Paris, University Paris 6.
- Lorincz, A. (1998). Forming independent components via temporal locking of reconstruction architectures: a functional model of the hippocampus. *Biol. Cybern.* 79, 263–275.
- Marr, D. (1971). Simple memory: a theory for archicortex. *Philos. Trans. R. Soc. Lond., B, Bio. Sci.* 262, 23–81.
- McNaughton, B. L., Battaglia, F. P., Jensen, O., Moser, E. I., and Moser, M.-B. (2006). Path integration and the neural basis of the 'cognitive map'. *Nat. Rev. Neurosci.* 7, 663–678.
- Michel, O. (2004). Webots(tm): professional mobile robot simulation. *Int. J. Adv. Robotic Syst.* 1, 39–42.
- Mink, J. W., and Thach, W. T. (1993). Basal ganglia intrinsic circuits and their role in behavior. *Curr. Opin. Neurobiol.* 3, 950–957.
- Mitchinson, B., Chan, T., Humphries, M., Chambers, J., Fox, C., and Prescott, T. (2008). BRAHMS: Novel middleware for integrated systems computation. Proceedings of the IEEE International Conference on Intelligent Robots and Systems. Nice, France.

ACKNOWLEDGEMENTS

This work was supported by the European Union Framework 6 IST project 027819 (ICEA project: www.iceaproject.eu) and the European Union Framework 7 ICT project 215910 (BIOTACT project: www.biotact.org).

SUPPLEMENTARY MATERIAL

Videos and source code from the simulation and speed comparisons are presented in the Supplementary Material. The Supplemental Material for this article can be found online at <http://www.frontiersin.org/neuroinformatics/paper/10.3389/neuro.11.006.2009>

- Mulder, A. B., Tabuchi, E., and Wiener, S. I. (2004). Neurons in hippocampal afferent zones of rat striatum parse routes into multi-pace segments during maze navigation. *Eur. J. Neurosci.* 19, 1923–1932.
- O'Keefe, J., and Conway, D. H. (1978). Hippocampal place units in the freely moving rat: why they fire where they fire. *Exp. Brain Res.* 31, 573–590.
- Prescott, T. J., Montes Gonzalez, F. M., Gurney, K., Humphries, M. D., and Redgrave, P. (2006). A robot model of the basal ganglia: behavior and intrinsic processing. *Neural Netw.* 19, 31–61.
- Quirk, G. J., Muller, R. U., and Kubie, J. L. (1990). The firing of hippocampal place cells in the dark depends on the rat's recent experience. *J. Neurosci.* 10, 2008–2017.
- Redgrave, P., Prescott, T. J., and Gurney, K. (1999). The basal ganglia: a vertebrate solution to the selection problem? *Neuroscience* 89, 1009–1023.
- Redish, A. D., and Touretzky, D. S. (1997). Cognitive maps beyond the hippocampus. *Hippocampus* 7, 15–35.
- Ritt, J., Andermann, M., Skowronski-Lutz, E., and Moore, C. (2006). Characterization of Vibrissa Motion During Volitional Active Touch. Atlanta, Barrels XIX.
- Rolls, E. (1995). A model of the operation of the hippocampus and cortex in memory. *Int. J. Neural Syst.* 6, 51–71.
- Rolls, E., Stringer, S., and Elliot, T. (2006). Entorhinal cortex grid cells can map to hippocampal place cells by competitive learning. *Network* 17, 447–465.
- Rossier, J., Kaminsky, Y., Schenk, F., and Bures, J. (2000). The place preference task: a new tool for studying the relation between behavior and place cell activity in rats. *Behav. Neurosci.* 114, 273–284.
- Sargolini, F., Florian, C., Oliverio, A., Mele, A., and Roullet, P. (2003). Differential involvement of NMDA and AMPA receptors within the nucleus accumbens in consolidation of information necessary for place navigation and guidance strategy of mice. *Learn Mem.* 10, 285–292.
- Smith-Roe, S. L., Sadeghian, K., and Kelley, A. E. (1999). Spatial learning and performance in the radial arm maze is impaired after n-methyl-D-aspartate (NMDA) receptor blockade in striatal subregions. *Behav. Neurosci.* 113, 703–717.
- Surmeier, D. J., Ding, J., Day, M., Wang, Z., and Shen, W. (2007). D1 and D2 dopamine-receptor modulation of striatal glutamatergic signaling in striatal medium spiny neurons. *Trends Neurosci.* 30, 228–235.
- Tabuchi, E., Mulder, A. B., and Wiener, S. I. (2003). Reward value invariant place responses and reward site associated activity in hippocampal neurons of behaving rats. *Hippocampus* 13, 117–132.
- Tabuchi, E. T., Mulder, A. B., and Wiener, S. I. (2000). Position and behavioral modulation of synchronization of hippocampal and accumbens neuronal discharges in freely moving rats. *Hippocampus* 10, 717–728.
- Treaves, A., and Rolls, E. (1994). Computational analysis of the role of the hippocampus in memory. *Hippocampus* 4, 374–391.
- Ueno, N., and Kaneko, M. (1994). Dynamic Active Antenna—A Principle of Dynamic Sensing. IEEE ICRA, San Diego, CA, USA, pp. 1784–1790.
- Ujfalussy, B., Eros, P., Somogyvari, Z., and Kiss, T. (2008). Episodes in space: a modelling study of hippocampal place representation. In *From Animals to Animats 10*, Vol. 5040 of LNAI, M. Asada, J. Hallam, J.-A. Meyer, and J. Tani, eds (Berlin, Springer-Verlag), pp. 123–136.
- van Groen, T., and Wyss, J. M. (1990). Extrinsic projections from area CA1 of the rat hippocampus: olfactory, cortical, subcortical, and bilateral hippocampal formation projections. *J. Comp. Neurol.* 302, 515–528.
- Whishaw, I. Q. (1998). Place learning in hippocampal rats and the path integration hypothesis. *Neurosci. Biobehav. Rev.* 22, 209–220.
- Whishaw, I. Q., Cassel, J. C., and Jarrad, L. E. (1995). Rats with fimbria-fornix lesions display a place response in a swimming pool: a dissociation between getting there and knowing where. *J. Neurosci.* 15, 5779–5788.
- Wiener, S. I. (1996). Spatial, behavioral and sensory correlates of hippocampal CA1 complex spike cell activity: implications for information processing functions. *Prog. Neurobiol.* 49, 335–361.

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

Received: 10 September 2008; paper pending published: 21 October 2008; accepted: 20 February 2009; published online: 09 March 2009.

Citation: Fox C, Humphries M, Mitchinson B, Kiss T, Somogyvari Z and Prescott T (2009) Technical integration of hippocampus, basal ganglia and physical models for spatial navigation. *Front. Neuroinform.* (2009) 3:6. doi: 10.3389/neuro.11.006.2009

Copyright © 2009 Fox, Humphries, Mitchinson, Kiss, Somogyvari and Prescott. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Python for information theoretic analysis of neural data

Robin A. A. Ince^{1*}, Rasmus S. Petersen¹, Daniel C. Swan² and Stefano Panzeri^{1,3*}

¹ Faculty of Life Sciences, University of Manchester, Manchester, UK

² Bioinformatics Support Unit, Institute of Cell and Molecular Biosciences, Newcastle University, Newcastle upon Tyne, UK

³ Robotics, Brain and Cognitive Sciences Department, Italian Institute of Technology, Genoa, Italy

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Oswaldo A. Rosso, The University of
Newcastle, Australia

Pietro Berkes, Brandeis University, USA

John M. Beggs, Indiana University, USA

*Correspondence:

Robin A. A. Ince, Faculty of Life
Sciences, 3.614 Stopford Building,
Oxford Road, Manchester,
M13 9PT, UK.

e-mail: robin.ince@postgrad.
manchester.ac.uk

Stefano Panzeri, Robotics, Brain and
Cognitive Sciences Department, Italian
Institute of Technology, Via Morego, 30,
16163 Genoa, Italy.

e-mail: stefano.panzeri@iit.it

Information theory, the mathematical theory of communication in the presence of noise, is playing an increasingly important role in modern quantitative neuroscience. It makes it possible to treat neural systems as stochastic communication channels and gain valuable, quantitative insights into their sensory coding function. These techniques provide results on how neurons encode stimuli in a way which is independent of any specific assumptions on which part of the neuronal response is signal and which is noise, and they can be usefully applied even to highly non-linear systems where traditional techniques fail. In this article, we describe our work and experiences using Python for information theoretic analysis. We outline some of the algorithmic, statistical and numerical challenges in the computation of information theoretic quantities from neural data. In particular, we consider the problems arising from limited sampling bias and from calculation of maximum entropy distributions in the presence of constraints representing the effects of different orders of interaction in the system. We explain how and why using Python has allowed us to significantly improve the speed and domain of applicability of the information theoretic algorithms, allowing analysis of data sets characterized by larger numbers of variables. We also discuss how our use of Python is facilitating integration with collaborative databases and centralised computational resources.

Keywords: Python, information theory, neural coding, entropy, maximum entropy, bias, e-science

INTRODUCTION

Information theory (Cover and Thomas, 2006; Shannon, 1948), the mathematical theory of communication in the presence of noise, is playing an increasingly important role in modern quantitative neuroscience, because it makes it possible to treat neural systems as stochastic communication channels and gain valuable, quantitative insights into their sensory coding function (Borst and Theunissen, 1999; Rieke et al., 1999; Victor, 2006). Information theory provides a set of fundamental mathematical quantities, such as entropy and mutual information, that quantify with meaningful numbers the reduction of uncertainty about stimuli gained from neural responses, without the need to make any specific assumption of what is signal and what is noise in the neuronal response.

Most laboratories (including ours) have so far implemented information theoretic analyses using MATLAB^{®1}. MATLAB is a numerical computing environment and programming language which is used by most neurophysiological laboratories to store, preprocess and plot experimental data. In our view, the reason for the choice of MATLAB for the implementation of such routines is that it allows interactive and rapid development of algorithms, though at the cost of some performance overhead. Traditionally, information calculations have not been demanding in terms of memory usage or CPU time because the information calculations were restricted to relatively small neural populations as a consequence of the limited sampling bias problem. Therefore, it has been convenient to perform the analysis with the tools used to obtain, preprocess and store the data. However, over the last few years, the

CPU and memory requirements of information calculations for neural data has significantly increased. This is due to a number of reasons. First, the improvement of the techniques to correct for the sampling bias problem (Panzeri et al., 2007) has allowed the information theoretic analysis of larger populations. Second, some of these bias corrections techniques are computationally intensive. Third, in the context of understanding whether the correlation structure of neural activity can be described by simple low order models, it has become important to compute distributions with maximum entropy in the presence of various sets of constraints (Schneidman et al., 2006; Shlens et al., 2006; Tang et al., 2008). These calculations are particularly demanding in terms of processor and memory resources. Fourth, while most information analysis has been applied to spike trains, in the context of the development of brain machine interfaces it has become important to evaluate the information content of other types of brain signals, such as local field potentials (LFPs) or Electroencephalograms (EEGs) which are analog in nature and must be represented at each time step (Belitski et al., 2008; Montemurro et al., 2008; Rubino et al., 2006; Waldert et al., 2008). The manipulation of these signals stretches computational requirements much more than using spikes, which due to their sparse binary nature can be represented compactly, for example by storing only the spike arrival times.

The increased demand on the information theoretic routines raises the question of whether it may be advantageous for the scientific community to implement information theoretic algorithms for the analysis of neural data using platforms other than MATLAB. In the continuing development of these methods, we have recently started using Python, together with the numerical libraries NumPy

¹The Mathworks, Inc, Natick, MA. <http://www.mathworks.com/>

and SciPy. We have found several key advantages to this change that make it more suitable for the analysis of the datasets we are currently studying and for future challenges such as implementing these methods into computational grids and clusters.

In this article, we first briefly present the principles of information theory and its importance to neuroscience. We then review some features of Python that are particularly useful for information theoretic analysis and consider in detail the implementation of the mathematical algorithms that are crucial for obtaining accurate and unbiased estimates of information from neural data. We also detail a method to compute the entropy of neural data given a number of plausible constraints, and we put particular emphasis on the specific advantages of Python in addressing these algorithmic challenges. We finally apply the methodology to real data recorded from the rat somatosensory cortex, and discuss the potential implications of wider use of Python in information theoretic analysis of the neural code.

INFORMATION THEORY FOR ANALYSIS OF NEURAL DATA

Information theory is a “mathematical theory of communication” developed in the 1940’s by Claude Shannon at Bell Labs (Cover and Thomas, 2006; Shannon, 1948). It formalises, in a mathematically rigorous way, a measure of “information” in a system with applications to coding and transmission of that information. While it was originally developed for analysis of artificial systems, such as transmission of signals along a telegraph wire, the generality of the formulation means it can be usefully applied to a wide range of problems.

Consider an experiment in which an animal is presented with a stimulus s selected with probability $P(s)$ from a stimulus set \mathbf{S} consisting of S elements, and the consequent response (either of a single neuron or an ensemble of neurons) is recorded and quantified in a certain post-stimulus time window. The aim of information theoretic analysis is to gain insight into how the neurons represent the stimuli. In most applications this is done by examining the information content of different candidate neural codes. To carry out such an analysis, the first step is to choose the neural code. In practice this means choosing a way to quantify the neuronal response that reflects our assumption of what is most salient in it. For example, if we think that only spike counts (not the precise temporal pattern of spikes) are important, we choose a spike-count code: we define a post-stimulus response interval and count the number of spikes it contains on each repetition (trial) of a stimulus. In most cases, the neural response is quantified as a discrete, multi-dimensional array $\mathbf{r} = \{r_1, \dots, r_L\}$ of dimension L . For example, to quantify the spike count response of a population of L cells, r_i would be the number of spikes emitted by cell i on a given trial in the response window. Alternatively, to quantify the spike timing response of a single neuron, the response window is divided into L bins of width Δt , so that r_i is the number of spikes fired in the i -th time bin (Strong et al., 1998). Here Δt is the assumed time precision of the code and can be varied parametrically to characterize the temporal precision of the neural code. We denote by \mathbf{R} the set of possible values taken by the response array.

Having quantified the response, the second step is to compute how much information can be extracted from the chosen response quantification. This allows an assessment of how good the

candidate neural code is. The more the response of a neuron varies across a set of stimuli, the greater its ability to transmit information about those stimuli (de Ruyter van Steveninck et al., 1997). The first step in measuring information is thus to measure the response variability. The most general way to do this is through the concept of *Shannon entropy*, referred to hereafter as *entropy*, which is a measure of the uncertainty associated with a random variable. Intuitively one can posit some desirable properties of any uncertainty measure. It should be *continuous*; that is small changes in the underlying probabilities should result in small changes in the uncertainty. It should be *symmetric*; that is the measure should not depend on the labelling or ordering of the variables and outcomes. The measure should take its maximum value when all outcomes are equally likely and for systems with uniform probabilities, the measure should increase with the number of outcomes. Finally, the measure should be *additive*; that is it should be independent of how the system is grouped or divided into parts. It can be shown (Cover and Thomas, 2006) that any measure of uncertainty about the neural responses satisfying these properties has the form

$$H(\mathbf{R}) = - \sum_{\mathbf{r} \in \mathbf{R}} P(\mathbf{r}) \log_2 P(\mathbf{r}) \quad (1)$$

where $P(\mathbf{r})$ is the probability of observing response \mathbf{r} across all trials to all stimuli. The response entropy quantifies how neuronal responses vary with the stimulus and thus sets the capacity of the spike train to convey information. In Eqs 1 and 2 the summation over \mathbf{r} is over all possible neuronal responses. However, neurons are typically noisy; their responses to repetitions of an identical stimulus differ from trial to trial. $H(\mathbf{R})$ reflects both variation of responses to different stimuli and variation due to trial-to-trial noise. Thus $H(\mathbf{R})$ is not a pure measure of the stimulus information actually transmitted by the neuron. We can quantify the variability specifically due to noise, by measuring the so-called *noise entropy*, which is the entropy conditional on stimulus presentation:

$$H(\mathbf{R}|\mathbf{S}) = - \sum_{s \in \mathbf{S}} P(s) \sum_{\mathbf{r} \in \mathbf{R}} P(\mathbf{r}|s) \log_2 P(\mathbf{r}|s) \quad (2)$$

The summation over s is over all possible stimuli. $P(\mathbf{r}|s)$ is the probability of observing a particular response \mathbf{r} given that stimulus s is presented. Experimentally, $P(\mathbf{r}|s)$ is determined by repeating each stimulus on many trials, while recording the neuronal responses. The probability $P(s)$ is usually chosen by the experimenter. The noise entropy quantifies the irreproducibility of the neuronal responses at fixed stimulus. The noisier is a neuron, the greater is $H(\mathbf{R}|\mathbf{S})$. The information that the neuronal response transmits about the stimulus is the difference between the response entropy and the noise entropy. This is known as the mutual information $I(\mathbf{S}; \mathbf{R})$ between stimuli and responses (in the following abbreviated to information).

$$I(\mathbf{S}; \mathbf{R}) = H(\mathbf{R}) - H(\mathbf{R}|\mathbf{S}) \quad (3)$$

Mutual information quantifies how much of the information capacity provided by stimulus-evoked differences in neural activity is robust to the presence of trial-by-trial response variability (de Ruyter van Steveninck et al., 1997). Alternatively, it quantifies

the reduction of uncertainty about the stimulus that can be gained from observation of a single trial of the neural response.

The mutual information has a number of important qualities that make it well suited to characterizing how a response is modulated by the stimulus (Borst and Theunissen, 1999; Fuhrmann Alpert et al., 2007; Panzeri et al., 2008; Rieke et al., 1999). First, as outlined above, it quantifies the stimulus discriminability achieved from a single observation of the response, rather than from averaging responses over many observations. Second, $I(\mathbf{S}; \mathbf{R})$ is the most general measure of correlation between the stimuli and the neural responses, because it automatically takes into account contributions of correlations at all orders. Third, computing information does not require specifying a stimulus–response model; it only requires computing the response probabilities in response to each stimulus condition. Therefore, the calculation of information does not require spelling out which stimulus features (e.g., contrast, orientation, etc.) are encoded. Fourth, $I(\mathbf{S}; \mathbf{R})$ takes into account the full stimulus–response probabilities, which include all possible effects of stimulus-induced responses and noise. Thus, it does not require the signal to be modeled as a set of response functions plus noise and is applicable even to situations when such decompositions are difficult or dubious. The last three points show that information theory can, in principle, be applied to any type of neural signal, including responses such as LFPs or spikes that are clearly nonlinear and difficult to model by a set of standard functions. Fifth, it is possible to analyze and combine the information given by different measures of neural activity e.g. spike trains and LFPs. These two signals have a very different nature and signal to noise ratios. Therefore, a certain increase of the peak height of an LFP cannot be compared to a certain change in the spike train to understand how well LFPs or spikes encode stimuli. In contrast, with information theory the LFPs and spikes can be directly compared because information theory projects both signals onto a common scale that is meaningful in terms of stimulus knowledge.

Information theoretic techniques have been successfully used to address a number of questions about sensory coding. For example, they have been used to address the question of whether neurons convey information by millisecond precision spike timing or simply by the total number of emitted spikes (the spike count). The application of information theory to spike train analysis has showed that the ms-precise timing of spikes provides important information that cannot be extracted from spike counts (Panzeri et al., 2001; Victor, 1999, 2006). Information theory has also been used to characterize the functional role of correlations in population activity, by investigating in which conditions correlations play a quantitatively important role in transmitting information about the stimulus (Averbeck et al., 2006; Dan et al., 1998; Hatsopoulos et al., 1998; Latham and Nirenberg, 2005; Panzeri, 1999; Petersen et al., 2001; Pola et al., 2003) or in constraining the dynamic range of network responses (Schneidman et al., 2006). Information theory has also been used to characterize the amount of interactions between neural populations (Honey et al., 2007).

WHY PYTHON?

For many years, the de facto standard for many groups working in the area of neurophysiological data analysis has been MATLAB®.

However, the Python programming language (van Rossum, 1995) combined with the numerical and scientific libraries NumPy and SciPy (Jones et al., 2001) provide a compelling alternative for scientific programming. Python is a modern, fully object-oriented programming language that is powerful, flexible and easy to learn. The NumPy library provides a multi-dimensional array object and associated vectorised operations, and SciPy enhances this with a range of scientific functions using the NumPy array object. The syntax is familiar to anyone coming from a background with MATLAB or another C derivative language and there are a comprehensive set of tools for plotting and interactive use (IPython and Matplotlib). Assignments are by reference rather than by copying, which allows finer grained control of memory usage, and there are several ways to rapidly extend the system with external code written in FORTRAN and C. The flexibility and good design of the Python language make large projects much more manageable than with MATLAB, where each function must reside in a separate file and refactoring to reduce code repetition grows increasingly difficult with project size. Python is a well developed language, with libraries available for almost any conceivable task, such as GUI development, network communication, support for different file formats, etc. It is possible to read and write MATLAB binary files, and even call MATLAB commands from within the Python environment, which allows for a smooth transition and means that time invested in an existing MATLAB code base is not wasted. Finally, the Python tool set is *open source*², rather than a proprietary product, which has several obvious advantages for scientific work. Its free availability allows better reproducibility of the results, since all interested parties are free to run the software without an expensive license. It is also inherently future-proof, since it will always be possible to obtain and use the version for which the code was written, whereas a commercial product may be withdrawn at some point in the future.

THE LIMITED SAMPLING BIAS PROBLEM

A major difficulty when applying techniques involving information theoretic quantities to experimental systems, is that they require measurement of the full probability distributions of the variables involved. If we had an infinite amount of data, we could measure the true stimulus-response probabilities precisely. However, any real experiment only yields a finite number of trials from which these probabilities must be estimated. The estimated probabilities are subject to statistical error and necessarily fluctuate around their true values. The significance of these finite sampling fluctuations is that they lead to both statistical error (variance) and systematic error (called *limited sampling bias*) in estimates of entropies and information. This bias is the difference between the expected value of the quantity considered, computed from probability distributions estimated with N trials or samples, and its value computed from the true probability distribution. The bias constitutes a significant practical problem, because its magnitude is often of the order of the information values to be evaluated, and because it

²“Open source is a development method for software that harnesses the power of distributed peer review and transparency of process. The promise of open source is better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in.” <http://www.opensource.org/>

cannot be alleviated simply by averaging over many neurons with similar characteristics.

ORIGINS OF THE BIAS

The most direct way to compute information and entropies is to estimate the response probabilities as the histogram of the experimental frequency of each response across the available trials. Plugging in these empirical probability estimates into Eqs 1–3 results in a direct estimate that we refer to as the “plug-in” method.

In general, both the full output entropy $H(\mathbf{R})$ and the noise entropy $H(\mathbf{R}|\mathbf{S})$ are biased downwards. That is, the estimated value is less than the true value, and the estimated value increases with the number of trials used, asymptotically approaching the true value. Intuitively, this is because finite sampling means it is less likely that the full range of responses will be included and so the measured responses seem less variable than they really are. In addition, estimates of $H(\mathbf{R}|\mathbf{S})$ are significantly more biased than those of $H(\mathbf{R})$, since the latter depends on $P(\mathbf{r})$ which is calculated with data gathered across all stimuli and is better sampled than the conditional distributions, which are each sampled with data from a single stimulus only. The bias in the mutual information is then the difference between the bias of $H(\mathbf{R})$ and that of $H(\mathbf{R}|\mathbf{S})$. This results in an upward bias in the information, since the magnitude of the bias of $H(\mathbf{R}|\mathbf{S})$ is greater, and its sign is reversed in Eq. 3. Again, this makes sense intuitively, since the finite sampling can introduce spurious stimulus-dependent differences in the response probabilities, which make the stimuli seem more discernible and hence the neuron more informative than it really is.

BIAS CORRECTION METHODS

Fortunately a number of techniques have been developed to address the issue of bias, and allow much more accurate estimates of information theoretic quantities than the “plug-in” method described above. Panzeri et al. (2007) provide a review of such methods, a selection of which are briefly outlined here. For other methods and approaches please see Panzeri et al. (2007) and Victor (2006).

Panzeri–Treves (PT)

In the so-called *asymptotic sampling regime*, when the number of trials is large enough that every possible response occurs many times, an analytical approximation for the bias (i.e. the difference between the true value and the plug-in estimate) of entropies and information can be obtained (Miller, 1955; Panzeri and Treves, 1996).

$$\begin{aligned} \text{BIAS}[H(\mathbf{R})] &= \frac{-1}{2N \ln 2} [\bar{R} - 1] \\ \text{BIAS}[H(\mathbf{R}|\mathbf{S})] &= \frac{-1}{2N \ln 2} \sum_s [\bar{R}_s - 1] \\ \text{BIAS}[I(\mathbf{S}; \mathbf{R})] &= \frac{-1}{2N \ln 2} \left\{ \sum_s [\bar{R}_s - 1] - [\bar{R} - 1] \right\} \end{aligned} \quad (4)$$

The value of the bias computed from the above expressions is then subtracted from the plug-in estimate to obtain the corrected values. This requires an estimate of the number of relevant responses \bar{R}_s . The simplest approach is to approximate \bar{R}_s by the

count of responses that are observed at least once – this is the “naive” count. However due to finite sampling this will be an underestimate of the true value. A Bayesian procedure (Panzeri and Treves, 1996) can be used to obtain a more accurate value.

Quadratic Extrapolation (QE)

In the asymptotic sampling regime, the bias of entropies and information can be approximated as second order expansions in $1/N$, where N is the number of trials (Strong et al., 1998; Treves and Panzeri, 1995). For example, for the information:

$$I_{\text{plugin}}(\mathbf{S}; \mathbf{R}) = I_{\text{true}}(\mathbf{S}; \mathbf{R}) + \frac{a}{N} + \frac{b}{N^2} \quad (5)$$

This property can be exploited by calculating the estimates with subsets of the original data, with $N/2$ and $N/4$ trials and fitting the resulting values to the polynomial expression above. This allows an estimate of the parameters a and b and hence $I_{\text{true}}(\mathbf{S}; \mathbf{R})$. To use all available data, estimates of two subsets of size $N/2$ and four subsets of size $N/4$ are averaged to obtain the values for the extrapolation. Together with the full length data calculation, this requires seven different evaluations of the quantity being estimated.

Nemenman–Shafee–Bialek (NSB)

The NSB method (Nemenman et al., 2002, 2004) utilises a Bayesian inference approach and does not rely on the assumption of the asymptotic sampling regime. It is based on the principle that when estimating a quantity, the least bias will be achieved when assuming an a priori uniform distribution over the quantity. This method is more challenging to implement than the other methods, involving a large amount of function inversion and numerical integration. However, it often gives a significant improvement in the accuracy of the bias correction (Montemurro et al., 2007b; Nemenman et al., 2002, 2004).

Shuffled Information Estimator (I_{sh})

Recently, an alternative method of estimating the mutual information has been proposed (Montemurro et al., 2007b; Panzeri et al., 2007). Unlike the methods above, this is a method for calculating the information only, and is not a general entropy bias correction. However, it can be used with the entropy corrections described above to obtain more accurate results. For this method, two new quantities are defined. $H_{\text{ind}}(\mathbf{R}|\mathbf{S})$ is the noise entropy that would be obtained if each individual component r_i of the response array \mathbf{r} were independent of any other component r_j ($i \neq j$) at fixed stimulus; that is the entropy calculated from the distribution $P_{\text{ind}}(\mathbf{r}|s) = \prod_i P(r_i|s)$. Since this value depends only on the first order marginal values of the response, it has a small bias. $H_{\text{sh}}(\mathbf{R}|\mathbf{S})$ is the entropy that results when stimulus conditional response correlations are removed by “shuffling” the data. That is, for each stimulus s , the individual response components r_i are shuffled independently across trials, to obtain a new set of vector responses \mathbf{r} . Both of these values provide estimates of the entropy of the system if correlations were removed and become equal for an infinite number of trials. However, with finite trials, $H_{\text{ind}}(\mathbf{R}|\mathbf{S})$ shows a small bias, while $H_{\text{sh}}(\mathbf{R}|\mathbf{S})$ shows a much larger bias, which is of the same order of magnitude as that of $H(\mathbf{R}|\mathbf{S})$, but typically slightly more negative. Using these

properties, a so-called shuffled information estimator, I_{sh} , can be computed as

$$I_{sh}(\mathbf{S}; \mathbf{R}) = H(\mathbf{R}) - H_{ind}(\mathbf{R}|\mathbf{S}) + H_{sh}(\mathbf{R}|\mathbf{S}) - H(\mathbf{R}|\mathbf{S}) \quad (6)$$

In the limit of a large number of trials $I_{sh}(\mathbf{S}; \mathbf{R}) = I(\mathbf{S}; \mathbf{R})$ since $H_{sh}(\mathbf{R}|\mathbf{S}) = H_{ind}(\mathbf{R}|\mathbf{S})$. For small numbers of trials, the biases of $H_{sh}(\mathbf{R}|\mathbf{S})$ and $H(\mathbf{R}|\mathbf{S})$ approximately cancel out, leaving the bias of $I_{sh}(\mathbf{S}; \mathbf{R})$ dominated by that of $H(\mathbf{R}) - H_{ind}(\mathbf{R}|\mathbf{S})$ which is much smaller than that of the normal information estimate $I(\mathbf{S}; \mathbf{R})$. Using this shuffling technique, combined with entropy bias correction methods as described above, can reduce the number of trials needed for a reliable estimate by a factor of four (Montemurro et al., 2007b; Panzeri et al., 2007).

James–Stein Shrinkage (“Shrink”) Estimator

Another recently proposed technique to compute entropies from limited samples is the so-called “James–Stein shrinkage” technique (Hausser and Strimmer, 2008), which works by improving the estimate of the underlying probabilities, rather than the entropy specifically. The James–Stein shrinkage technique is based on averaging two models with different properties; a high dimensional model with low bias and high variance and a lower dimensional one with larger bias but smaller variance. The probabilities p_r of each response r are determined by

$$p_r^{Shrink} = \lambda t_r + (1 - \lambda) p_r^{ML} \quad (7)$$

where $\lambda \in [0, 1]$ is the shrinkage intensity, p_r^{ML} is the normal maximum likelihood estimate from frequency counts and t_r is the shrinkage target. The maximum entropy uniform distribution is suggested as a convenient target in Hausser and Strimmer (2008). The shrinkage intensity λ is then given by the following

$$\lambda^* = \frac{1 - \sum_r (p_r^{ML})^2}{(n-1) \sum_r (t_k - p_r^{ML})^2} \quad (8)$$

This is repeated for all the stimulus conditional distributions, and the entropy is calculated from the corrected probability values using the plug-in method.

Comparative performance of different estimators

Figure 1 reports the results of the performance of bias correction procedures on a set of simulated spike trains from eight simulated neurons. Each of these neurons could emit a spike or not with a probability obtained from a Bernoulli process. The spiking probabilities were exactly equal to those measured, in the 10–15 ms post-stimulus interval, from eight neurons in rat somatosensory cortex responding to 13 stimuli consisting of whisker vibrations of different amplitude and frequency (Arabzadeh et al., 2004). The 10–15 ms interval was chosen since it was found to be the interval containing highest information values. Figure 1A shows that (with the exception of the James–Stein shrinkage) all bias correction procedures generally improve the estimate of $I(\mathbf{S}; \mathbf{R})$ with respect to the plug-in estimator, and the NSB correction is especially effective. For the James–Stein shrinkage estimator, a uniform target distribution was used, and this may account for the relatively poor performance of that method outside of the

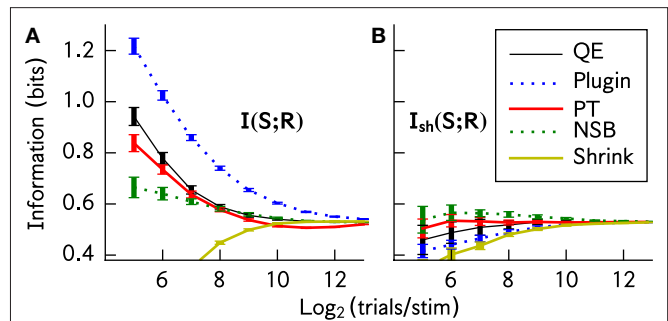


FIGURE 1 | Comparison of the performance of different bias correction methods. The methods were applied to spike trains of eight simulated somatosensory cortical neurons (see text). The information estimates $I(\mathbf{S}; \mathbf{R})$ and $I_{sh}(\mathbf{S}; \mathbf{R})$ are plotted as a function of the available number of trials per stimulus. **(A)** Mean \pm SD/2 (over 50 simulations) of $I(\mathbf{S}; \mathbf{R})$. **(B)** Mean \pm SD/2 (over 50 simulations) of $I_{sh}(\mathbf{S}; \mathbf{R})$. This calculation is very similar to that in Panzeri et al. (2007, Figure 3), which also used realistic simulations of cortical spike trains (the only difference was that for this figure, the simulated population did not contain any correlations). This figure was produced using the Python library for bias corrections described in Section “A Python Library for Information Theoretic Estimates”; and the code to produce it is available at <http://code.google.com/p/pyentropy/>.

asymptotic regime. Figure 1B shows that the bias-corrected estimation of information is much improved by using $I_{sh}(\mathbf{S}; \mathbf{R})$ rather than $I(\mathbf{S}; \mathbf{R})$. The use of $I_{sh}(\mathbf{S}; \mathbf{R})$ makes the residual errors in the estimation of information much smaller and almost independent from the bias correction method used. Taking into account both bias correction performance and computation time, for this simulated system the best method to use is the shuffled information estimator combined with the Panzeri–Treves analytical correction. Using this, an accurate estimate of the information is possible even when the number of samples per stimulus is $\frac{n}{4}$ where R is the dimension of the response space.

While the basic plug-in entropy calculation is a straightforward sum of logarithms, the correction methods described above add significant complexity to the required calculations. In QE, the underlying entropy calculations have to be run many times, for PT the Bayesian estimate of the number of stimulus responses involves additional calculations and NSB involves a complicated procedure of many numerical integrations. For large data sets, with the large probability spaces that can often arise from modern physiological techniques, performance can be an issue as these computational methods become increasingly CPU and memory intensive. Since the performance of bias correction procedures depends on the statistics of data under analysis, in each data analysis task it is also important to test the accuracy of information estimation methods on simulated data with statistical properties similar to the actual experimental data of interest (Panzeri et al., 2007). It is therefore crucial that these methods be implemented as efficiently as possible. An advantage of Python is that one can benefit both from the improved development time due to the simple syntax and interactive environment, as well as a number of well developed methods for optimising the performance critical portions of the code when necessary. There are tools for automatically converting Python to C inline, inserting your own C code within a Python program, writing full C and FORTRAN extension modules or using Cython, which

is a variant of the Python language with a similar syntax but that compiles straight to C code.

A PYTHON LIBRARY FOR INFORMATION THEORETIC ESTIMATES

The study and development of techniques for estimation of information theoretic quantities and associated bias corrections has developed into a field of its own. In order for the results of this work to be useful outside of this small community it must be possible for non-specialists to easily apply these techniques to their data. We have therefore developed a library of tools with the dual purpose of allowing easy application of the most suitable cutting edge bias corrections, while also providing a framework for continued enhancement of existing methods as well as development of new techniques. Although this has been developed for application to investigations of neural coding, the library has been designed to be as general as possible, in the hope that it might also be of use in other areas, and it is publicly available under an open source license³. There are similar packages available in other languages, such as the Rentropy library⁴ and the MATLAB Spike Train Analysis Toolbox⁵, but the authors are not aware of any similar Python package.

At the core of the library are two classes, `DiscreteSystem` and `SortedDiscreteSystem` which sample and store the probability distributions associated with a system and contain methods to compute different entropy quantities. `DiscreteSystem` is the most general and can take arbitrarily ordered input. The class is initialised as `s=DiscreteSystem(X, X_dims, Y, Y_dims)` where `X_dims=(Xn, Xm)` and `Y_dims=(Yn, Ym)` are tuples of values describing the parameters of the X and Y spaces respectively. `Xn` and `Yn` are the number of variables in the space, each of which is quantised to take one of `Xm` or `Ym` possible values, respectively. In total therefore there are Xm^{Xn} possible values in the X space and Ym^{Yn} in the Y space for each trial. X and Y are provided as integer arrays with values in $[0, Xm - 1]$ and $[0, Ym - 1]$ respectively with `Xn`, `Yn` rows representing the constituent variables and a column for each trial. It is important the columns match, that is the value of X in a given column corresponds to the same trial as the value of Y in the same column, but there are no further requirements on the format of the input. `SortedDiscreteSystem` requires the input trials to be grouped in values of the variable Y. This allows much more efficient sampling of the required probability distributions, since the trials for a given Y value can be easily isolated without having to search through the whole data set. This requires the space Y to be a single finite alphabet variable, so it should be decimalised beforehand if necessary. The class is initialised as `s = SortedDiscreteSystem(X, X_dims, Ym, Ny)` where X, X_dims are as above and Ym is the number of possible values for the single variable Y space. Ny is an array containing the number of trials available for each Y value. For example, `Ny[0]` is the number of trials available with `Y = 0`, and the corresponding X values are found at `X[0 : Ny[0]]`. Both of these classes inherit from a base class `BaseSystem` which contains the common entropy and information calculations, reducing code duplication and increasing maintainability.

In neural coding applications such as those described previously, Y would be the stimulus space S, while X would be the response space R. Since the stimuli are usually controlled by the experimenter, the results are often available already sorted by stimulus, allowing use of the more efficient `SortedDiscreteSystem` class. Mutual information is symmetric, $I(X; Y) = I(Y; X)$, so in fact the stimulus and response spaces can be provided in any order, but due to the way the conditional probabilities are sampled it is strongly suggested that the smaller of the two spaces be provided as the Y parameter.

Once initialised as above, entropy quantities can be calculated using the method `s.calculate_entropies(method, sampling, calc)` where `method` is one of `['plugin', 'pt', 'qe', 'nsb']` and selects the bias correction technique to use, `sampling` is one of `['naive', 'beta:x', 'shrink']` which selects the method for estimating the probability distributions and `calc` is a list containing a number of entropies to calculate. The entropies available are `['HX', 'HY', 'HXY', 'SiHXi', 'HiX', 'HiXY', 'HshXY', 'ChiX']`, which in the case where, as described above, the space X corresponds to the response space R and Y to the stimulus space S, denote respectively $H(R)$, $H(S)$, $H(R|S)$, $\sum_{i=1}^{Rn} H(R_i)$, $H_{ind}(R)$, $H_{ind}(R|S)$, $H_{sh}(R|S)$ and $\chi(R)$. $\chi(R)$ is a quantity needed for the information breakdown of (Pola et al., 2003) and is reported in Eq. 25 therein. This function will first decimalise the X and Y spaces, if required (if $n > 1$) which involves converting the length- n base- m words representing the values for each space to a single decimal integer value in $[0, m^n - 1]$. The probabilities required for the requested output entropies are then computed using the sampling method specified. “naive” represents the standard histogram bin counting method which is usually used. The add-constant estimator (Schürmann and Grassberger, 1996) is implemented through the “beta:x” method. The β parameter is provided after the colon in the option, so “beta:0.01” would use the add-constant estimator with $\beta = 0.01$. The “shrink” option selects the James–Stein shrinkage estimator (Hausser and Strimmer, 2008). All the entropy estimates are currently implemented in pure Python, except for the NSB estimator. This is implemented using existing publicly available optimised codes⁶. We have not yet implemented a direct link to the NSB codes, but instead write the data for analysis to a file, for processing by the standalone external program before reading back results from a file. Python’s heritage as a scripting language makes this process of reading and writing formatted files and programmatically calling an external program from the code very easy. The functions `s.I()` and `s.Ish()` can be used to obtain the mutual information estimate and shuffled mutual information estimate respectively, provided the required entropies have been computed. Similarly `s.pola_decomp()` will return the computed values for the decomposition of the mutual information presented in Pola et al. (2003), again provided the required entropies were computed.

The module has been designed to be as flexible as possible, allowing comparison of the different methods at every stage. For example, the `DiscreteSystem` instance contains the sampled probability distributions, so it is possible to compare the different probability estimation methods directly. It is easy to add additional entropic

³See <http://code.google.com/p/pyentropy/>

⁴See <http://www.strimmerlab.org/software/entropy/index.html>

⁵See <http://neuroanalysis.org/toolkit/>

⁶From <http://nsb-entropy.sourceforge.net/>

quantities or new functions of them to the class. The code is documented through use of Python *docstrings*, which are embedded in the source and accessible through the interactive interpreter. Having the code documented in this way makes it easier for others to understand and contribute to.

There are several properties of Python that make it well suited to this application. Many loops can be vectorised into a single operation acting on arrays which is implemented through the NumPy interface to a highly efficient linear algebra library (ATLAS). When taking slices (extracting a single row or column) of a NumPy array, for example when determining the independent probabilities of the X variables, a new *view* is created, but points to the same original data. In contrast, in MATLAB, taking such a slice always results in the extracted row being copied in memory to a new array object. As discussed, the object-oriented nature of Python allows code reuse through inheritance. To give an example of the performance of the Pyentropy library, for the preparation of the data for the Plugin, PT and QE methods in **Figure 1**, the time taken using the Pyentropy library on a 2.4GHz Core 2 Duo laptop was 439 s. This includes data simulation for 50 trials at each sample size. The same task, using similar MATLAB code on an equivalent laptop was 987 s. There is also work in progress to extend the Pyentropy code with a more direct calculation of the core estimates in Cython. Cython is a language for writing C extensions to Python, and it shares a very similar syntax. This provides an easy way to quickly develop fast C modules to speed up the execution of Python code.

FINITE ALPHABET MAXIMUM ENTROPY SOLUTIONS CORRELATIONS AND MAXIMUM ENTROPY MODELS

Simultaneous recordings of the activity of individual neurons placed within local networks in the central nervous system show that most pairs of neurons are weakly correlated: the probability of observing simultaneous spiking is typically slightly – but significantly – different to the product of the probability of observing the individual spikes (Averbeck et al., 2006; Mastronarde, 1983). These correlations are hypothesized by many investigators to be a fundamental part of the neural population code; they may contribute, for example, by tagging the occurrence of particular salient stimulus combinations (Gray et al., 1989), or by constraining the number of possible network states so that the network may perform error corrections (Schneidman et al., 2006). Whatever the role of correlated firing, an observer of neural activity (either a data analyst or a downstream neural system) trying to assess the importance of correlated activity has to face a hard problem: correlations are difficult to sample because they are described by a number of parameters that increases exponentially with the number of cells considered. Therefore, it is important to establish whether it is possible to describe all correlations between neurons with a small number of parameters that preserve all the relevant features of the joint distribution of simultaneous responses. One way to find compact representations of the correlation structure of response probability can be obtained by using the technique of *maximum entropy* (Montemurro et al., 2007b; Schneidman et al., 2003; Tang et al., 2008; Victor, 2006), as follows.

The question addressed by maximum entropy models is how well we can describe all interactions between all variables in terms of subsets of interactions between up to K variables only, or whether

and to what degree higher order interactions are present and important. The maximum entropy technique compares the measured response probability to one that takes into account all the observed interactions of up to K elements but does not impose any additional structure on the data. Measuring all interactions of up to K variables means measuring all the marginal response probabilities involving up to K variables. Therefore any probability matching the observed interactions of up to K elements must obey (apart from the usual non negativity and normalization constraints) the following linear constraints. Here we consider a response vector $\mathbf{r} = \{r_1, \dots, r_L\}$ of dimension L , with each variable r_i taking values from a finite alphabet \mathbf{A} containing m elements.

$$\begin{aligned} P_K(r_i) &= P(r_i) \equiv \eta_i^{r_i} \\ P_K(r_i, r_j) &= P(r_i, r_j) \equiv \eta_{ij}^{r_i r_j} \\ &\dots \dots \\ P_K(r_{i_1}, \dots, r_{i_K}) &= P(r_{i_1}, \dots, r_{i_K}) \equiv \eta_{i_1 \dots i_K}^{r_{i_1} \dots r_{i_K}} \end{aligned} \quad (9)$$

Each line above denotes a family of constraints on a model distribution $P_K(\mathbf{r})$ enforcing equality of the marginal values of a given order to those of the true distribution $P(\mathbf{r})$. These marginals are denoted by η with subscript indices representing the variables involved in the marginal and superscript indices the corresponding values. The a^{th} order constraint applies for all unique combinations of a variables, and every permutation of possible values that those variables can take. Thus the a^{th} line above represents $m^a \binom{L}{a}$ constraints, the product of permutations of a values with choices of a variables.

The probability distribution $P_K(\mathbf{r})$ with maximum entropy among those satisfying the above constraints is the one that does not impose the presence of any additional higher order correlations or interactions between the variables. To choose a distribution with lower entropy would correspond to the assumption of some additional structure that we do not know; to choose one with a higher entropy would necessarily violate the constraints that we wish to enforce.

Following Amari (2001); Cover and Thomas (2006) it can be shown that there is a unique solution to the constrained maximum entropy problem, which can be written in the following exponential form:

$$P_K(\mathbf{r}; \theta) = \exp \left\{ \theta_0 + \sum_{a=1}^K \left[\sum_{\substack{1 \leq i_1 < \dots < i_a \leq L \\ r_{i_1}, \dots, r_{i_a} \in \mathbf{A}}} \delta_{r_{i_1} \dots r_{i_a}}^{r_{i_1} \dots r_{i_a}}(\mathbf{r}) \theta_{i_1 \dots i_a} \right] \right\} \quad (10)$$

The set of indices i_1, \dots, i_a label the subsets of a variables among the total L considered. The set of indices r_{i_1}, \dots, r_{i_a} labels a specific set of values of these variables. The first term in the sum is a finite alphabet Kronecker delta function which takes the value 1 when the variables of the argument specified by the subscript indices take the values specified by the superscript indices, and 0 otherwise. As with the marginal constraints, the second sum for each order is over all unique combinations of a variables and all permutations of a values that those variables can take; there are $m^a \binom{L}{a}$ summands, and the same number of distinct θ coefficients of that order.

In order to compute the maximum entropy distribution $P_K(\mathbf{r}; \theta)$ compatible with all the known interactions up to K -th order, we need to find the θ coefficients with up to K indices to construct the solution above. These can be determined from the knowledge of the experimental η marginal probabilities of up to K elements through a set of algebraic equations, as detailed in the following section.

Previous applications of the maximum entropy approach have included temporal sequences of spiking activity, or multi-unit spiking activity across a population, both of which are binary. This simplifies the calculation of the maximum entropy solutions. The extension to a finite alphabet probability space is a significant one, since it greatly increases the scope of possible applications for the method. For example, if larger time bins are used, there will sometimes be more than one spike occurring in each bin. At the moment these values are generally binarized, but using the finite alphabet method allows use of extended time bins, while keeping the effect of all spikes. It can therefore be used to investigate the effect of bursting. Similarly, the finite alphabet extension means the method can be applied to other data, such as LFPs (Belitski et al., 2008) or fMRI, which are inherently continuous but may be meaningfully quantised into a finite alphabet. It also allows investigation of the reverse problem, neural encoding, where one studies the properties of the stimulus, given that a response (such as a spike) as occurred.

In the following, we describe an implementation of the finite-alphabet maximum entropy computation using Python. In analogy to Schneidman et al. (2003), we apply the maximum entropy calculation to $P(\mathbf{r})$. However, the same procedure could be in principle applied to $P(\mathbf{r}|s)$.

AN ALGORITHM FOR FINITE-ALPHABET MAXIMUM ENTROPY SOLUTIONS

The key concept in the algorithm we use to obtain the maximum entropy solution is the idea of identifying a specific probability distribution using different *coordinate systems*. The most obvious way of characterising a discrete probability distribution is by specifying the full list of probabilities for each element of the space. For example, if we have a finite alphabet response vector $\mathbf{r} = \{r_1, \dots, r_L\}$ as above, then there are m^L possible values for \mathbf{r} and so the probability distribution $P(\mathbf{r})$ can be characterised by $m^L - 1$ probability values, since one degree of freedom is removed by the normalisation constraint. These are called the p -coordinates. An alternative way of uniquely determining a probability distribution is by listing the marginal probability values. As mentioned in the previous section, there are $m^k(k)$ marginals containing of order k , so the collection of all marginals has $\sum_{k=1}^L m^k(k) = m^L - 1$ elements. This way of describing the probability is called the η -coordinates. For the final characterisation of a probability distribution, we consider the form suggested by Eq. 10. Taking $K = L$, $P_K(\mathbf{r}) = P(\mathbf{r})$ and Eq. 10 shows that any probability can be computed from the set of coefficients, θ . Again there are $m^k(k)$ coefficients of each order k . θ_0 is fixed by the normalisation condition, so again we have $m^L - 1$ numbers that uniquely identify the probability distribution. Expressing a probability distribution in this way is also known as the *log-linear* form, and the coefficients, θ are called the *log-linear effects*. Here we refer to them as the θ -coordinates.

A given probability distribution is represented in any of these coordinate systems by a vector of values. In the following \mathbf{p} denotes a vector describing a probability distribution in the p -coordinates, η denotes a vector of η -coordinate values and θ a vector of θ -coordinates. The \mathbf{p} vector is ordered so that the value of the vector at a given index represents the probability of the underlying state which, when interpreted as a length L base m word, has the decimal value of the index. This ordering was chosen since it is easy to convert between state values and vector indices using existing change of basis functions. The vector $\eta = (\eta_1, \eta_2, \dots, \eta_L)$ where η_i is the set of all marginals of order i and similarly $\theta = (\theta_1, \theta_2, \dots, \theta_L)$. The ordering of the vector within the subsets of different orders is arbitrary, however it is important that the subsets θ_i and η_i share the same ordering for each i .

These notions are rigorously developed in Amari (2001) using the framework of information geometry, in which the set of probability distributions on a given vector space are treated as a manifold, and the properties of the coordinate systems described above are formalised.

Coordinate Transformations

An important step in the numerical method for obtaining the maximum entropy solution is the implementation of the transformations between the different coordinate systems described above for representing a probability distribution.

η - p transforms. The key transformation is that from p -coordinates to η -coordinates. This is a linear transformation which performs the summation of relevant probabilities for calculating the marginal. With the coordinates arranged in vectors, as described above, it can be expressed as

$$\eta = A\mathbf{p} \quad (11)$$

where A is a square matrix containing binary values. Each row of A contains a 1 in the column for each p coordinate that contributes to that marginal. The inverse transformation, p coordinates from η coordinates is simply

$$\mathbf{p} = A^{-1}\eta \quad (12)$$

The matrix A is invertible since it is square and all its constituent rows are linearly independent.

θ - p transforms. For the θ - p transformations, first notice from Eq. 10 that in vector form $\mathbf{p} = e^{\theta_0 + A^T\theta}$. This is because, for a given probability, the θ terms required are those corresponding to the non-zero elements of that specific state vector. Similarly, for a given probability, that probability will appear in the sum for the marginals corresponding to the same non-zero elements of the state vector. The marginals that a given probability appears in are given by the columns of the matrix A , so provided the θ vector is ordered in the same way as the η vector, the sum of θ terms required in the exponential of Eq. 10 for each probability is given by $A^T\theta$. By evaluating Eq. 10 for the zero state vector $p_0 = P(\{r_i = 0\}_{i=1}^L)$ we see that the constant factor in the log-linear model, e^{θ_0} , is in fact p_0 . From $\mathbf{p} = p_0 e^{A^T\theta}$, it is trivial to obtain the following transformation from p coordinates to θ coordinates.

$$\theta = A^{-T} [\ln \mathbf{p} - \ln p_0] \quad (13)$$

The other direction is slightly more complicated, since for a closed expression for \mathbf{p} we must compute p_0 from the theta vector. The normalisation condition requires that $\sum \mathbf{p} + p_0 = 1$, since the vector \mathbf{p} does not include the p_0 value. Substituting the expression above gives $p_0 \sum e^{A_{i0}} + p_0 = 1 \Rightarrow p_0 = (1 + \sum e^{A_{i0}})^{-1}$, yielding

$$\mathbf{p} = \frac{e^{A_{i0}}}{1 + \sum e^{A_{i0}}} \quad (14)$$

Numerical Optimisation

The advantages of the different coordinate systems described above are that they allow us to easily represent our constraints on the maximum entropy solution. From Eq. 9 fixing interactions up to order K to those of the measured distribution corresponds to setting the low order η -coordinates of the maximum entropy solution equal to those of the measured distribution. From Eq. 10 the maximum entropy constraint is enforced by setting the high order components of the θ -coordinates to zero. By enforcing these constraints simultaneously, we obtain a set of N simultaneous equations in N unknowns, where $N = \sum_{j=1}^k m^j \binom{L}{j}$ is the number of coordinates up to order k . Again m is the size of the finite alphabet.

In the following η_k represents the N low order (up to order k) marginals of the sampled distribution. $\bar{\theta}_k, \bar{\theta}_{k^+}$ represent the low and high order theta coordinates of the maximum entropy distribution. $\check{p}(\cdot)$ denotes the coordinate transformation from θ to p coordinates from Eq. 14 and $\check{\eta}_k(\cdot)$ denotes the coordinate transformation in Eq. 11 but with only the low order marginals returned. Setting the high order theta's, $\bar{\theta}_{k^+}$, to zero ensures that there are no higher order interactions. It is then possible to find the low order theta's that produce the same low order marginals as the sampled distribution, η_k . These low order theta's, $\bar{\theta}_k$, completely characterise the maximum entropy distribution. In vector form the equations are:

$$\eta_k - \check{\eta}_k[\check{p}(\bar{\theta}_k, \bar{\theta}_{k^+} = 0)] = 0 \quad (15)$$

Once the $\bar{\theta}_k$ are determined by numerically solving the equation above, one can convert back to p -coordinates to obtain the corresponding maximum entropy distribution and calculate its entropy.

PYTHON IMPLEMENTATION

Initially the method described above was implemented in MATLAB. Later, the same algorithm was converted to Python with NumPy and SciPy. This was both because we were having performance issues with MATLAB in the finite alphabet case, and partly as a way to evaluate Python as a platform for our work. This gives the opportunity to make comparisons between the two systems. However, as well as moving the code to Python, we continued to develop and improve the algorithms, making it difficult to provide rigorous performance comparisons between the two systems. Instead we hope to provide an overview of our experiences and impressions of using Python in an ongoing research project.

A major difference in the code between the two systems is the structure of the program. In MATLAB the notion of the

global *workspace* was exploited. Here a setup script is used to define the coordinate transformation functions in the global workspace, from where they can be easily called by other scripts or used to interactively investigate data. In Python, an object-oriented approach was taken featuring two main classes. The first of these, *AmarISolve*, contains the parameters related to the underlying probability distribution, the required coordinate transformations and the code for performing the numerical solution. This is initialised with two parameters, the number of variables and the finite alphabet of each variable, since this is the only information required to implement the solution. The second class, *AmarISystem*, contains the data related to a specific system being studied, and contains the sampled probability distributions, calculated maximum entropy distributions and associated entropies. In this way the data independent analysis code is separated from the system specific code and data – the idea being that a single *AmarISolve* instance can be used on different data sets, providing the dimensions of the probability space are the same. It was found this approach gave much more flexibility than the global workspace, which could be confusing to manage during development, for example by requiring a full copy of the setup script to be maintained for every change to the algorithm investigated.

A key step in the implementation of the algorithm is the generation of the matrix A which provides the transformation between probabilities (p -coordinates) and marginals (η -coordinates). A recursive function is used in a loop over each order, to compute the elements of A row by row. The code implements the long-hand approach used for manual calculation of smaller matrices. The idea is that each marginal is the sum over all variables not fixed by the specification of the marginal. For each order a vector called *terms* is created which contains all base m words of length $L - o$, where o is the order being considered. Then for each marginal, if columns of the appropriate value are inserted into the appropriate position in the *terms* array, the result contains a row for each probability state included in that marginal. These are converted to decimal, which directly gives the index in the probability vector, and the corresponding columns in A are set to 1. To cover the different marginals, first the alphabet value and then the position is looped over. For orders higher than one, this process is recursive, so the first alphabet value is looped over, then within that the first position, then within that the second alphabet, then the second position and so on. This transformation matrix can be very large since its dimensions are the dimensions of the full probability space. However, it is highly sparse in structure, so in both implementations the provided sparse array construct was used to reduce the amount of memory required. In SciPy, the sparse array module is very flexible, providing a number of formats and datatypes. The advantage of this was that the binary matrix A could be stored as a sparse array of 8-bit integers in SciPy, which provided a factor of eight memory saving over the 64-bit double which is the only type the MATLAB sparse matrix supports. Equations 12 and 13 show that some coordinate transformations require inversion of the matrix A . Although this is not required directly for the computation of the maximum entropies, it was frequently useful while investigating properties of the system and of the different maximum entropy solutions. SciPy offers a very

flexible direct interface to the UMFPACK⁷ library of sparse solvers (Davis, 2004), that allowed us to easily pre-factor the matrix and store the results allowing rapid calculation of the coordinate transforms when needed.

The numerical optimisation step is very similar in both implementations, using the `fsolve` function of the respective system. In MATLAB a Gauss–Newton method was used, while in SciPy `fsolve` is a wrapper around the MINPACK (Moré et al., 1999) hybrid algorithm which implements a modification of the Powell hybrid method. Both of these methods performed similarly. The function that the optimiser runs is the same in both implementations and this is a direct implementation of the left hand side of Eq. 15; the Python version is shown below. Here `Asmall` is a subset of the transformation matrix *A* containing only the rows required and `Bsmall` is the transpose of this. `Asmall` is extracted from *A* using the `slice` operator, for example in Python, `Asmall = A[:, :]`. Python again provides a significant advantage here in terms of memory used. In MATLAB, any such slice results in a *copy* of the data. However, with NumPy, the slice results in a *view* of the original data. Similarly, in NumPy the transpose is also a view, with a different starting point and striding, but the same data buffer as the original array. In MATLAB the transpose operation also produces a copy.

```
def solvefunc(self, theta_un, Asmall, Bsmall, eta_sampled):
    b = np.exp(Bsmall.matvec(theta_un))
    y = eta_sampled(Asmall.matvec(b)/(b.sum() + 1))
    return y
```

As the method was developed and applied to increasing large probability spaces, it became clear that the limiting factor for these more challenging parameter sets was the memory usage rather than the computation time. The Python implementation was therefore optimised to reduce the memory usage.

This enhancement was simplified by using the object-oriented features of Python. New classes were created which inherited from `AmariSolve` and `AmariSystem` described above. It was then possible to change only the required functions, for example the matrix generation routine, to stop at the required row. This minimised the other changes and duplication of code. Also, developing in this way meant very few changes were required to the analysis scripts to take advantage of this change – in most cases a simple substitution of the class name at the top of the script was enough to use the new method. One of the memory optimisations was to produce the matrix *A* in smaller blocks, writing the rows and columns of the non-zero elements directly to files on disk to reduce memory overhead. Once this procedure was completed a sparse matrix in coordinate (COO) format could be generated directly from these files, and then converted to compressed sparse column (CSC) format for efficient matrix-vector multiplication. This is another example of where good results were obtained by using low level features that would not have been available in MATLAB.

As an example of the relative performance of Python and MATLAB, maximum entropy solutions of up to second order were computed for a system with $n = 4$, $m = 9$ (four variables each taking 1 of 9 values). The MATLAB code took 17 s with a peak resident

memory usage of 340 MB and the Python code took 12 s with a peak resident memory usage of 110 MB. These results are typical of our experience across a range of parameter values. The numerical optimisation routine took almost exactly the same time in both systems, with the difference being due to the improved performance of the sampling of the probability distributions in Python. This is likely to be due to the reduced amount of data copying needed with NumPy when using slicing and other array operations.

In conclusion, for the development of this technique the use of Python with NumPy and SciPy libraries as an alternative to MATLAB was highly successful. The computational speed was very similar, but using NumPy allowed us to reduce the memory requirement by around two-thirds. This is important, because as described above, memory usage was the limiting factor restricting the size of the probability space over which the analysis could be performed. As well as the vectors representing the actual probability distribution, the sparse matrix *A* must be calculated and held in memory. The ability to use an 8-bit integer for this binary matrix with Python provided a factor of 8 memory saving over the MATLAB equivalent. More significantly, the algorithm requires extraction of the submatrix of up to the relevant order, and the transpose of that, which in MATLAB consists of copies (meaning for each order the data is copied in memory three times, once for the full matrix *A*, once for the extracted `Asmall` for the given order, and once for the transpose thereof, `Bsmall`). As an example, this meant that on a workstation with 2 GB of RAM the largest binary probability space that could be analysed up to order 3 was 12 variables for the MATLAB implementation, but 18 variables for the Python version. It is also worth noting that, while being similar to MATLAB, the Python language is a great pleasure to work with.

Example of application to thalamic neural recordings

To illustrate the application of maximum entropy techniques, here we compute maximum entropy models from a neuron in the ventro posterior medial nucleus (VPM), which is the principal whisker-related relay nucleus in the rat thalamus. Using extracellular microelectrodes, we recorded the responses of single VPM units in anaesthetised rats whose whiskers were mechanically stimulated with a piezoelectric wafer driven by a low-pass filtered white noise (see Montemurro et al., 2007a, for details). We used two types of white noise stimulation. The first sequence was identical on every trial (repeated stimulus); the second was independently generated on every trial (non-repeated stimulus). **Figure 2B** shows a raster plot of the spikes fired by a single neuron in response to 70 repetitions of the stimulus in **Figure 2A**. As previously reported (Montemurro et al., 2007a; Petersen et al., 2008), VPM responses to white noise were highly repeatable and temporally precise. An information theoretic analysis of these data revealed that these neurons convey information at sub-ms temporal precision (Montemurro et al., 2007a) and that there are correlations between the times of individual spikes. One source of correlation came from the refractoriness of neurons, and another source of correlation came from their tendency to fire spikes in bursts (Montemurro et al., 2007a). An important question is whether these correlations between the times of spikes emitted by the same neuron have a significant impact on the information and entropy of the neural spike train, and if these correlations can be described by simple pairwise models or if

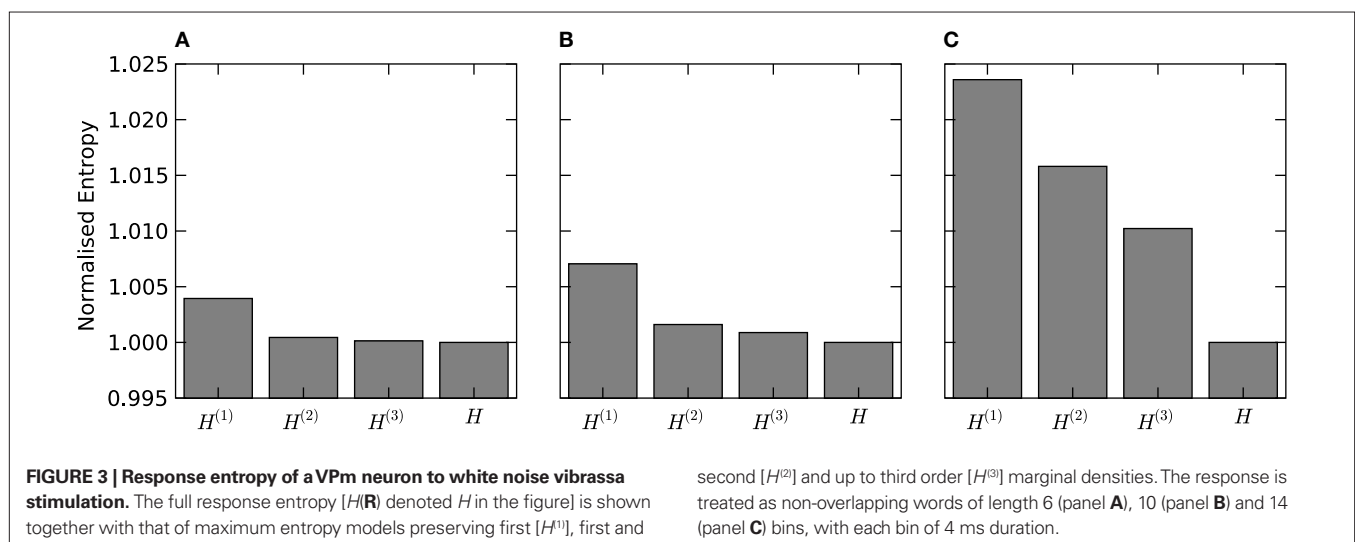
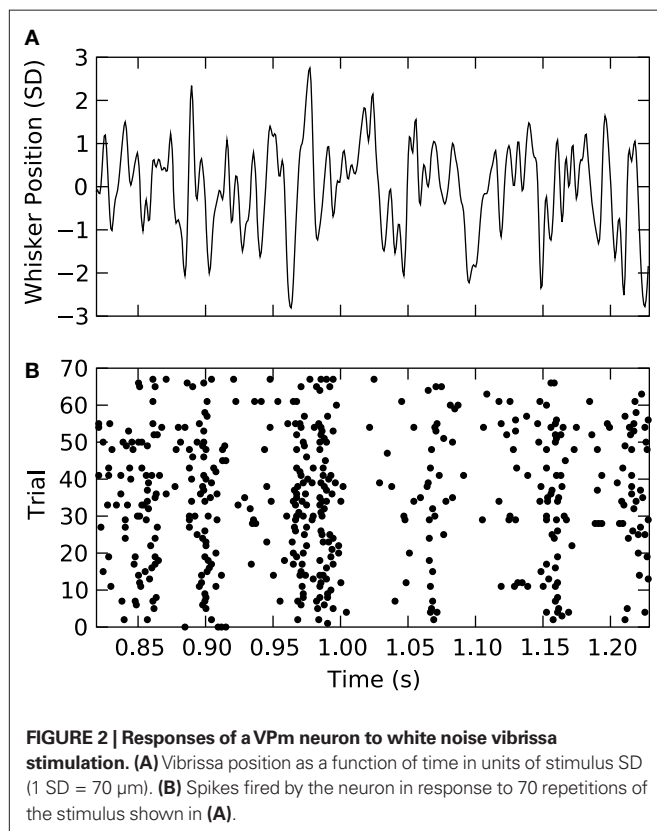
⁷<http://www.cise.ufl.edu/research/sparse/umfpack/>

they rather need a complex, high order characterization. Here we will address these questions by using maximum entropy models which, as explained above, provide a natural framework to study the impact of different orders of correlation to spike train entropy and information. Previous studies employing maximum entropy have focussed mainly on correlations across a population of neurons (Schneidman et al., 2006; Shlens et al., 2006). Here, we extend this study to focus on correlations in time between spikes of a single neuron. This is interesting because finding a compact maximum

entropy representation of within cell correlations is an important step towards understanding spike timing codes and representing them efficiently (Nirenberg and Victor, 2007; Tang et al., 2008).

We discretized the time into small bins of size $\Delta t = 4$ ms and quantified the response of the considered VPM neuron as a binary sequence of 1's and 0's (spikes or silence in that bin respectively), characterising the neural response \mathbf{r} as non-overlapping binary words of length L extracted from this signal. We then considered the probability of response $P(\mathbf{r})$ in response to all patterns of whisker stimulation obtained from the non-repeated white noise sequences, and we compared its entropy to that of the maximum entropy probability $P_K(\mathbf{r})$ at level K ($K = 1, \dots, 3$) and to the entropy of the true distribution. Results are reported in **Figure 3**. We found that the lowest order model ($K = 1$, which considers spikes in each bin as independent from each other) provides an entropy very close to that carried by higher order probability models. The difference between lower and higher order entropies becomes proportionally larger as the length L of the binary word increases. However, differences remain small: for $L = 14$, the difference between the independent-model, $K = 1$ entropy and the true one remain within 3%. This suggests that the spike train could be quantitatively well described even by a simple model that ignores correlations between spikes at different time bins. It should be noted that in the Python implementation of this calculation, the limit on the maximum number of time bins L and the order K that could be analysed was set by the number of trials available and the effectiveness of the sampling bias corrections implemented, whereas in the corresponding MATLAB implementation the limit was reached when the available memory was consumed. For a binary system as described here that limit was $L = 12$, $K = 2$ on our workstation. This highlights the advantages of Python for these implementations.

It should be noted that while we are applying the analysis here to data from a single cell, the computational challenge is determined solely by the dimension of the underlying probability space. In this case, the largest underlying probability space considered has a dimension of 2^{14} which is computationally equivalent to the case of the binary response of 14 simultaneously recorded neurons.



COLLABORATIVE COMPUTING

There is a growing trend in neuroscience towards the development and use of collaborative computing services. These are multi-user systems, accessed over the internet which provide computational resources while facilitating interaction between users. This is a natural evolution for the field, as rapid advances in physiological techniques of many kinds result in data sets of increasing size and with an associated proliferation of analysis tools of increasing complexity. The idea is to provide an environment to foster collaboration, especially between experimentalists and theoreticians, by providing databases of experimental results, and online analytical tools for application to those data.

The field of bioinformatics has pioneered the development of such systems, which are now well established and playing an important role. However, implementing such systems for neuroscience presents some challenges not faced by the bioinformatics community. The greatest of these is the volume and variety of experimental data. While traditional bioinformatics services tend to process data as strings – which is partly why the Perl programming language still underpins much bioinformatics analysis – in neuroscience we deal with large sets of binary data in a variety of different formats. This presents difficulties for the decentralised model of separately provided and hosted services that has become popular in the bioinformatics community. This data requires significant contextual detail, or metadata, to be useful and is large enough to make the sharing of terabytes of data between labs a significant issue. It therefore seems that neuroscience requires a stronger organisational structure for these systems, to facilitate easier interoperability of data and provide security and access control.

The adoption of Python is highly advantageous in this context. The Python language is flexible, extensible and runs on a wide range of platforms. It also has the fast array mathematics crucial for neuroscience work, which are not available in languages such as Perl, which have been traditionally used for bioinformatics services. Like Perl though, it is a dynamic interpreted language, which simplifies the deployment of code on distributed systems. It has a similar syntax to MATLAB, the established standard in the field, and although there are no automated tools, translating code and algorithms from one to the other is relatively straightforward. Unfortunately it is difficult to use MATLAB to provide these kinds of multi-user services due to licensing restrictions. We are working on adapting our information theoretic techniques for use in systems of this type, and this was one of the factors that influenced our decision to investigate Python.

The Code, Analysis, Repository and Modelling for e-Neuroscience (CARMEN)⁸ project is a consortium effort to create a virtual laboratory for neurophysiology (Gibson et al., 2008), and is one example of project attempting to provide a centralised organisational structure for collaborative computing in neuroscience, as discussed above. CARMEN is an e-Science Pilot Project funded by the Engineering and Physical Sciences Research Council (UK) and involves investigators from 11 UK universities.

The goals of the CARMEN project are to create a decentralised computing resource used by experimentalists and theoreticians alike; a repository for both experimental data and analysis code that

can be made available to all users of the system. We are working to provide our Python-based information theoretic algorithms as “services” on the CARMEN system. Providing such packaged services as modules that can be used in easy to construct “workflows” has many advantages. It allows easy comparison of different analytical techniques on the same dataset, as well as allowing application of a given technique to a number of different datasets that might otherwise be hard to obtain or convert to a suitable format. It allows application of the techniques of information theory by experimentalists and others who may otherwise lack the mathematical background, programming skills or inclination to implement such techniques by hand from the literature. It should also allow better reproducibility of published results, as well as providing a substantial computational resource allowing calculations that could be too time consuming for a user to perform on a desktop computer.

PYTHON WEB SERVICES

A “web service” is “a software system designed to support interoperable machine-to-machine interaction over a network”⁹. Web services are well suited to collaborative computing services, and they have been proven as a successful model for e-Science through their use in the bioinformatics community. They are also used as the foundation of the analysis code in the CARMEN project described above. Web services are operating system, location and language neutral. This is exploited in CARMEN to allow dynamic deployment of services to different computational nodes, and also simplifies the use and integration of analysis code written in a range of languages.

There are a number of standards governing the behaviour of web services, largely provided by the World Wide Web Consortium (W3C), which are required to allow them to interact. The fact that these standards are vendor neutral has enabled them to gain traction where previous attempts to provide interoperable services has failed. Simple Object Access Protocol (SOAP)¹⁰ is a standard XML based messaging format used to pass data and parameters to an analysis service, and then receive the results back. All clients and web services are capable of passing and decoding SOAP messages. The other pivotal standard is that of the Web Services Description Language (WSDL)¹¹, an XML document for the description of a web service; that is the method calls it provides, the arguments they require and the results they return. The WSDL that represents a web service is sufficiently informative to allow automatic generation of clients capable of binding to the service.

As part of our work we are making the information theoretic techniques that we are developing available as web services, for use in CARMEN and similar systems. Python greatly eases this process. We can create a Python-based service for a specific information theoretic task simply by importing our information theoretic library and calling the appropriate function with the appropriate arguments. This reduces code repetition, and the flexibility and simplicity of the Python module system makes the process easy to manage. For example, if the algorithmic code was actually

⁸<http://www.carmen.org.uk/>

⁹<http://www.w3.org/TR/ws-gloss/>

¹⁰<http://www.w3.org/TR/soap/>

¹¹<http://www.w3.org/TR/wsdl>

included in the service programs, this would exist in every service performing an information theoretic calculation with a copy on every node to which the service had been deployed. By having a library with a consistent API, this can be updated in a single place on each computational node without having to change any of the existing services.

Once there is a Python script to perform the required task, it is necessary to “wrap” it to create a web service. There are a number of toolkits to do this including the Python native Zolera SOAP Infrastructure (ZSI) and SOAPpy. However, the method we have been using is InstantSOAP¹² a generic toolkit capable of exposing legacy applications as web services. Initially, we have created Python scripts that run as command line applications. This is straightforward since Python includes an excellent tool for easily parsing command line options. InstantSOAP provides a native command line processor to wrap any command line application into a web service through the creation of a single XML file. Work is currently in progress to extend InstantSOAP to natively support Python services, allowing direct deployment of a Python function as a web service, without requiring the developer to understand the web services stack, a significant barrier to entry in developing web services in any language. Python’s licensing model is also important in the deployment of distributed services; MATLAB suffers from licensing restrictions for collaborative deployment. This makes it harder both to provide open services to a large number of users and to employ the dynamic deployment architecture through which code may run on a number of computational nodes. For example, whilst CARMEN is capable of providing MATLAB web services, it is through compiled MATLAB scripts, supported by the MATLAB runtime environment, and has no native interface to MATLAB *per se*, adding additional complexity to the procedure of creating, deploying and managing web services. There are also a number of ongoing technical challenges related to running the compiled MATLAB binaries within the web service environment.

DISCUSSION

In modern neuroscience a growing challenge is handling and interpreting increasingly large volumes of physiological data of many different types. To face this challenge computational techniques are becoming more and more important. We have described information theory, which is one such technique that is particularly suited to the challenges posed by neurophysiological datasets, and can provide valuable insights into neural coding and the function of the nervous system.

Information theory provides a natural framework to study communication in most systems, and the brain is no exception. An obstacle to a wider spread of its use among sensory neurophysiology laboratories has been the technical difficulties associated with its calculation (mostly the problem of bias corrections) and the lack of well defined, cross-platform packages that can handle generic datasets. The work presented in this paper is an attempt to address this limitation and provide the neuroscience community with open source packages that allow unbiased calculation

of information from various types of neural data, from spikes to field potentials. The use of Python helps to develop flexible tools that can easily be applied or extended (because of the flexibility of the Python language) to handle different types of neurophysiological signals (because of the ability to manage memory efficiently) and to different data formats (because of the ability of Python to easily read a variety of data formats commonly used in neuroscience).

We have also described a current area of intensive research on neural coding; namely a new implementation for computing solutions of maximum entropy given marginal constraints. Although the example presented in **Figure 3** was on a binary data space, the ability of the code to support finite alphabet probability spaces is significant and allows the application of the maximum entropy technique to a wide range of new areas. In our own experience with simulated data (results not shown here, but partly reported in Lüdtke et al., 2009), using the Python implementation described here we were able to solve maximum entropy solutions of order 2 on spaces of up to 7 variables quantised to 9 levels (a probability space with dimension ~ 4.7 m) on a well-equipped workstation in a reasonable amount of time (~ 1 day). This was a dramatic improvement over what we were initially able to achieve with the MATLAB version of code; indeed the MATLAB version would have been unable to solve for a system of that size due to memory limitations. Other potential finite alphabet applications include analysis of quantised naturally continuous signals, such as LFP or fMRI as well as opening the possibility of studying the interactions between the stimulus features encoded by spiking responses, where instead of response given stimulus we consider the properties of the stimulus given a response.

Looking to the future of inter-disciplinary science, we have considered the possibilities offered by collaborative computing services based on grid or cloud architectures. While such systems have been developed for use in other areas, neuroscience poses some unique challenges. We have outlined our work as part of the CARMEN project, which hopes to address these challenges and provide a valuable service for storage, processing and analysis of electrophysiological data. We are developing information theoretic analysis tools as web services, which will make them available to greater range of practitioners, and hopefully increase their use within the neuroscience community.

The development of analysis tools like the ones discussed here has potentially significant implications for the refinement, reduction and replacement (3R) of animals in research. In our specific case, the opportunity to easily run information analysis on a number of different existing datasets (which as discussed, is facilitated by Python) maximizes the probability of obtaining new insights into neural codes without the need to sacrifice new animals. The free availability of advanced routines for calculation of bias-corrected information estimates offers neurophysiological laboratories the possibility of reliably computing information from a smaller number of trials, thereby maximizing the potential to record from multiple sites in the same animal and thus reducing the total number of animals needed for statistical significance. The ability of the code to adapt to the different types of neural signals that can simultaneously be extracted from

¹²<http://instantsoap.sourceforge.net/>

the extracellular signal also increases the amount of information that can be obtained without increasing the invasiveness of the recording procedures.

We have found significant advantages to using Python for all of the work described above. As discussed, we have found it well suited both to reimplementing existing techniques for exposure to a wider audience, as open-source packages and hosted computational services, and to the research and development of new techniques and algorithms. Together with the excellent interactive environment IPython¹³, it provides much of the power available from low level C coding with a numerical library, but with greatly reduced complexity and development time. For example, a major advantage for our maximum entropy application was the way we were able to fine tune the use of the sparse matrix structures. The interactive nature, familiar to users of MATLAB, is crucial to aid research, both in terms of investigation of data as well as development of algorithms. Compared to MATLAB, we have seen performance increases in moving our code to Python, particularly related to memory management in the case of our more demanding algorithms. In addition, increased productivity and code manageability, for example from the ability to use object-oriented programming techniques, speed development and ease collaboration with other researchers.

We have experienced few problems with migrating our code from MATLAB. We have been able to easily access existing data stored in .MAT files and also to smoothly translate code. It is even possible to call MATLAB from Python, through the `mlabwrap` module¹⁴, which we have used to run existing MATLAB code provided by colleagues for preprocessing data. Initially the required packages were difficult to install, requiring compilation from source of a range of packages with complicated dependencies. Actually getting the software installed was therefore the greatest challenge when we began using Python. However, since then, the community has done a lot of work in improving this process, and there are now regular binary releases of all the important components, as well as a number of projects that distribute a complete scientific tool chain with all required components through a common installer¹⁵. Another challenge was adapting to the pass by reference semantics of Python rather than the pass by value style of MATLAB, as well as adapting to 0 based indexing. However, once these mental adjustments had been made we found ourselves more productive with Python than we were with MATLAB. Other disadvantages of Python are that the documentation of the included functions, while still available interactively, is not as comprehensive as that provided with MATLAB and the plotting functionality provided by `matplotlib`, is not quite as easy to use or well developed as the MATLAB version, especially with regard to 3D plotting.

We have been able to easily provide our Python code as web services, for integration into collaborative systems such as CARMEN, without requiring a significant time investment to adjust or tune

the code for this purpose. In fact, Python is an excellent fit for projects such as CARMEN. It provides the flexibility of dynamic interpreted languages such as Perl, that have traditionally been used to provide services in systems of this type, while including the fast array mathematics that are crucial for the efficient analysis of neurophysiological data. It is difficult to use MATLAB in systems such as this, due to licensing restrictions which pose problems, both for allowing multiple users to access the service, and for running the service on different nodes in a grid infrastructure. Obviously, with Python being open source, there are no such issues. The benefits of open source extend beyond collaborative computing projects however; there is a compelling open-access argument for avoiding expensive proprietary software in published scientific work.

So far we have only scratched the surface in terms of what is available in the Python ecosystem that could be of benefit for our work. The extensive collection of modules available for Python allow great flexibility, for example making it much easier to develop GUI interfaces and handle a wide variety of data formats. There are also several methods to easily extend Python code with natively compiled C extensions, to increase the performance of critical sections of code, while still allowing the interactive use and rapid development of Python. We are currently focussed on optimising our information theoretic codes through the use of Cython¹⁶, which we are finding significantly easier to use and less error prone than the MATLAB equivalent (the MEX interface). Another area we are actively investigating in the use of parallelism. In many cases our problems are *embarrassingly parallel*, for example calculating information theoretic bias-corrected quantities over a number of data sets or computing maximum entropy solutions of different orders and conditional distributions. A number of open source solutions exist for parallel computing with Python, and we are investigating using these features of IPython to easily distribute these types of jobs to available machines.

SUPPLEMENTARY MATERIAL

The Python library for information theoretic estimates described in Section “A Python Library for Information Theoretic Estimates”, including code for producing **Figure 1**, can be found at <http://code.google.com/p/pyentropy/>. The code for obtaining the finite alphabet maximum entropy solutions can also be found on that page. This code is provided as Supplementary Material on the conditions that (1) the authorship of the software shall be acknowledged, (2) the present article shall be correctly cited in any publication that uses results generated by the software, (3) any publication that uses results generated by our software shall correctly cite the original articles (cited in this paper) which developed any bias correction methods used.

ACKNOWLEDGEMENTS

This work was supported by the EPSRC “CARMEN” grant and by IIT. We are indebted to C. Magri, R. Senatore, F. Montani, N. Ludtke and M. A. Montemurro for useful discussions on the implementation of entropy methods and for important contributions to the development of the information theoretic algorithms.

¹³“An enhanced interactive Python shell and architecture for interactive parallel computing”, <http://ipython.scipy.org/> (Perez and Granger, 2007)

¹⁴“A high-level Python to MATLAB bridge”, <http://mlabwrap.sourceforge.net/>

¹⁵See for example <http://www.pythonxy.com/> and <http://www.enthought.com/products/epd.php>

¹⁶The Cython language, “C extensions for Python”, <http://cython.org/>

REFERENCES

- Amari, S. I. (2001). Information geometry on hierarchy of probability distributions. *IEEE Trans. Inf. Theory* 47, 1701–1711.
- Arabzadeh, E., Panzeri, S., and Diamond, M. E. (2004). Whisker vibration information carried by rat barrel cortex neurons. *J. Neurosci.* 24, 6011–6020.
- Averbeck, B. B., Latham, P. E., and Pouget, A. (2006). Neural correlations, population coding and computation. *Nat. Rev. Neurosci.* 7, 358–367.
- Belitski, A., Gretton, A., Magri, C., Marayama, Y., Montemurro, M. A., Logothetis, N. K., and Panzeri, S. (2008). Low-frequency local field potentials and spikes in primary visual cortex convey independent visual information. *J. Neurosci.* 28, 5696–5709.
- Borst, A., and Theunissen, F. E. (1999). Information theory and neural coding. *Nat. Neurosci.* 2, 947–957.
- Cover, T. M., and Thomas, J. A. (2006). *Elements of Information Theory*, 2nd Edn. Hoboken, NJ, John Wiley & Sons.
- Dan, Y., Alonso, J. M., Usrey, W. M., and Reid, R. C. (1998). Coding of visual information by precisely correlated spikes in the lateral geniculate nucleus. *Nat. Neurosci.* 1, 501–507.
- Davis, T. A. (2004). Algorithm 832: UMFPACK V4.3 – An unsymmetric-pattern multifrontal method. *ACM Trans. Math. Soft.* 30, 196–199.
- de Ruyter van Steveninck, R., Lewen, G., Strong, S., Koberle, R., and Bialek, W. (1997). Reproducibility and variability in neural spike trains. *Science* 21, 1805–1808.
- Fuhrmann Alpert, G., Sun, F., Handwerker, D., D'Esposito, M., and Knight, R. (2007). Spatio-temporal information analysis of event-related BOLD responses. *Neuroimage* 34, 1545–1561.
- Gibson, F., Austin, J., Ingram, C., Fletcher, M., Jackson, T., Jessop, M., Knowles, A., Liang, B., Lord, P., Pitsilis, G., Periorellis, P., Simonotto, J., Watson, P., and Smith, L. (2008). The CARMEN Virtual Laboratory: Web-Based Paradigms for Collaboration in Neuroscience. 6th International Meeting on Substrate-Integrated Microelectrodes. Reutlingen, Germany.
- Gray, C. M., König, P., Engel, A. K., and Singer, W. (1989). Oscillatory responses in cat visual cortex exhibit inter-columnar synchronization which reflects global stimulus properties. *Nature* 338, 334–337.
- Hatsopoulos, N. G., Ojakangas, C. L., Paninski, L., and Donoghue, J. P. (1998). Information about movement direction obtained from synchronous activity of motor cortical neurons. *Proc. Natl. Acad. Sci.* 95, 15706–15711.
- Hausser, J., and Strimmer, K. (2008). Entropy inference and the James–Stein estimator. Preprint, arXiv:0811.3579v1.
- Honey, C. J., Kotter, R., Breakspear, M., and Sporns, O. (2007). Network structure of cerebral cortex shapes functional connectivity on multiple time scales. *Proc. Natl. Acad. Sci.* 104, 10240–10245.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open Source Scientific Tools for Python. URL <http://www.scipy.org/>
- Latham, P. E., and Nirenberg, S. (2005). Synergy, redundancy, and independence in population codes, revisited. *J. Neurosci.* 25, 5195–5206.
- Lüdtke, N., Ince, R. A. A., Brown, M., Kell, D. B., and Panzeri, S. (2009). A comparative evaluation of entropy and variance based methods for sensitivity analysis. In Preparation.
- Mastronarde, D. N. (1983). Correlated firing of cat retinal ganglion cells. I. Spontaneously active inputs to X- and Y-cells. *J. Neurophysiol.* 49, 303–324.
- Miller, G. A. (1955). Note on the bias of information estimates. In *Information Theory in Psychology: Problems and Methods*, H. Quastler, ed. (Glencoe, Ill, Free Press), pp. 95–100.
- Montemurro, M. A., Panzeri, S., Maravall, M., Alenda, A., Bale, M. R., Brambilla, M., and Petersen, R. S. (2007a). Role of precise spike timing in coding of dynamic vibrissa stimuli in somatosensory thalamus. *J. Neurophysiol.* 98, 1871–1882.
- Montemurro, M. A., Senatore, R., and Panzeri, S. (2007b). Tight data-robust bounds to mutual information combining shuffling and model selection techniques. *Neural Comput.* 19, 2913–2957.
- Montemurro, M. A., Rasch, M. J., Murayama, Y., Logothetis, N. K., and Panzeri, S. (2008). Phase-of-firing coding of natural visual stimuli in primary visual cortex. *Curr. Biol.* 18, 375–380.
- Moré, J., Garbow, B., and Hillstrom, K. (1999). Minpack. URL <http://www.netlib.org/minpack>
- Nemenman, I., Bialek, W., and de Ruyter van Steveninck, R. (2004). Entropy and information in neural spike trains: progress on the sampling problem. *Phys. Rev. E* 69, 56111.
- Nemenman, I., Shafee, F., and Bialek, W. (2002). Entropy and inference, revisited. *Adv. Neural. Inf. Process. Syst.* 14, 95–100.
- Nirenberg, S., and Victor, J. (2007). Analyzing the activity of large populations of neurons: how tractable is the problem? *Curr. Opin. Neurobiol.* 17, 397–400.
- Panzeri, S. (1999). Correlations and the encoding of information in the nervous system. *Proc. R. Soc. B* 266, 1001–1012.
- Panzeri, S., Magri, C., and Logothetis, N. (2008). On the use of information theory for the analysis of the relationship between neural and imaging signals. *Magn. Reson. Imaging* 26, 1015–1025.
- Panzeri, S., Petersen, R., Schultz, S., Lebedev, M., and Diamond, M. (2001). The role of spike timing in the coding of stimulus location in rat somatosensory cortex. *Neuron* 29, 769–777.
- Panzeri, S., Senatore, R., Montemurro, M., and Petersen, R. (2007). Correcting for the sampling bias problem in spike train information measures. *J. Neurophysiol.* 98, 1064–1072.
- Panzeri, S., and Treves, A. (1996). Analytical estimates of limited sampling biases in different information measures. *Netw. Comput. Neural Syst.* 7, 87–107.
- Perez, F., and Granger, B. (2007). Ipython: a system for interactive scientific computing. *Comput. Sci. Eng.* 9, 21–29.
- Petersen, R., Brambilla, M., Bale, M., Alenda, A., Panzeri, S., Montemurro, M., and Maravall, M. (2008). Diverse and temporally precise kinetic feature selectivity in the VPM thalamic nucleus. *Neuron* 60, 890–903.
- Petersen, R., Panzeri, S., and Diamond, M. (2001). Population coding of stimulus location in rat somatosensory cortex. *Neuron* 32, 503–514.
- Pola, G., Thiele, A., Hoffmann, K., and Panzeri, S. (2003). An exact method to quantify the information transmitted by different mechanisms of correlational coding. *Netw. Comput. Neural Syst.* 14, 35–60.
- Rieke, F., Bialek, W., Warland, D., and Van Steveninck, R. (1999). *Spikes: Exploring the Neural Code*. Bradford Book. Cambridge, MA, MIT Press.
- Rubino, D., Robbins, K., and Hatsopoulos, N. (2006). Propagating waves mediate information transfer in the motor cortex. *Nat. Neurosci.* 9, 1549–1557.
- Schneidman, E., Berry, M., II, Segev, R., and Bialek, W. (2006). Weak pairwise correlations imply strongly correlated network states in a neural population. *Nature* 440, 1007–1012.
- Schneidman, E., Still, S., Berry, M., and Bialek, W. (2003). Network information and connected correlations. *Phys. Rev. Lett.* 91, 238701.
- Schürmann, T., and Grassberger, P. (1996). Entropy estimation of symbol sequences. *Chaos* 6, 414–427.
- Shannon, C. (1948). A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 379–423.
- Shlens, J., Field, G., Gauthier, J., Grivich, M., Petrusca, D., Sher, A., Litke, A., and Chichilnisky, E. (2006). The structure of multi-neuron firing patterns in primate retina. *J. Neurosci.* 26, 8254.
- Strong, S., Koberle, R., de Ruyter van Steveninck, R., and Bialek, W. (1998). Entropy and information in neural spike trains. *Phys. Rev. Lett.* 80, 197–200.
- Tang, A., Jackson, D., Hobbs, J., Chen, W., Smith, J. L., Patel, H., Prieto, A., Petrusca, D., Grivich, M. I., Sher, A., Hottowy, P., Dabrowski, W., Litke, A. M., and Beggs, J. M. (2008). A maximum entropy model applied to spatial and temporal correlations from cortical networks *in vitro*. *J. Neurosci.* 28, 505–518.
- Treves, A., and Panzeri, S. (1995). The upward bias in measures of information derived from limited data samples. *Neural Comput.* 7, 399–407.
- van Rossum, G. (1995). Python Reference Manual. CWI Reports CS-R 9525.
- Victor, J. (1999). Temporal aspects of neural coding in the retina and lateral geniculate. *Netw. Comput. Neural Syst.* 10, 1–66.
- Victor, J. (2006). Approaches to information-theoretic analysis of neural activity. *Biol. Theory* 1, 302–316.
- Waldert, S., Preissl, H., Demandt, E., Braun, C., Birbaumer, N., Aertsen, A., and Mehring, C. (2008). Hand movement direction decoded from MEG and EEG. *J. Neurosci.* 28, 1000–1008.

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

Received: 21 September 2008; paper pending published: 20 November 2008; accepted: 27 January 2009; published online: 11 February 2009.

Citation: Ince RAA, Petersen RS, Swan DC and Panzeri S (2009) Python for information theoretic analysis of neural data. *Front. Neuroinform.* (2009) 3:4. doi: 10.3389/neuro.11.004.2009

Copyright © 2009 Ince, Petersen, Swan and Panzeri. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



OMPC: an open-source MATLAB®-to-Python compiler

Peter Jurica* and Cees van Leeuwen

Perceptual Dynamics Laboratory, RIKEN Brain Science Institute, Wako-Shi, Saitama, Japan

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Ellif Müller, Brain Mind Institute, EPFL,
Switzerland

Dan Goodman, École Normale
Supérieure, France

***Correspondence:**

Peter Jurica, Perceptual Dynamics
Laboratory, RIKEN Brain Science
Institute, Hirosawa 2-1, 351-0198
Wako-Shi, Saitama, Japan.
e-mail: pjurica@brain.riken.jp

Free access to scientific information facilitates scientific progress. Open-access scientific journals are a first step in this direction; a further step is to make auxiliary and supplementary materials that accompany scientific publications, such as methodological procedures and data-analysis tools, open and accessible to the scientific community. To this purpose it is instrumental to establish a software base, which will grow toward a comprehensive free and open-source language of technical and scientific computing. Endeavors in this direction are met with an important obstacle. MATLAB®, the predominant computation tool in many fields of research, is a closed-source commercial product. To facilitate the transition to an open computation platform, we propose Open-source MATLAB®-to-Python Compiler (OMPC), a platform that uses syntax adaptation and emulation to allow transparent import of existing MATLAB® functions into Python programs. The imported MATLAB® modules will run independently of MATLAB®, relying on Python's numerical and scientific libraries. Python offers a stable and mature open source platform that, in many respects, surpasses commonly used, expensive commercial closed source packages. The proposed software will therefore facilitate the transparent transition towards a free and general open-source *lingua franca* for scientific computation, while enabling access to the existing methods and algorithms of technical computing already available in MATLAB®. OMPC is available at <http://ompc.juricap.com>.

Keywords: technical computation, Python, Matlab, compiler

INTRODUCTION

Scientific progress is optimally served when everyone has access to the relevant information. No matter how effective commercial organizations, such as publishers or software houses, are in distributing information; their copyright and proper use requirements are often an impediment to information sharing. Open-access scientific journals attempt to remedy this problem; but this is only a first step, involving the free distribution of scientific results. The next step is to make auxiliary and supplementary materials that accompany scientific publications, such as methodological and data-analysis procedures, open and accessible to the scientific community in the form of freely downloadable software.

Sharing software tools requires a common platform. Currently one platform dominates the sciences: MATLAB®. As a commercial product, this language has successfully conquered the market for scientific communication (Moler, 2004, 2006) because it is easy to adopt for beginners as well as professionals, and because of its policy to offer licenses at reduced rates to educational institutions. However, it does not meet our criteria to be used as a common standard for free sharing of software tools. Using a method implemented in MATLAB® requires a full MATLAB® license. Moreover, its core software is closed source, preventing users from verifying, updating, and improving it.

While some MATLAB® users find the features of the language sufficient and see no reason to switch to an alternative, those who want to move to another platform feel the weight of code already written in MATLAB® impeding on their decision. Developers who have tried to offer an open-source alternative have made efforts to offer a level of compatibility with MATLAB®. Examples of such

products are Octave and Scilab. None of these packages ever reached 100% compatibility and failed to meet the challenge of catching up with a platform with substantial financial support.

We propose OMPC as a possible alternative strategy to facilitate transition to an open-source platform. OMPC aims to offer a bridge between MATLAB® and Python. Development of the Python programming language project was started in late 1980s (<http://www.artima.com/intv/python.html>) at the National Research Institute for Mathematics and Computer Science in the Netherlands as an open-source scripting language for gluing components of an operating system. Today, powerful hardware allows Python to be used as a general purpose programming language. Over the years, the community contributing to the development of the Python language has grown considerably. Programmers and scientists alike are attracted by the simplicity of its syntax and its powerful set of features. Python is a good bet for a future free and open-source product that will develop far and fast enough to become the new *lingua franca* of technical computing (Fangohr, 2004; Langtangen, 2006).

Since the early stages there have been attempts to develop a Python package that offers certain features available in MATLAB®-compatible languages (<http://matpy.sourceforge.net/>). Scientific computation libraries were developed in the 1990s (Oliphant, 2006) and have been updated several times (Ascher et al., 2001; Oliphant, 2007), gaining in reliability, stability and versatility over years of development and use. The most important ones, especially in the context of our project, are *numpy*, *scipy* and *matplotlib* (<http://numpy.scipy.org/>, <http://www.scipy.org/> and <http://matplotlib.sourceforge.net/> respectively). The first two provide functions

largely equivalent to those of MATLAB®, while *matplotlib* is providing plotting functionality. Within the controlled development of Python, a proposal was made in 2000 to enhance Python with a feature that has been one of the major assets of MATLAB®: the availability of both matrix and element-wise operators (Zhu and Lielens, 2000). Another proposal has been to include a numerical array package *numpy* into the standard Python library, resulting in a revision of the buffer interface for the Python 3.0 (Oliphant and Banks, 2006). The new buffer interface facilitates the sharing of multi-dimensional data between different Python extension modules. All these developments point to an expanding role for Python in scientific computation.

The main problem with these packages is that each offers only a subset of MATLAB® features, but they lack a common, standardized interface. Our first aim, therefore, is to organize the available numerical libraries and provide them with a common interface. Our second aim is to provide 100% compatibility with MATLAB® syntax and with its dynamic interpreter (the MATLAB® engine). One advantage is that users will be able to download MATLAB® applications and run them for free. For programmers, OMPC offers the advantage of a free and open collaboration platform allowing reuse of code developed for the commercial MATLAB® platform without laborious rewriting.

OMPC is basically a translator of MATLAB® code to Python-compatible syntax. This paper discusses the compiler and the fundamental concepts that allow it to generate interpretable code; in particular code that will handle certain dynamic MATLAB® features not present in Python. For the generated code to work, OMPC needs to be complemented by a library that will ensure the proper interpretation of the translated code. We refer to this library as OMPClib. OMPClib contains, in particular, numerical objects that emulate the dynamical behavior of their MATLAB® counterparts. Proof-of-concept implementations of OMPClib that possess additional functionality just sufficient to reproduce the results of a spiking neural-network simulation (from Izhikevich, 2003) are presented in the Supplementary Material. OMPClib is a work in progress. A regularly updated version is found at the project's website (<http://ompc.juricap.com>). The current implementation of the OMPClib is an integral component of the OMPC package and is based on the extension modules *numpy*, *scipy* and *matplotlib*.

PROBLEM STATEMENT

In part, the translation of MATLAB® into Python code is a straightforward, technical problem. We need a compiler to generate Python compatible code from MATLAB® code (see The Compiler). In addition, there are four MATLAB® types (string, cell array, array, and slice) that have features not available in the corresponding Python objects. For these, we introduce Python objects that act as proxies for their MATLAB® equivalents (see Numerical Library).

The central, unique feature of the present translation problem is that both languages are interpreted languages, but have different dynamic features. Usually “dynamic” refers to a property of variable types and means that variables do not have to have a declared purpose or type – we refer to an object by its name and the interpreter decides at run-time if an operation on the variable is allowed. However, MATLAB® also adds dynamics to a number of other aspects of the language. The dynamic features of the MATLAB®

engine differ from those of Python as well as most other general-purpose interpreters, because of the specific purpose for which MATLAB® was designed. These issues include: array slicing, on-demand updating of the variable namespace and populating it with implied variables such as *nargin/nargout*, element-wise operations, and implied returns. The dynamic feature of MATLAB® that is the most difficult to implement in languages other than Python is the *nargin/nargout* implied variable. The slicing syntax, although available in Python, differs in syntax. In subsequent Sections “Array Slicing, Index Base 1”, “Dynamic Update of the Variable Name Space, Emulation of *nargin/nargout*”, “Assignments to Novel Variables, Assignments to Slices”, “Element-wise Operations” and “Implied Returns”, we show how each of these particular problems can be solved. In Section “The *mfunction* Decorator” we mention how OMPC allows integration of these solutions with a minimum impact on the structure of the original MATLAB® code. Our approach illustrates that it is possible, given enough knowledge of the compiler of a particular language, to interpret code written in an arbitrary programming language, provided that the emulated language has a subset of the features of the emulating one. This translation maxim may apply universally between any pair of languages. However, as we argue, Python in addition is syntactically close, sufficiently dynamic, and has a large enough library to enable translation that leaves the original structure intact.

Any platform for technical and scientific computation should keep up to the standards of speed and quality of MATLAB®. This is only possible if such a platform is built on the base of standard numerical packages. Indeed at the base of all of currently competing scientific packages we find ATLAS (Automatically Tuned Linear Algebra Software). This is the reason why results of operations on matrices are bit-by-bit equivalent in MATLAB®, Python, Octave and many other tools. Also the speed of execution of operations defined in this library does not change significantly between different engines. There is no essential difference in speed of execution compared to compiled languages like C/C++; C/C++ code written by the average user can even be slower compared to implementations available from the ATLAS BLAS/LAPACK libraries used by *numpy/scipy*. This is because optimization of the elementary operations is done automatically at the time of compilation of the library and the speed of the result is not affected by the programming language from which this library is initiated (except for translation of parameters). The functionality of many toolboxes of MATLAB® is dependent on a number of other open-source packages as well. These are all available to Python users and probably have already been wrapped into a Python package. For custom made, non-standard packages (MEX extensions), we still need a way to allow OMPC to use them. This issue is discussed in Section “OMPC Extensions”.

PROPOSED SOLUTION

An underappreciated aspect of Python, especially in scientific computing, is a feature known as introspection. Python offers built-in modules that allow run-time inspection of its own bytecode. Bytecode is the equivalent of the machine language in interpreted and just-in-time compiled languages. Introspection makes possible the run-time modification of the bytecode of a program, provided that the engine allows this. Python offers this facility. Where the

specific dynamic features of the MATLAB® engine have made it impossible for Python to interpret MATLAB® code directly, we show that with the help of introspection it is possible to emulate the remaining features. The following section presents specific features that together implement the proposed solution. Supplementary Material files on the project site include Python scripts that demonstrate the features presented in this section.

OMPC – A MATLAB®-TO-PYTHON COMPILER

OMPC is a compiler that translates MATLAB® code to functionally equivalent Python code. The design philosophy of OMPC is to enable seamless integration of existing MATLAB® code in Python programs. As a feature of convenience, OMPC allows automatic loading and translation of *.m* files using the Python *import* statement. Thus, assuming there is an *m*-function called *add* implemented in a file called *add.m*, an example Python session using this file would look as follows¹:

```
>>> import ompc
>>> import add
>>> add(1,2)
ans = 3
```

The steps taken during execution are schematically illustrated in **Figure 1**. They are:

1. **import ompc** – OMPC installs a so-called import hook into the current instance of the interpreter. This allows OMPC to act at every import statement and compile *m*-files to Python code on demand. From this point on it is possible to import *.m* files.
2. **import add** – the OMPC import hook is called and searches for *add.m* on the current path (an equivalent to MATLAB®'s

path variable). OMPC compiles *add.m* to a *.pym* file and submits this file to Python's built-in `__import__` function that will compile this file as any other regular Python file.

3. **add(1, 2)** – is a Python function call. It is running in the current Python instance as a Python function working with Python variables. In other words, MATLAB® is not involved at any stage of this process.

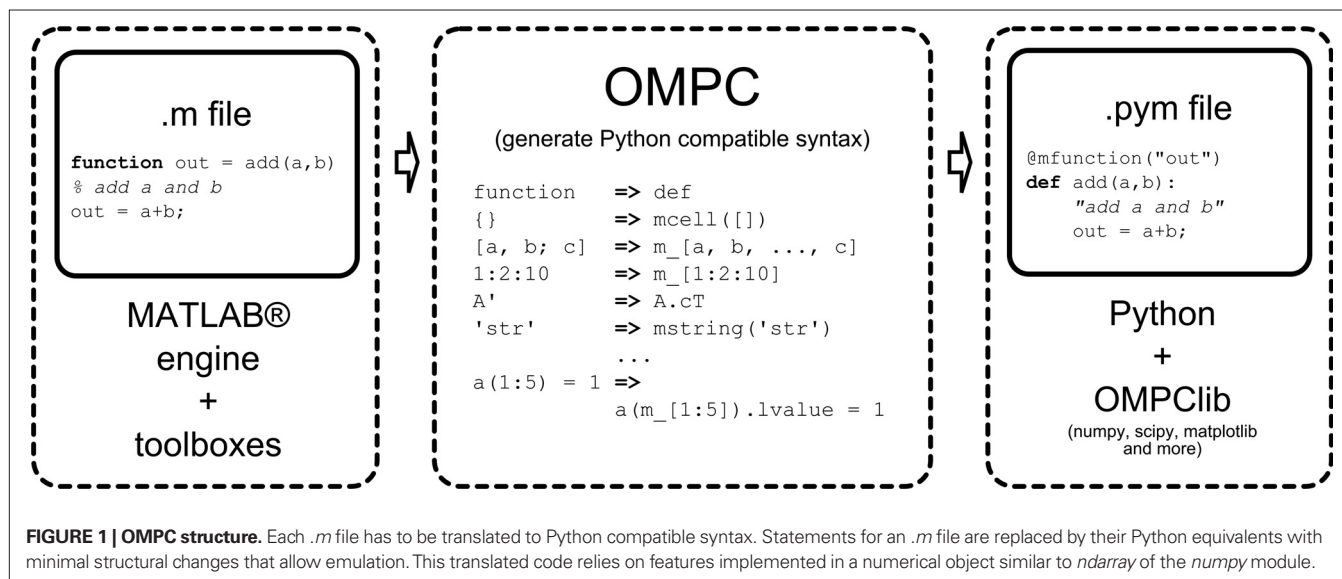
OMPC is complemented by the module *OMPClib*. This module provides implementations of objects that act as proxies of dynamic features specific to MATLAB®.

Note that the mentioned enhancement of functionality is realized without any change to the Python language itself. It is absolutely important not to change the Python language in favor of a single package. Changes to the interpreter should only be made if they are met with general acceptance among the users of the language. Otherwise it would lead to the opposite of the unification aimed for. Moreover, a program translated by OMPC preserves the structure of the original MATLAB® program. The resulting program, in all but three cases (function declaration, switch statement, multiple statements on a single line), corresponds line by line to its MATLAB® source code. An example of equivalent MATLAB® and Python compatible codes can be found in the “Results” section.

THE COMPILER

To use MATLAB® code in Python, an intermediate step of MATLAB®-to-Python *syntax adaptation* is needed. The MATLAB® code must be parsed and translated into Python code that is functionally equivalent to its original. To parse MATLAB® source code we used a free 100% Python implementation of lex and yacc parsing tools called PLY (<http://www.dabeaz.com/ply/>). The compiler is implemented in a single Python file (`examples/ompc/ompcply.py`). This file is a collection of grammar definitions. Each definition is associated with a processing function for a specific language construct (keyword, number, assignment, index access and others). The grammatical rule for each construct is specified in the

¹The following sections contain listings of code in both programming languages. We adhere to the following convention: The mark `>>` at the beginning of a statement signifies a MATLAB® program, while the mark `>>>` signifies Python code. Each of the concepts introduced in the following subsections has a corresponding executable script that is part of the Supplementary Material.



documentation string of its processing function. The functions are designed to cover every syntactically correct MATLAB® language statement. The *PLY module* uses the grammar file to generate a parser, which searches a source text for language constructs and passes these to their corresponding processing functions. The parser produces the translated Python-compatible code. In the case of strings, the syntactical rule is the regular expression `STRING = r'((?:"[^\n']*)|'([^']*'))'` and the processing function looks as follows:

```
def p_expression_string(p):
    "expression : STRING"
    p[0] = "mstring(%s)"%p[1]
```

Every MATLAB® string that passes through this function will be enclosed in the expression `mstring(.)`. Such a string can have all the features of a MATLAB® string. If this is not required, it is possible to replace the last line with `p[0] = p[1]`. As a result the strings from the original will stay intact.

The important advantage of using Python for the translation is that its code is easy to read and can be easily modified. Modifying a Python program does not require installation of a large complicated development system, common for low level languages like C++ or Java. The development advantages outweigh the negligible differences in processing speed.

NUMERICAL LIBRARY

Here we present the additional objects necessary for full compatibility with MATLAB®. The following MATLAB® example illustrates the impossibility of differentiating between variables and functions at translation.

```
>> add = @(a,b) a+b;
>> add(1,2)                % Python -> add(1,2)
ans = 3
>> add = 1:10;
>> add(1,2)
ans = 2                    % Python -> add[0,1]
```

MATLAB® uses the same syntax for calling a function and retrieving elements from an array. This makes it impossible to determine if an identifier *add* in the above listing is a variable or a function. Therefore it is not possible to correctly translate the statement `>> add(1,2)` at compilation time. Our solution is based on the fact that object-oriented programming allows overloading of operators. We therefore have the option to overload the object's `__call__` function. Thus the OMPC code can be executed in Python, behaving equivalently to its MATLAB® original, independently of whether *add* is a function or a variable. Note that this added feature enhances the original numerical array (*numpy* in our examples) without altering its original function. The new object *marray* inherits all functionality from the original numerical array. This object enhanced by an overloaded `__call__` operator allows the following example to run in Python:

```
>>> add = lambda a,b: a+b;
>>> add(1,2)
3
>>> add = mslice[1:10];
>>> add(1,2)
ans = 2.0
```

The supplementary OMPC numerical object is currently based on *numpy*'s array object. This is however not the only option. It is possible to use base objects from another package like *Numarray*, *CVXOPT* (<http://abel.ee.ucla.edu/cvxopt>) or others. For non-numerical objects we can enhance Python built-in types. For example the OMPC string is based on the Python string implementation. The OMPC's cell array object is based on the Python built-in list object, which is equivalent in features to the cell array but, as is obvious from the following example, the performance boost achieved by using the Python list object is considerable.

```
>>> m = {}; tic, for i=1:100000, m{i} = 12; end, toc
Elapsed time is 9.637410 seconds.
```

Python does not allow on-demand growing of lists, but this feature can easily be emulated:

```
>>> class mcellarray(list):
    def __setitem__(self,i,v):
        if i >= len(self):
            self.extend([None]*(i-len(self)) + [v])

>>> m = mcellarray()
>>> tic()
>>> for i in xrange(100000): m[i] = 12
>>> toc()
Elapsed time is 0.372690 seconds.
```

The above example is not the optimal way of using the cell array. Such incorrect use of MATLAB®'s benevolent interpreter is, however, very common. As the last example shows, Python can help to greatly enhance the usability of such sub-optimal code.

ARRAY SLICING, INDEX BASE 1

The first element of a Python sequence type is 0, while MATLAB® uses 1 as the base for indexing, for instance `a[0]` in Python is equivalent to `a(1)` in MATLAB®. OMPC solves this incompatibility by overloading the numerical object's `__call__` method. The same technique of overloading the `__call__` function also makes it possible to use MATLAB® style array slicing. Consider again:

```
>>> b = a(1:10);
```

it is unclear until run-time if *a* is a function accepting a vector or a vector from which we are retrieving the first 10 elements. Python does not allow using a slice object outside of the index `[]` operator. By translating this statement into Python acceptable syntax

```
>>> b = a(mslice[1:10]);
```

and making *a* an object with overloaded `__call__` operation, this code can be executed in Python, behaving equivalently to its MATLAB® original independently of whether *a* is a function or a variable.

The *mslice* proxy object does two things. First it allows a slice object to be used as a parameter to a function call. Secondly it adapts MATLAB® index-base-1 slices from the syntax *start:stop:step* to Python's *start:stop:step*. Python's slice object returns slices up to the stop element, while MATLAB®'s slices range up to the stop element including it.

DYNAMIC UPDATE OF THE VARIABLE NAME SPACE, EMULATION OF *NARGIN/NARGOUT*

Python comes with a built-in module called *inspect*. Using this module it is possible to look into the execution stack to see in what context a function is being executed. This means that at any time a function is called we can look a couple of steps back in history and ask the interpreter about the code from which our function has been called. Consider the following statement:

```
>>> [a, b] = sort(rand(10,1))
```

Python accepts both *a*, *b*, and *[a, b]* (the correct syntax in MATLAB®) as left-value for an assignment. The *inspect* module makes it possible to ask the interpreter for the number of arguments on the left side of the assignment at the moment just before a function was called. The *OMPCLib* module contains a function *_getnargout* that does exactly this. The following Python statement that leaves *nargout* undefined:

```
def f(x):
    if nargout == 2:
        return 1, 2
    else:
        return 1
```

can thus be rewritten to:

```
def f(x):
    nargout = _getnargout()
    if nargout == 2:
        return 1, 2
    else:
        return 1
```

The *mfunction* decorator, which will be discussed in detail in Section “The *mfunction* Decorator”, makes sure that a call to the *_getnargout* function is inserted in the preamble of all functions translated by OMPC. This means that the original MATLAB® function body again can stay intact; we only need to apply the *mfunction* decorator that inserts *nargout* and, similarly, *nargin* into the variable namespace of the function during runtime.

ASSIGNMENTS TO NOVEL VARIABLES, ASSIGNMENTS TO SLICES

We explained that it is possible to use the *__call__* function to allow MATLAB®-style array slicing. There is one exception, however: Python does not allow function calls to be used for assignment. We circumvent this restriction by assigning to a property of the slice. The property mediates the assignment operation and makes the syntax acceptable to the Python parser. For instance,

```
>>> a(1) = 1          # Syntax error
```

is not allowed, but the following is:

```
>>> a(1).lvalue = 1
```

MATLAB® allows assignment to slices of variables that were not previously initialized. The module *inspect* allows us to detect assignment to non-existent variables. In the translated code, the variables are initialized during runtime by the *mfunction* decorator (see The *mfunction* Decorator).

ELEMENT-WISE OPERATIONS

MATLAB® offers a convenient way of differentiating between operations for matrices and their element-wise equivalents. Although such a differentiation was repeatedly proposed for Python (Zhu and Liens, 2000) it never gained enough support from the broader Python community. In *numpy*, all numerical operations on arrays are element-wise by default. In principle, it would not have been a problem to use function calls to differentiate between these and matrix operations, for instance:

```
a .* b => multiply(a, b)    and    a * b => dot(a, b)
```

However in accordance with our principle to preserve as much as possible the original structure of the MATLAB® code, we suggest another solution. This solution is inspired by a recipe from the community-driven Python cookbook (<http://code.activestate.com/recipes/384122/>). Python allows overriding of operators on either side of an operand. This feature is commonly used to enable automatic coercion of types. For example, it allows the user to apply an arithmetic operation between a *numpy* array and anything else. So, for adding to array *x* a list *[1,2]*, instead of having to convert it to an array: *x + array([1,2])*, we can simply write: *x + [1,2]*. Therefore it is possible to change the above translation rule as follows:

```
a .* b => a * elmul* b    and    a * b => a * b
```

The *elmul* is an instance of an object that has overloaded the *** operator (the *__mul__* and *__rmul__* function). Independently of the execution order of the operations in the statement, the *elmul* object remembers the operand from the first multiplication and instructs the second operand to perform element-wise multiplication (*a*elmul -> elmul.left = a*, *elmul*b -> elmul.left*b*).

IMPLIED RETURNS

MATLAB® uses implied returns; the “return” statement without parameters serves only for breaking the execution of a function. The return parameters of a function are specified in the function declaration. Python requires specification of these variables at each point of exit from the function. Python’s return statement consists of a list of variables to be returned from a function call. Absence of the list means the empty object *None* is returned.

```
function [mi,ma] = minmax(a)
mi = min(a);
if nargout > 1, ma = max(a); end

@mfunction("mi, ma")
def minmax(a=None)
    mi = min(a)
    if nargout > 1: ma = max(a)
```

In the above example it is not possible to simply append a return statement *return mi, ma*. Because its value is being assigned to a single object (*mi*), the *minmax* function is expecting to return a single value. Python would therefore automatically assign a sequence, or tuple, containing both return values to the single variable at the output of the function call. This is illustrated in the following:

```
>>> mi = minmax(rand(1,10));
ans = (0.0574, None)
```

It would, in principle, be possible to add a statement `return (mi, ma)[:nargout]` in all locations where function exit could occur. This strategy would already rely on the introspection function to determine the value of `nargout`. However, adding such statements is cumbersome and destroys the structure of the original syntax. Introspection allows us to preserve the structure by automatically modifying the bytecode of translated functions, inserting the equivalent code wherever needed. This and other previously mentioned modifications to the bytecode are handled by the `mfunction` decorator.

THE MFUNCTION DECORATOR

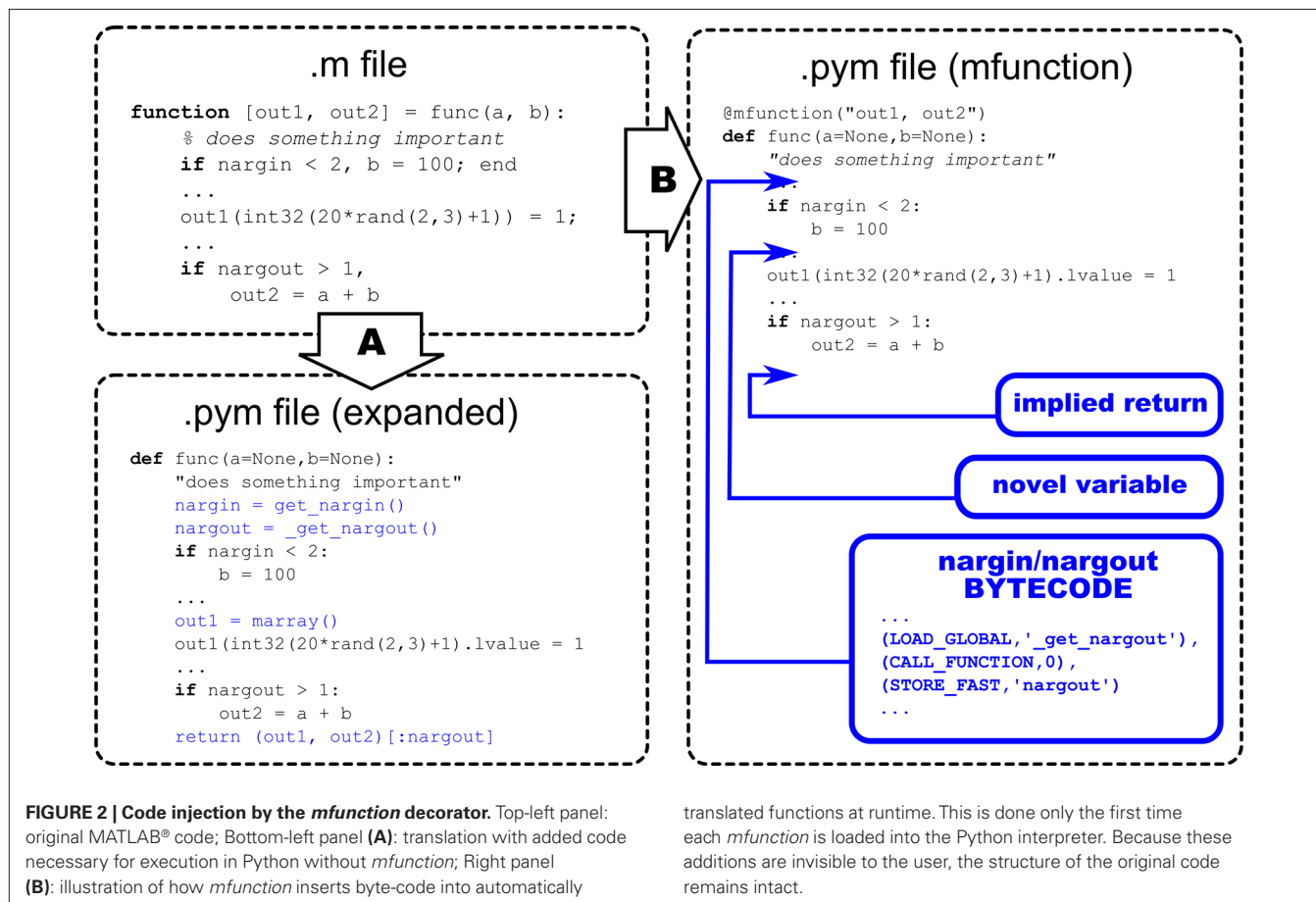
Python offers the feature of decorators since version 2.4. Simply put, decorators are function factories. They allow us to turn a regular Python function into one that behaves like a MATLAB® function. A Python decorator receives a function just before it is loaded into the current workspace. The decorator can manipulate the function in arbitrary ways. The `mfunction` decorator modifies each function translated by OMPC. We use the decorator to emulate the existence of the variables `nargin/nargout`, to allow assignments to novel variables, and to implement implied returns (Figure 2).

This modification of byte-code happens at run-time. It happens only once when the interpreter loads a function, not every time the function is called. The performance of the decorated function does not differ from the performance of a function where modifications are stated explicitly in the source code.

OMPC EXTENSIONS

Here we deal with the issue of how OMPC handles C/C++ and FORTRAN (MEX) extensions for MATLAB®. Both MATLAB® and Python allow extensions and both have an official protocol for writing them. However, the interface between platform and extension differs considerably between the two respective languages. Extensions written for MATLAB®, therefore, do not work in Python. We can solve this problem by implementing a C support library that allows compilation of extensions independently of MATLAB. Compilation turns these routines into dynamic-link libraries that can be called by any language, including Python. The Supplementary Material has an example that shows how the `mxCreateDoubleMatrix` function can be implemented for example, using the Standard Template Library of C++.

In general it is very easy in Python to wrap external libraries by using the open-source application GCCXML (<http://www.gccxml.org/>). The Python community extensively uses this application for automatically generating Python extensions for libraries with complex structure and large numbers of exported symbols. The advantage of GCCXML over tools like Cython or Pyrex (<http://cython.org/>) and the multipurpose Swig (<http://www.swig.org/>) is that it is based on a production-stable GCC compiler. This means that any large project that relies on the latest features of C++, including the use of templates, can be automatically correctly parsed and analyzed to be further processed to



generate extensions (www.boost.org/doc/libs/release/libs/python/doc/, <http://pypi.python.org/pypi/ctypeslib/> and many others).

RESULTS

A website has been created for the project, <http://ompc.juricap.com/>. The compiler is also available on-line at <http://omplib.appspot.com/>. This site will serve as a bug-tracking utility that will allow users to submit files that are not correctly processed by OMPC.

Because the formal specification of the MATLAB® syntax is not publicly available, it is difficult to properly test the OMPC compiler. However, we have successfully translated *m-files* that are part of the standard MATLAB® distribution. In addition, the compiler was tested successfully using source code collected from a number of users within the RIKEN Brain Science Institute and outside collaborators. The styling of MATLAB® source code varied significantly from person to person.

The following example consists of original source code, contained in online Supplementary Material to a neuroscience publication (Izhikevich, 2003). The example shows the original MATLAB® *m-file* and its fully automatic translation by OMPC.

```
% Created by Eugene M. Izhikevich, February 25, 2003
% Excitatory neurons   Inhibitory neurons
Ne=800;                Ni=200;
re=rand(Ne,1);         ri=rand(Ni,1);
a=[0.02*ones(Ne,1);    0.02+0.08*ri];
b=[0.2*ones(Ne,1);     0.25-0.05*ri];
c=[-65+15*re.^2;       -65*ones(Ni,1)];
d=[8-6*re.^2;          2*ones(Ni,1)];
S=[0.5*rand(Ne+Ni,Ne),-rand(Ne+Ni,Ni)];

v=-65*ones(Ne+Ni,1); % Initial values of v
u=b.*v;              % Initial values of u
firings=[];           % spike timings

for t=1:1000          % simulation of 1000 ms
    I=[5*randn(Ne,1);2*randn(Ni,1)]; % thalamic input
    fired=find(v>=30); % indices of spikes
    if ~isempty(fired)
        firings=[firings; t+0*fired, fired];
        v(fired)=c(fired);
        u(fired)=u(fired)+d(fired);
        I=I+sum(S(:,fired),2);
    end;
    v=v+0.5*(0.04*v.^2+5*v+140-u+I);
    v=v+0.5*(0.04*v.^2+5*v+140-u+I);
    u=u+a.*(b.*v-u);
end;
plot(firings(:,1),firings(:,2),'.');
```

The OMPC equivalent is:

```
# Created by Eugene M. Izhikevich, February 25, 2003
# Excitatory neurons   Inhibitory neurons
Ne = 800
Ni = 200;

re = rand(Ne, 1)
ri = rand(Ni, 1);
```

```
a = mcat([0.02 * ones(Ne, 1),
          OMPSEMI, 0.02 + 0.08 * ri])
b = mcat([0.2 * ones(Ne, 1),
          OMPSEMI, 0.25 - 0.05 * ri])
c = mcat([-65 + 15 * re **elpow** 2,
          OMPSEMI, -65 * ones(Ni, 1)])
d = mcat([8 - 6 * re **elpow** 2,
          OMPSEMI, 2 * ones(Ni, 1)])
S = mcat([0.5 * rand(Ne + Ni, Ne), -rand(Ne + Ni, Ni)])

v = -65 * ones(Ne + Ni, 1)      # Initial values of v
u = b * elmul * v               # Initial values of u
firings = mcat([])              # spike timings

for t in mslice[1:1000]:        # simulation of 1000 ms
    I = mcat([5 * randn(Ne, 1), OMPSEMI,
              2 * randn(Ni, 1)]) # thalamic input
    fired = find(v >= 30)        # indices of spikes
    if not isempty(fired):
        firings = mcat([firings, OMPSEMI,
                        t + 0 * fired, fired])
        v(fired).lvalue = c(fired)
        u(fired).lvalue = u(fired) + d(fired)
        I = I + sum(S(mslice[:, fired], 2)
                    end
    v = v + 0.5 * (0.04 * v **elpow** 2 + 5 *
                  v + 140 - u + I)
    v = v + 0.5 * (0.04 * v **elpow** 2 + 5 *
                  v + 140 - u + I)
    u = u + a * elmul * (b * elmul * v - u)
end
plot(firings(mslice[:, 1]), firings(mslice[:, 2]),
     mstring('.')')
```

In this example we observe how well the translation preserves the structure of the original MATLAB® program. The above OMPC code is generated using rules that result in maximum compatibility. For example the last line contains the Python object *mstring*('.') that emulates the MATLAB® string object. As a consequence, the string is modifiable, as in the original. Since this is not necessary in the context of this program, a simple Python string could be used instead, as explained in Section “The Compiler”. It is possible to further simplify the syntax by syntactical shortcuts, so called *index tricks* (*r_*, *c_*, *mgrid*), that are already part of the *numpy* library (Oliphant, 2006). The *plot* statement of the last program could therefore be simplified to, for example:

```
plot(firings(m_[:, 1]), firings(m_[:, 2]), '.')
```

The structural equivalence of both programs was made possible by using the introspection functionality of Python. Some of the dynamical features, however, can equally well be resolved by the OMPC compiler, provided that we are willing to compromise on structural equivalence. This would enhance the clarity of code for Python developers not familiar with implied variables of MATLAB®. Only adopting and testing OMPC will allow the users to make the correct decision. The final form of code generated by OMPC has still to be agreed upon. Future developments of the compiler will enable such options through switches.

In the Supplementary Material to this paper, we provide OMPC executables of the spiking neuron model described in (Izhikevich,

2003). At the moment of writing, two versions are available. One is based on the *ndarray* numerical array of the *numpy* library. However, optimized numerical packages such as *numpy* are not available yet for the newest Python interpreters. The other version, therefore, shows a pure Python implementation of an *n*-dimensional numerical array. This version is significantly slower for operations on large arrays but, because it runs on a clean Python installation, can be run on other realizations of Python as well; we have successfully tested this for Jython 2.5a1, Python 2.6 and 3.0. The standard Python modules *array*, *random* and *math* are at the core of this second version; any Python interpreter sufficiently developed to contain these modules will execute the model. Maintaining a pure Python version of OMPC could enable acceleration of OMPC modules using PyPy (<http://codespeak.net/pypy/>, Rigo and Pedroni, 2006) or Shedskin (An Optimizing Python-to-C++ compiler, <http://shedskin.blogspot.com/>).

DISCUSSION

A number of different implementations of Python are currently available. We choose CPython because it is the primary Python engine; the most mature and stable implementation. All numerical extensions were originally developed for CPython. CPython, moreover, offers by default the *ctypes* module, which is of crucial importance as a support library for OMPC. CPython allows easy and efficient access to extension modules written in C/C++, FORTRAN and many other languages that allow us to create dynamic-link libraries.

Amongst forthcoming Python implementations that may influence the future development of OMPC, the most interesting one is PyPy. PyPy is an implementation of Python in Python itself and supports compilation of a restricted subset called RPython (Restricted Python, <http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html#rpython>, Section 1.4) into the C language and from there on into native binary executables. Although this possibility has not been tested, if PyPy will support specific CPython features it should be possible to compile OMPC generated files to native executables.

OMPC aims ultimately to offer full compatibility with the syntax and the engine of MATLAB®. A number of its features, however, have not yet been addressed in this article. The most sought after ones relate to its GUI components. Implementing these is practicable, based on the fact that the MATLAB® application GUI-designer stores its information in “.fig” files, which are actually .mat data files. This means that they can be loaded into Python using OMPC, enabled through the *scipy.io* module. These files hold enough information to identify and reconstruct the GUI components within a figure.

There is currently no plan to implement embedded Java, because we consider it not to be a crucial part of MATLAB®. While Java can be useful in MATLAB®, for example, for networking applications, the verbosity and complexity of Java are a great obstacle to use for anybody without a professional software engineering background. Moreover, all features that Java offers as an enhancement of MATLAB® are, most likely, present in Python as well. For networking purposes, therefore, Python is a much more suitable extension than Java for a high level language such as MATLAB®. Python includes support for networking by

default. It contains modules with ready-to-use implementations of client-server applications. A good example is the OMPC on-line compiler currently hosted as a Python service at <http://ompc.lib.appspot.com/>.

In a broader scope, one of the great advantages of being able to parse source code is that it allows analysis and possible optimization of the code that will be executed. This is the approach taken by platforms based on virtual machines like .NET, Java and LLVM. Source code that can be parsed and translated into an intermediate format (CIL, formerly known as MSIL, Java Bytecode, or LLVM IR) can be run or translated to another low-level language including machine code. PyPy uses this technique to translate a sufficiently static subset of Python into C (Rigo and Pedroni, 2006). OMPC is an example of how to use Python byte-code as an intermediate representation.

Choosing Python as a platform for technical computation offers a number of additional benefits. As a popular general-purpose language, Python offers up-to-date facilities for online sharing, and enhancing the visibility of projects, in which computational methods are naturally embedded. The online OMPC compiler included in the Supplementary Material is one example of such an application. Python is currently one of the most popular tools in server-side Web 2.0 development.

The introduction mentions a number of attempts to provide MATLAB® functionality in Python. Currently there is only one actively developed project MlabWrap (<http://mlabwrap.sourceforge.net/>) that allows the use of MATLAB® functions along with the numerical extensions of Python. This project embeds the MATLAB® engine in a Python extension. This extension however requires a licensed copy of MATLAB®. A similar approach could be taken with the open-source library *liboctave* that is at the core of the GNU Octave (<http://www.gnu.org/software/octave/>). The design of OMPC allows any implementation of OMPClib to be used for execution of the OMPC generated Python code. An OMPClib could be built with *liboctave*'s Array class as its base numerical object. The advantage of wrapping a library instead of embedding an interpreter is the great simplification of memory management. Embedding a interpreter in an extension is very similar to running a second process of which the data in memory are not directly accessible to Python and another extensions.

The interest of the scientific community in the Python language is growing (Langtang, 2006, <http://www.scipy.org/>, <http://www.neuralensemble.org/>), making it ever more likely that it will become the main open-source language of scientific computation. One of the important obstacles in this transition is the large amount of legacy code written in MATLAB®. A fully automatic translation system could enable the reuse of large projects, the size of which makes human translation infeasible. By presenting OMPC, we demonstrated that Python could adopt MATLAB® code for reuse; without human intervention this code can be translated into Python. OMPC does this in a manner that, whenever possible, preserves the structure of the original. The syntax and design of MATLAB® language proved to be easy for beginners. In MATLAB® every object is also a multi-dimensional array, even a number is a 1×1 matrix. Python users however face the challenge of understanding concepts such as different types (numbers and arrays) and others common in programming, for example object reference. A

number of MATLAB® inspired features could help removing many obstacles for a user introduced to Python's numerical facilities. We discussed such features and their implementation in OMPC. By providing automatic translation of MATLAB® code to Python and the enhanced ease of use, OMPC will promote Python as the open-source alternative for scientific computation. To the Python community, OMPC offers this bridge as an incentive towards the further enhancement of numerical computation capabilities.

REFERENCES

- Ascher, D., Dubois, P. E., Hinsien, K., Hugunin, J., and Oliphant, T. (2001). Numerical Python, Technical Report UCL-MA-128569, Lawrence Livermore National Laboratory. Available at: <http://numpy.scipy.org>.
- Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering, Lecture Notes in Computer Science, Vol. 3039/2004. Berlin/Heidelberg, Springer, pp. 1210–1217.
- Izhikevich, E. M. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572.
- Langtangen, H. P. (2006). Python Scripting for Computational Science. Basel, Birkhäuser.
- Moler, C., The Creator of MATLAB (2004). The Origins of MATLAB. Available at: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/dec04.html.
- Moler, C. (2006). The Growth of MATLAB and The MathWorks over Two Decades. Available at: http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.
- Oliphant, T. E. (2006). Guide to NumPy. Trelgol Publishing, Spanish Fork, UT. Available at: <http://numpy.scipy.org>.
- Oliphant, T. E., (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20.
- Oliphant, T. E., and Banks, C. (2006). Index of Python Enhancement Proposals (PEPs), PEP 3118: Revising the Buffer Protocol. Available at: <http://www.python.org/dev/peps/pep-3118/>.
- Rigo, A., and Pedroni, S. (2006). PyPy's Approach to Virtual Machine Construction, Dynamic Languages Symposium at OOPSLA. Available at: <http://codespeak.net/svn/pppy/extradoc/talk/dls2006/pppy-vm-construction.pdf>.
- Zhu, H., and Lielens, G. (2000). Index of Python Enhancement Proposals (PEPs), PEP 225: Elementwise/Objectwise Operators. Available at: <http://www.python.org/dev/peps/pep-0225/>.
- could be construed as a potential conflict of interest.

ACKNOWLEDGMENTS AND REMARKS

MATLAB® is a registered trademark of The MathWorks, Inc. “Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation.

SUPPLEMENTARY MATERIAL

The Supplemental Data for this article can be found online at <http://ompc.juricap.com/>.

Received: 14 September 2008; paper pending published: 13 October 2008; accepted: 30 January 2009; published online: 10 February 2009.

Citation: Jurica P and van Leeuwen C (2009) OMPC: an open-source MATLAB®-to-Python compiler. Front. Neuroinform. (2009) 3:5. doi: 10.3389/neuro.11.005.2009

Copyright © 2009 Jurica and van Leeuwen. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



PyMVPA: a unifying approach to the analysis of neuroscientific data

Michael Hanke^{1,2†}, Yaroslav O. Halchenko^{3,4,5†}, Per B. Sederberg^{6,7}, Emanuele Olivetti^{8,9}, Ingo Fründ^{1,10,11}, Jochem W. Rieger^{2,12,13}, Christoph S. Herrmann^{1,11,13}, James V. Haxby^{14,15}, Stephen José Hanson^{3,5} and Stefan Pollmann^{1,2,13 *}

¹ Department of Psychology, University of Magdeburg, Magdeburg, Germany

² Center for Advanced Imaging, Magdeburg, Germany

³ Psychology Department, Rutgers Newark, New Jersey, USA

⁴ Computer Science Department, New Jersey Institute of Technology, Newark, New Jersey, USA

⁵ Rutgers University Mind Brain Analysis, Rutgers Newark, New Jersey, USA

⁶ Department of Psychology, Princeton University, Princeton, New Jersey, USA

⁷ Princeton Neuroscience Institute, Princeton University, Princeton, New Jersey, USA

⁸ Center for Information Technology (Irtst), Fondazione Bruno Kessler, Trento, Italy

⁹ Center for Mind/Brain Sciences (CIMeC/NI Lab), University of Trento, Italy

¹⁰ Leibniz Institute for Neurobiology, Magdeburg, Germany

¹¹ Bernstein Group for Computational Neuroscience, Magdeburg, Germany

¹² Department of Neurology, University of Magdeburg, Magdeburg, Germany

¹³ Center for Behavioral Brain Sciences, Magdeburg, Germany

¹⁴ Center for Cognitive Neuroscience, Dartmouth College, Hanover, New Hampshire, USA

¹⁵ Department of Psychological and Brain Sciences, Dartmouth College, Hanover, New Hampshire, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Martin A. Spacek, The University of
British Columbia, Canada
Samuel Garcia, Université Claude
Bernard Lyon I, France

*Correspondence:

Stefan Pollmann, Institut für
Psychologie II, Otto-von-Guericke-
Universität Magdeburg, PF 4120,
D-39016 Magdeburg, Germany.
e-mail: stefan.pollmann@ovgu.de

[†]Hanke and Halchenko contributed
equally to this article.

The Python programming language is steadily increasing in popularity as the language of choice for scientific computing. The ability of this scripting environment to access a huge code base in various languages, combined with its syntactical simplicity, make it the ideal tool for implementing and sharing ideas among scientists from numerous fields and with heterogeneous methodological backgrounds. The recent rise of reciprocal interest between the machine learning (ML) and neuroscience communities is an example of the desire for an inter-disciplinary transfer of computational methods that can benefit from a Python-based framework. For many years, a large fraction of both research communities have addressed, almost independently, very high-dimensional problems with almost completely non-overlapping methods. However, a number of recently published studies that applied ML methods to neuroscience research questions attracted a lot of attention from researchers from both fields, as well as the general public, and showed that this approach can provide novel and fruitful insights into the functioning of the brain. In this article we show how *PyMVPA*, a specialized Python framework for machine learning based data analysis, can help to facilitate this inter-disciplinary technology transfer by providing a single interface to a wide array of machine learning libraries and neural data-processing methods. We demonstrate the general applicability and power of *PyMVPA* via analyses of a number of neural data modalities, including fMRI, EEG, MEG, and extracellular recordings.

Keywords: functional magnetic resonance imaging, electroencephalography, magnetoencephalography, extracellular recordings, machine learning, Python

INTRODUCTION

Understanding how the brain is able to give rise to complex behavior has stimulated a plethora of brain measures such as non-invasive EEG¹, MEG², MRI³, PET⁴, optical imaging, and invasive extracellular and intracellular recordings, often in conjunction with new methods, models, and techniques. Each data acquisition method has offered a unique set of properties in terms of spatio-temporal resolution, signal to noise, data acquisition cost,

applicability to humans, and the corresponding neural correlates that result from the measurement process.

Neuroscientists often focus on only one or a smaller subset of these neural modalities partly due to the kinds of questions investigated and partly due to the cost of learning to analyze data from these different modalities. The diverse measurement approaches to brain function can heavily influence the selection of a research question and, in turn, the development of specific software packages to answer them. Consequently, the peculiarities of each data acquisition modality and the lack of strong interaction between the neuroscience communities employing them have produced distinct software packages specialized for the conventional analyses within a particular modality. Some analysis techniques have

¹Electroencephalography.

²Magnetoencephalography.

³Magnetic resonance imaging.

⁴Positron emission tomography.

become, due to normative concerns, *de facto* standards despite their limitations and inappropriate assumptions for the given data type. For instance, the general linear model (GLM) is the prevalent approach used in fMRI data analysis, despite being a restrictive mass-univariate method (Kriegeskorte and Bandettini, 2007; O'Toole et al., 2007).

While specialized software packages are useful when dealing with the specific properties of a single data modality, they limit the flexibility to transfer newly developed analysis techniques to other fields of neuroscience. This issue is compounded by the closed-source, or restrictive licensing of many software packages, which further limits software flexibility and extensibility.

However, outside the neuroscience community, machine learning (ML) research has spawned a set of analysis techniques that are typically generic, flexible (e.g., classification, regression, clustering), powerful (e.g., multivariate, linear and non-linear) and often applicable to various data modalities with minor modality-specific pre-processing (see Pereira et al., in press, for a tutorial on application of ML methods to the analysis of fMRI data). Moreover, large parts of this community favor the open-source software development model (Sonnenburg et al., 2007, see also *MLOSS*⁵ project website), which leads to an increase in scientific progress due to the superior accessibility of information and reproducibility of scientific results. These advantages have recently attracted considerable interest throughout the neuroscience community (see Haynes and Rees, 2006; Norman et al., 2006, for reviews).

Nevertheless, various factors have delayed the adoption of these newer methods for the analysis of neural information. First and foremost, existing conventional techniques are well-tested and often perfectly suitable for the standard analysis of data from the modality for which they were designed. Most importantly, however, a set of sophisticated software packages has evolved over time that allow researchers to apply these conventional and modality-specific methods without requiring in-depth knowledge about low-level programming languages or underlying numerical methods. In fact, most of these packages come with convenient graphical and command line interfaces that abstract the peculiarities of the methods and allow researchers to focus on designing experiments and to address actual research questions without having to develop specialized analyses for each study.

However, only a few software packages exist that are specifically tailored towards straightforward and interactive exploration of neuroscientific data using a broad range of ML techniques, such as the Matlab[®] MVPA toolbox for fMRI data⁷ (Detre et al., 2006). At present only independent component analysis (ICA), an unsupervised method, seems to be supported by numerous software packages (see Beckmann and Smith, 2005, for fMRI, and Makeig et al., 2004, for EEG data analysis). Therefore, the application of machine learning analyses, referred to in the literature as *decoding* (Haynes et al., 2007; Kamitani and Tong, 2005), *information-based analysis* (Kriegeskorte et al., 2006) or *multi-voxel pattern analysis* (Norman et al., 2006), usually involves the development of a significant amount of custom code. Hence, users are typically required

to have in-depth knowledge about both data modality peculiarities and software implementation details.

At the same time, Python has become the open-source scripting language of choice in the research community to prototype and carry out scientific data analyses or to develop complete software solutions quickly. It has attracted attention due to its openness, flexibility, and the availability of a constantly evolving set of tools for the analysis of many types of data. Python's automatic memory management, in conjunction with its powerful libraries for efficient computation (*NumPy*⁸ and *SciPy*⁹) abstracts users from low-level "software engineering" tasks and allows them to fully concentrate their attention on the development of computational methods.

As an interpreted, high-level scripting language with a simple and consistent syntax, a plethora of available modules, easy ways to interface to low-level libraries written in other languages¹⁰ and high-level computing environments¹¹, Python is the language of choice for solving many scientific computing problems. **Table 1** lists a number of Python modules which might be of interest in the neuroscientific context, and is meant to complement the material presented in the other articles in this special issue.

Despite the fact that it is possible to perform complex data analyses solely within Python, it *once again* often requires in-depth knowledge of numerous Python modules, as well as the development of a large amount of code to lay the foundation for one's work. Therefore, it would be of great value to have a framework that helps to abstract from both data modality specifics and the implementation details of a particular analysis method. Ideally, such a framework should help to expose any form of data in an optimal format applicable to a broad range of machine learning methods, and on the other hand provide a versatile, yet simple, interface to plug in additional algorithms operating on the data. In the neuroscience context it would also be useful to bridge between well-established neuroimaging tools and ML software packages by providing cross library integration and transparent data handling for typical containers of neuroimaging data (e.g., NIfTI in fMRI research).

As an attempt to provide such a framework we have implemented PyMVPA¹² (MultiVariate Pattern Analysis in Python) – a free and open-source Python framework to facilitate uniform analysis of the neural information obtained from different neural modalities. PyMVPA heavily utilizes Python's ability to access libraries written in a large variety of programming languages and computing environments to interface with the wealth of existing machine learning packages developed outside the neuroscience community. Although the framework is eminently suited for neuroscientific datasets, it is by no means limited to this field. However, the neuroscience tuning is a unique aspect of PyMVPA in comparison to other Python-based ML or computing toolboxes, such as *MDP*¹³ or *scipy-cluster*¹⁴ which are developed as domain-neutral packages.

⁸<http://numpy.scipy.org>.

⁹<http://www.scipy.org>.

¹⁰e.g., ctypes, SWIG, SIP, Cython.

¹¹e.g., mlabwrap and RPy.

¹²<http://www.pymvpa.org>.

¹³<http://mdp-toolkit.sourceforge.net>.

¹⁴<http://code.google.com/p/scipy-cluster/>.

⁵<http://www.mloss.org>.

⁶Closed source commercial product of MathWorks[®].

⁷It is possible to use the low-level functions of this toolbox for other modalities.

Table 1 | Various free and open-source projects, either written in Python or providing Python bindings, which are germane to acquiring or processing neural information datasets using machine learning (ML) methods. The last column indicates whether PyMVPA internally uses a particular project or provides public interfaces to it.

Name	Description	URL	PyMVPA
MACHINE LEARNING			
Elephant	Multi-purpose library for ML	http://elefant.developer.nicta.com.au	
Shogun	Comprehensive ML toolbox	http://www.shogun-toolbox.org	✓
Orange	General-purpose data mining	http://www.ailab.si/orange	
PyML	ML in Python	http://pyml.sourceforge.net	
MDP	Modular data processing	http://mdp-toolkit.sourceforge.net	✓
hcluster	Agglomerative clustering	http://code.google.com/p/scipy-cluster	✓
–	Other Python modules	http://www.mloss.org/software/language/python	
NEUROSCIENCE RELATED			
NiPy	Neuroimaging data analysis	http://neuroimaging.scipy.org	
PyMGH	Access FreeSurfers.mghfiles	http://code.google.com/p/pyfsio	
PyNifti	Access Nifti/Analyzefiles	http://niftilib.sourceforge.net/pynifti	✓
OpenMEEG	EEG/MEG inverse problems	http://www-sop.inria.fr/odyssee/software/OpenMEEG	
STIMULI AND EXPERIMENT DESIGN			
PyEPL	Create complete experiments	http://pyepl.sourceforge.net	
VisionEgg	Visual stimuli generation	http://www.visionegg.org	
PsychoPy	Create psychophysical stimuli	http://www.psychopy.org	
PIL	Python Imaging Library	http://www.pythonware.com/products/pil	
INTERFACES TO OTHER COMPUTING ENVIRONMENTS			
RPy	Interface to R	http://rpy.sourceforge.net	✓
mlabwrap	Interface to Matlab	http://mlabwrap.sourceforge.net	
GENERIC			
Matplotlib	2D Plotting	http://matplotlib.sourceforge.net	✓
Mayavi2	Interactive 3D visualization	http://code.enthought.com/projects/mayavi	
PyExcelerator	Access MS Excel files	http://sourceforge.net/projects/pyexcelerator	
pywavelets	Discrete wavelet transforms	http://www.pybytes.com/pywavelets	✓

The following section provides a short summary of the principal design concepts, and the basic building blocks of the PyMVPA framework. The main focus of this article is, however, a demonstration of PyMVPA's flexibility by applying various ML techniques to typical EEG, MEG, fMRI and extracellular recordings datasets.

PyMVPA

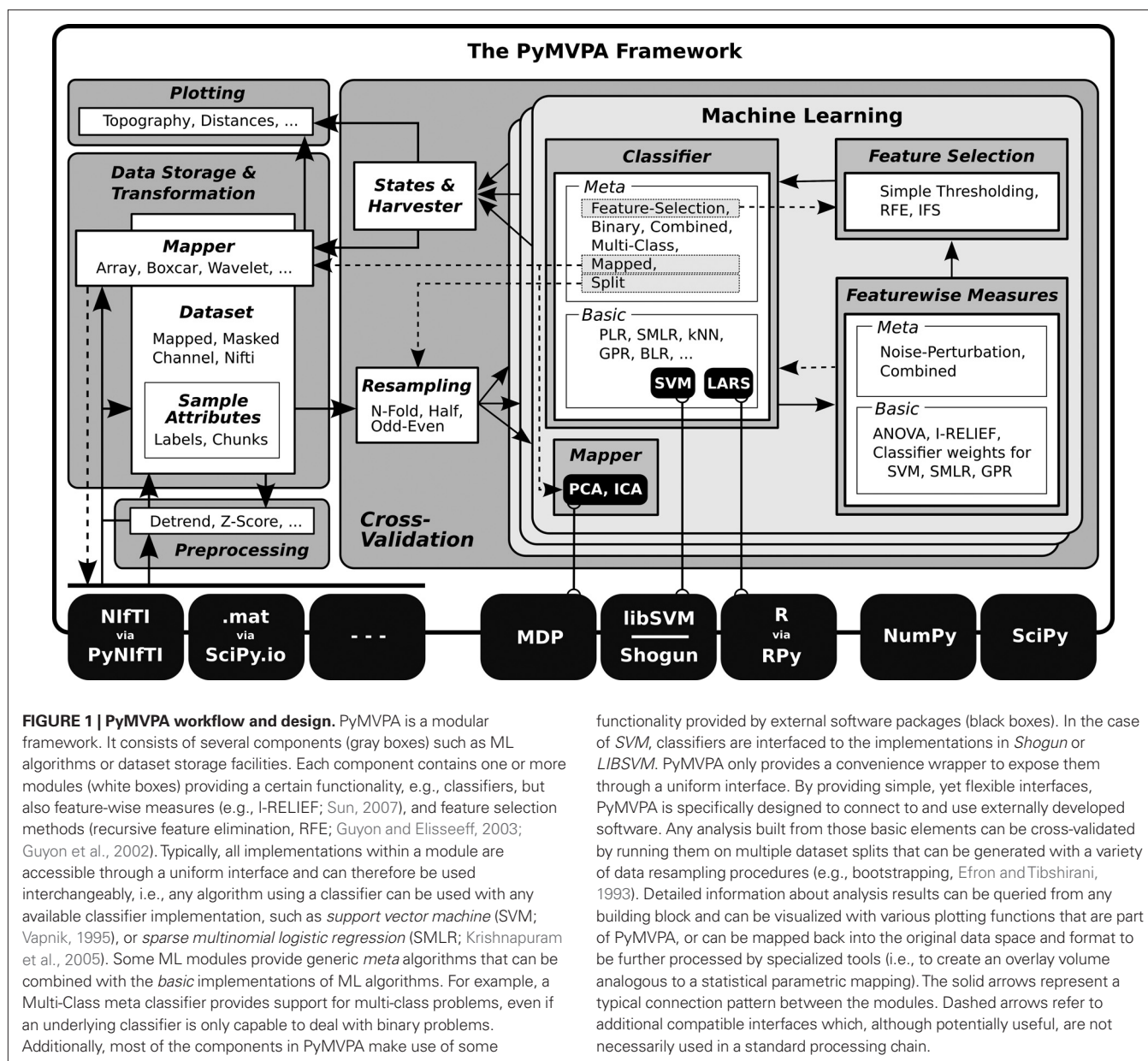
One of the main goals of PyMVPA is to reduce the gap between the neuroscience and ML communities. To reach this goal, we designed PyMVPA to provide a convenient, easy to use, community developed (free and open source¹⁵), and extensible framework to facilitate the use of ML techniques on neural information. PyMVPA combines Python data processing, visualization, and basic I/O facilities together with I/O code and examples tailored for neuroscience. For an easy start into PyMVPA a fMRI example dataset (a single subject from the study by Haxby et al., 2001) is available for download from the PyMVPA website.

¹⁵PyMVPA is distributed under an MIT license, which complies with both Free Software and Open Source definitions.

As **Table 1** highlighted, PyMVPA is not the only ML framework available for scripting and interactive data exploration in Python. In contrast to some of the primarily GUI-based ML toolboxes (e.g., Orange, Elephant), PyMVPA is designed to provide not just a toolbox, but a framework for concise, yet intuitive, scripting of possibly complex analysis pipelines. To achieve this goal, PyMVPA provides a number of building blocks that can be combined in a very flexible way. **Figure 1** shows a schematic representation of the framework design, its building blocks and how they can be combined into complete analysis pipelines.

This article does not aim to provide a detailed description of the PyMVPA framework, and therefore only a rough overview about the most important technical aspects is presented here. However, a comprehensive introduction is available in Hanke et al. (2009) and the PyMVPA manual (Hanke et al., 2008).

In PyMVPA, each building block (e.g., all classifiers) follows a simple, standardized, interface. This allows one to use various types of classifiers interchangeably, without additional changes in the source code, and makes it easy to test the performance of newly developed algorithms on one of the many didactical neuroscience-related examples and datasets that are included in PyMVPA.



In addition, any implementation of an analysis method/algorithm benefits from the basic *house-keeping* functionality done by the base classes, reducing the necessary amount of code needed to contribute a new fully-functional algorithm. PyMVPA takes care of hiding implementation-specific details, such as a classifier algorithm provided by an external C++ library. At the same time it tries to expose all available information (e.g., classifier training performance) through a consistent interface (for reference, this interface is called *states* in PyMVPA).

PyMVPA makes use of a number of external software packages, including other Python modules and low-level libraries (e.g., LIBSVM¹⁶) and computing environments (e.g., R¹⁷). Using

externally developed software instead of reimplementing algorithms has the advantage of a larger developer and user base and makes it more likely to find and fix bugs in a software package to ensure a high level of quality. However, using external software also carries the risk of breaking functionality when any of the external dependencies break. To address this problem PyMVPA utilizes an automatic testing framework performing various types of tests ranging from unittests (currently covering 84% of all lines of code) to sample code snippet tests in the manual and the source code documentation itself to more evolved “real-life” examples. This facility allows one to test the framework within a variety of specific settings, such as the unique combination of program and library versions found on a particular user machine.

At the same time, the testing framework also significantly eases the inclusion of code by a novel contributor by catching errors that

¹⁶<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>.

¹⁷<http://www.r-project.org>.

would potentially break the project's functionality. Being open-source does not always mean *easy to contribute* due to various factors such as a complicated application programming interface (API) coupled with undocumented source code and unpredictable outcomes from any code modifications (bug fixes, optimizations, improvements). PyMVPA welcomes contributions, and thus, addresses all the previously mentioned points:

Accessibility of source code and documentation: All the source code (including website and examples) together with the full development history is publicly available via a distributed version control system¹⁸ which makes it very easy to track the development of the project, as well as to develop independently and to submit back into the project.

Inplace code documentation: Large parts of the source code are well documented using reStructuredText¹⁹, a lightweight markup language that is highly readable in source format as well as being suitable for automatic conversion into HTML or PDF reference documentation. In fact, *Ohloh.net*²⁰ source code analysis judges PyMVPA as having “extremely well-commented source code.”

Developer guidelines: A brief summary defines a set of coding conventions to facilitate uniform code and documentation look and feel. Automatic checking of compliance to a subset of the coding standards is provided through a custom *PyLint*²¹ configuration, allowing early stage minor bug catching.

Moreover, PyMVPA does not raise barriers by being limited to specific platforms. It could fully or partially be used on any platform supported by Python (depending on the availability of external dependencies). However, to improve the accessibility, we provide binary installers for Windows, and MacOS X, as well as binary packages for Debian GNU/Linux (included in the official repository), Ubuntu, and a large number of RPM-based GNU/Linux distributions, such as OpenSUSE, RedHat, CentOS, Mandriva, and Fedora. Additionally, the available documentation provides detailed instructions on how to build the packages from source on many platforms.

A final important feature of PyMVPA is that it allows, by design, researchers to compress complex analyses into a small amount of code. This makes it possible to complement publications with the source code actually used to perform the analysis as Supplementary Material. Making this critical piece of information publicly available allows for in-depth reviews of the applied methods on a level well beyond what is possible with verbal descriptions. To demonstrate this feature, this paper is accompanied by the full source code to perform all analyses shown in the following sections.

ILLUSTRATIVE EXAMPLES: PyMVPA ON DIFFERENT MODALITIES

In this section we provide example analyses of four datasets, each from a different modality (EEG, MEG, fMRI, and extracellular recordings). All examples follow the same basic analysis pipeline: initial modality-specific preprocessing, application of ML methods, and visualization of the results. For the modality-independent

machine learning stage, all four examples employ the same analysis with *exactly* the same source code. Specifically, we first perform cross-validation with one or more classifiers on each dataset then compute feature-wise sensitivity measures. These measures can then be examined to reveal their implications in terms of the underlying research question.

These examples do not aim to provide an overview of the full functionality available within PyMVPA, but rather to show that ML methods can be easily applied to various types of data to provide meaningful and even thought-provoking results.

EEG

The dataset used for the EEG example consists of a single participant from a previously published study on object recognition (Fründ et al., 2008). In the experiment, participants indicated, for a sequence of images, whether they considered each particular image a meaningful object or just object-like with a meaningless configuration. This task was performed for two sets of stimuli with different statistical properties and under two different speed constraints. EEG was recorded from 31 electrodes at a sampling rate of 500 Hz using standard recording techniques. Details of the recording procedure can be found in Fründ et al. (2008). A detailed description of the stimuli can be found in Busch et al. (2006, colored images) and in Herrmann et al. (2004, line-art pictures).

Fründ et al. (2008) performed a wavelet-based time-frequency analyses of channels from a posterior region of interest (ROI) (i.e., no multivariate methods were employed). Here, we apply multivariate methods to differentiate between two conditions: trials with colored stimuli (broad spectrum of spatial frequencies and a high level of detail) and trials with black and white line-art stimuli (**Figure 2A**), collapsing the data across all other conditions. This discrimination is orthogonal to the participants task of indicating object vs. non-object stimuli.

The data for this analysis were 700 ms EEG segments starting 200 ms prior to the stimulus onset of each trial, to which we applied the following preprocessing procedure. We only included trials that passed the semi-automatic artifact rejection procedure performed in the original study, yielding 852 trials (422 color and 430 line-art). Each trial timeseries was downsampled to 200 Hz, leaving 140 sample points per trial and electrode. We then defined each trial, including the EEG signal of all sample points from all channels, as a sample to be classified (4340 features total). Finally, all features for each sample were normalized to zero mean and unit variance (z-scored).

As the main analysis we applied a standard sixfold cross-validation²² procedure with *linear support vector machine* (linC-SVM; Vapnik, 1995), *sparse multinomial logistic regression* (SMLR; Krishnapuram et al., 2005) and *Gaussian process regression* with linear kernel (linGPR; Rasmussen and Williams, 2006) classifiers. Additionally, we computed the multivariate I-RELIEF (Sun, 2007) feature sensitivity measures, and, for comparison, a univariate analysis of variance (ANOVA) *F*-score on the same cross-validation dataset splits.

All three classifiers performed with high accuracy on the independent test datasets, achieving 86.2% (linCSVM), 91.8% (SMLR),

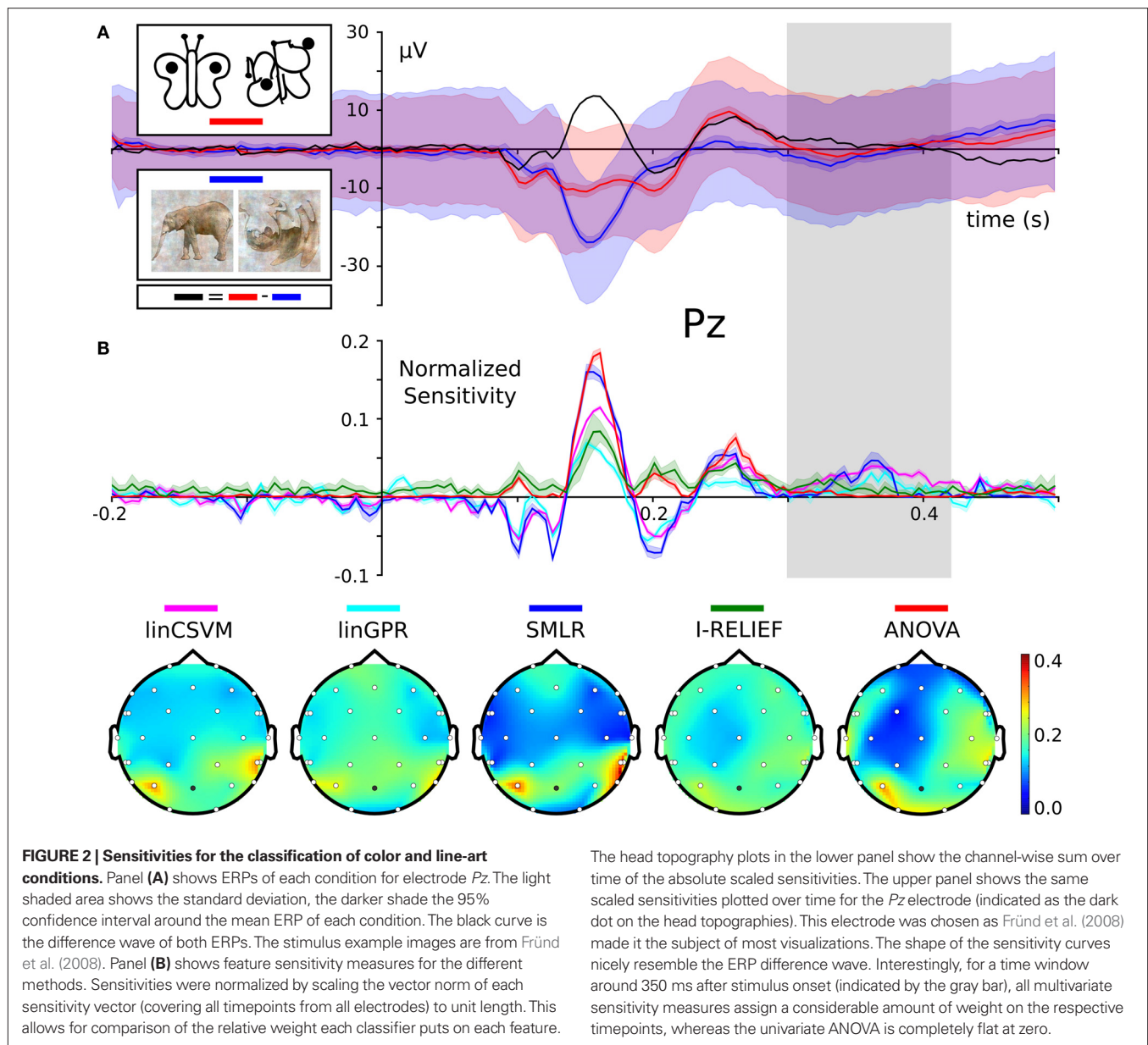
¹⁸http://en.wikipedia.org/wiki/Version_control_system.

¹⁹<http://en.wikipedia.org/wiki/ReStructuredText>.

²⁰<http://www.ohloh.net/projects/pymvpa/factoids>.

²¹<http://www.logilab.org/projects/pylint>.

²²http://en.wikipedia.org/wiki/Cross-validation#K-fold_cross-validation.



and 89.6% (linGPR) correct single trial predictions, respectively. However, more interesting than the plain accuracy are the features each classifier relied upon to perform its predictions. PyMVPA makes it very easy to extract feature sensitivity information from all its classifiers using a uniform interface. **Figure 2B** shows the computed sensitivities from all classifiers and measures. There is a striking similarity between the shape of the classifier sensitivities plotted over time and the corresponding event-related potential (ERP) difference wave between the two experimental conditions (**Figure 2A**; example shown for electrode Pz, Fründ et al., 2008). The head topography plot of the sensitivities reveals a high variability with respect to the specificity among the multivariate measures. SVM, GPR and SMLR weights congruently identify three posterior electrodes as being most informative (SMLR weights provide the highest contrast of all measures). The I-RELIEF topography is much

less specific and more similar to the ANOVA topography in its global spatial structure than to the other multivariate measures. It should be noted, however, that these topographies aggregate information over all timepoints and, therefore, do not provide information about specific temporal EEG components.

One particularly interesting result is the difference between the multivariate sensitivities and the univariate ANOVA *F*-scores from 300 to 400 ms following stimulus onset. Only the multivariate methods (especially SMLR, linCSVM and linGPR) detected a relevant contribution to the classification task of the signal in this time window. This late signal may be related to the intracranial EEG gamma-band responses that Lachaux et al. (2005) observed at around the same time range when participants viewed complex stimuli. Given that the present data also seem to show a similar evoked gamma-band response (Fründ et al., 2008), it is possible

that the multivariate methods are sensitive to the gamma-band activity in the data. Still, further work would be required to prove this correlation.

MEG

The example MEG dataset was collected with the aim to test whether it is possible to predict the recognition of briefly presented natural scenes from single trial MEG-recordings of brain activity (Rieger et al., 2008) and to use ML methods to investigate the properties of the brain activity that is predictive of later recognition. On each trial participants saw a briefly presented photograph (37 ms) of a natural scene that was immediately followed by a pattern mask (1000–1400 ms). The short masked presentation effectively limits the processing interval of the scene in the brain (Rieger et al., 2005) and, therefore, participants will later recognize only some of the scenes. After the mask was turned off, participants indicated via button presses whether they would subsequently recognize the photograph, or if they would fail. Immediately after this judgement, four natural scene photographs were presented and participants had to indicate which of the four scenes had been previously presented (i.e., a four-alternative forced-choice delayed match to sample task).

The MEG was recorded with a 151 channel CTF Omega MEG system from the whole head (sampling rate 625 Hz and a 120 Hz analogue low pass filter) while participants performed this task. The 600 ms interval of the MEG time series data that was used for the analysis started at the onset of the briefly presented scene and ended before the mask was turned off. As in the original study, we analyzed only those trials in which participants both judged they would be correct and also correctly recognized the scene (*RECOG*) and the trials in which participants both predicted they would fail and gave an incorrect response (*NRECOG*). For details about the rationale of this selection, the stimulus presentation information, and the recording procedure see Rieger et al. (2008). In this example analysis we have used data from a single participant (labeled P1 in the original publication).

The MEG timeseries were first downsampled to 80 Hz and then all trial segments were channel-wise normalized by subtracting their mean baseline signal (determined from a 200 ms window prior to scene onset). Only timepoints within the first 600 ms after stimulus onset were considered for further analysis. The resulting dataset consisted of 151 channels with 48 timepoints each (7248 features), and a total of 294 samples (233 *RECOG* trials and 61 *NRECOG* trials).

The original study contained analyses based upon SVM classifiers, which revealed, by means of the spatio-temporal distribution of the sensitivities, that the theta band alone provides the most discriminative signal. The authors also addressed the topic of how to interpret heavily unbalanced datasets²³. Given this comprehensive analysis, we aimed here to replicate their basic analysis strategy with PyMVPA and were able to achieve almost identical results.

As with the EEG data, we applied a standard cross-validation procedure, this time eightfold, using linear SVM and SMLR classifiers. Additionally, we again computed univariate ANOVA *F*-scores

on the same cross-validation dataset splits. The SVM classifier was configured to use different per-class *C*-values²⁴, scaled with respect to the number of samples in each class to address the unbalanced number of samples. Similar to Rieger et al. (2008), we also ran a second cross-validation on balanced datasets (by performing multiple selections of a random subset of samples from the larger *RECOG* category).

Both classifiers performed almost identically on the full, unbalanced dataset, achieving 84.69% (SMLR) and 82.31% (linCSVM) correct single trial predictions (83.0% in the original study). **Figure 3** shows sample timeseries of the classifier sensitivities and the ANOVA *F*-score of two posterior channels. Due to the significant difference in the number of samples of each category, it is important to additionally report mean true positive rate (TPR)²⁵, that amounted to 72% (SMLR), and 76% (linCSVM) respectively. The second SVM classifier trained on the balanced dataset achieved a comparable accuracy of 76.07% correct predictions (mean across 100 subsampled datasets), which is a slightly larger drop in accuracy when compared to the 80.8% achieved in the original study (see Table 3 in Rieger et al., 2008).

Importantly, these results show that PyMVPA produces reproducible results that depend on the ML methods employed, but not on a particular implementation. However, the integrated framework of PyMVPA allowed us to achieve these results with much less effort than what was necessary in the original study.

fMRI

A single participant (participant 1) from a study published by Haxby et al. (2001), which has been repeatedly reanalyzed since the original publication (Hanson and Halchenko, 2008; Hanson et al., 2004; O'Toole et al., 2007), served as the example fMRI dataset. The dataset itself consists of 12 runs. In each run, the participant passively viewed grayscale images of eight object categories, grouped in 24 s blocks separated by rest periods. Each image was shown for 500 ms and was followed by a 1500 ms inter-stimulus interval. Full-brain fMRI data were recorded with a volume repetition time of 2500 ms, thus, a stimulus block was covered by roughly nine volumes. For a complete description of the experimental design and fMRI acquisition parameters see Haxby et al. (2001).

First, the raw fMRI data were motion corrected using FLIRT²⁶ from *FSL*²⁷ (Jenkinson et al., 2002). All subsequent data processing was done with PyMVPA. After motion correction, linear detrending was performed for each run individually. No additional spatial or temporal filtering was applied.

For the sake of simplicity, we reduced the dataset to a four-class problem (*faces, houses, cats, and shoes*). All volumes recorded during any of these blocks were extracted and voxel-wise *z*-scored. This normalization was performed individually for each run to prevent any kind of information transfer across runs.

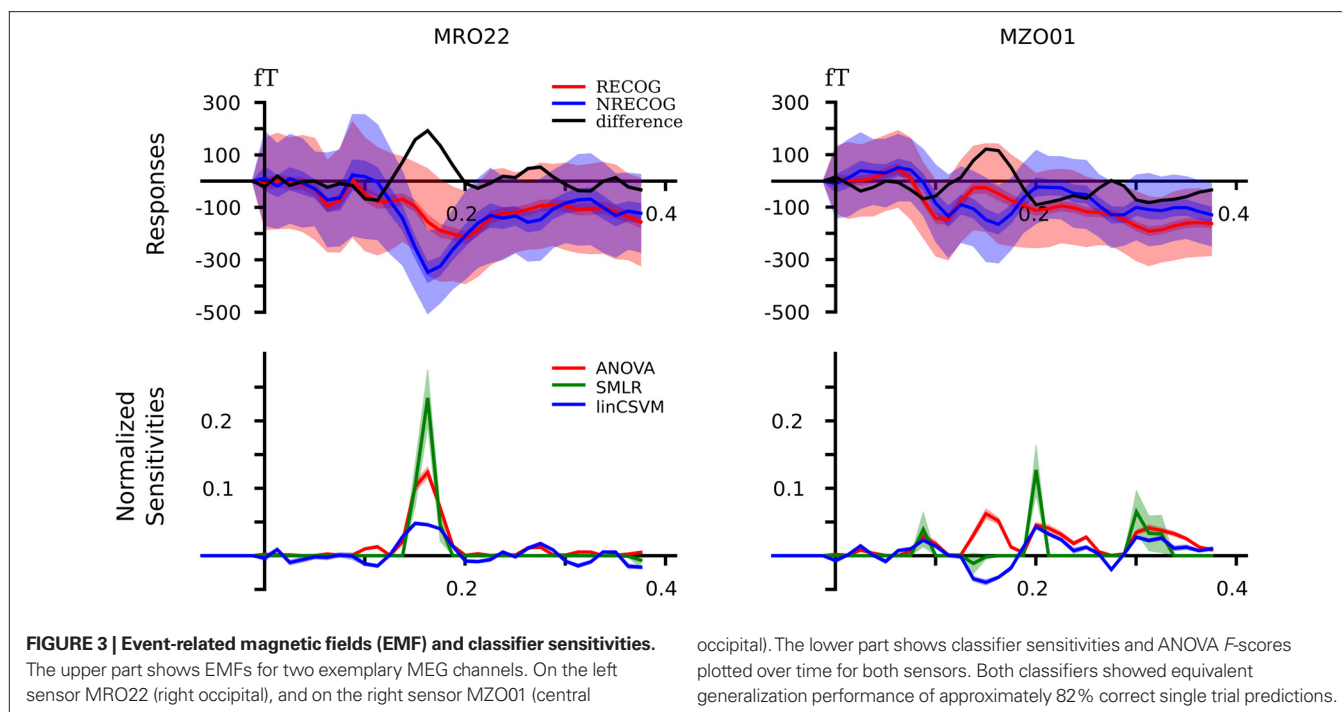
²⁴Parameter *C* in soft-margin SVM controls a trade-off between width of the SVM margin and number of support vectors (see Veropoulos et al., 1999, for an evaluation of this approach).

²⁵Mean TPR is equivalent to accuracy in balanced sets, and is 50% at chance performance even with unbalanced set sizes (see Rieger et al., 2008, for a discussion of this point).

²⁶FMRIB's Linear Image Registration Tool.

²⁷<http://www.fmrib.ox.ac.uk/fsl>.

²³Unbalanced datasets have a dominant category which has considerably more samples than any other category. That potentially leads to the problem when a classifier prefers to assign the label of that category to all samples to minimize total prediction error.



After preprocessing, we applied the same sensitivity analysis performed for all other data modalities to this dataset. Here, only a SMLR classifier was used (sixfold cross-validation, with 2 of the 12 experimental runs grouped into one chunk, and trained on single fMRI volumes that covered the full brain). For comparison, a univariate ANOVA was again computed for the same cross-validation dataset splits.

The SMLR classifier performed very well on the independent test datasets, correctly predicting the category for 94.7% of all single volume samples in the test datasets. To examine what information was used by the classifier to reach this performance level, we computed ROI-based sensitivity scores for 48 non-overlapping structures defined by the probabilistic Harvard-Oxford cortical atlas (Flitney et al., 2007), as shipped with FSL (Smith et al., 2004). To create the ROIs, we thresholded the probability maps of all structures at 25% and assigned ambiguous voxels to the structure with the higher probability. The resulting map was projected into the space of the functional dataset using an affine transformation and nearest neighbor interpolation.

In order to determine the contribution of each ROI, the sensitivity vector was first normalized (across all ROIs), so that all absolute sensitivities summed up to 1 (L1-normed). Afterwards ROI-wise scores were computed by taking the sum of all sensitivities in a particular ROI. The upper part of **Figure 4** shows these scores for the 20 highest-scoring and the three lowest-scoring ROIs.

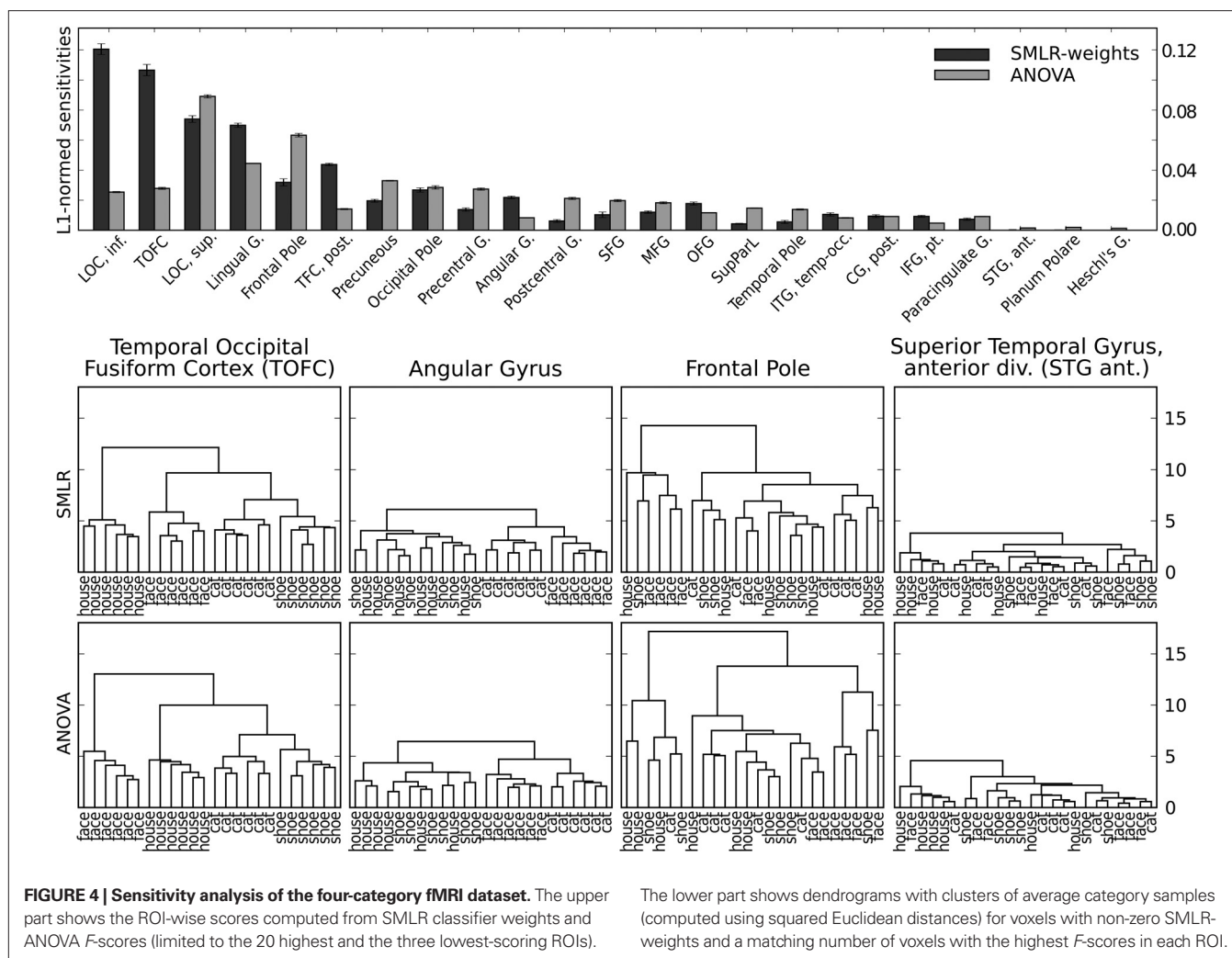
The lower part of the figure shows dendrograms from a hierarchical cluster analysis²⁸ on relevant voxels from a block-averaged variant of the dataset (but otherwise identical to the classifier training data). For SMLR, only voxels with a non-zero sensitivity were considered in each particular ROI. For ANOVA, only the voxels with

the highest *F*-scores (limited to the same number as for the SMLR case) were considered. For visualization purposes the dendrograms show the distances and clusters computed from the average samples of each condition in each dataset chunk (i.e., two experimental blocks), yielding six samples per condition.

The four chosen ROIs clearly show four different cluster patterns. The 92 selected voxels in temporal occipital fusiform cortex (TOFC) show a clear clustering of the experimental categories, with relatively large sample distances between categories. The pattern of the 36 voxels in angular gyrus reveals an animate/inanimate clustering, although with much smaller distances. The largest group of 148 voxels in the frontal pole ROI seems to have no obvious structure in their samples. Despite that, both sensitivity measures assign substantial importance to this region. This might be due to the large inter-sample distances visualized in the corresponding dendrogram in **Figure 4**. Each leaf node (in this case an average volume of two stimulation blocks) is approximately as distinct from any other leaf node, in terms of the employed distance measure, as the semantic clusters identified in the TOFC ROI. Finally, the ROI covering the anterior division of the superior temporal gyrus shows no clustering at all, and, consequently, is among the lowest-scoring ROIs of both measures. On the whole, the cluster patterns from voxels selected by SMLR weights and *F*-scores are very similar in terms of inter-cluster distances.

Given that these results only include the data of a single participant, no far-reaching implications can be drawn from them. However, the distinct cluster patterns might provide indications for different levels of information encoding that could be addressed in future studies. Although voxels selected in both angular gyrus and the frontal pole ROIs do not provide a discriminative signal for all four stimulus categories, they nevertheless provide some disambiguating information and, thus, are picked up by the classifier.

²⁸PyMVPA provides hierarchical clustering facilities through *hcluster* (Eads, 2008).



In angular gyrus, this seems to be an animate/inanimate pattern that additionally also differentiates between the two categories of animate stimuli. Finally, in the frontal pole ROI the pattern remains unclear, but the relatively large inter-sample distances indicate a differential code of some form that is not closely related to the semantic stimulus category.

EXTRACELLULAR RECORDINGS

The extracellular dataset analyzed in this section is previously unpublished, thus, we first briefly describe the experimental and acquisition setup. Animal experiments were carried out in accordance with the National Institute of Health Guide for the Care and Use of Laboratory Animals and approved by Rutgers University. Sprague-Dawley rats (300–500 g) were anaesthetized with urethane (1.5 g/kg) and held with a custom naso-orbital restraint. After preparing a 3 mm square window in the skull over auditory cortex, the dura was removed and a silicon microelectrode consisting of eight four-site recording shanks (NeuroNexus Technologies, Ann Arbor, MI, USA) was inserted. The recording sites were in the primary auditory cortex, estimated by stereotaxic coordinates, vascular structure (Sally and Kelly, 1988) and

tonotopic variation of frequency tuning across recording shanks, and located within layer V, determined by electrode depth and firing patterns.

Five pure tones (3, 7, 12, 20, 30 kHz at 60 dB) and five different natural sounds (extracted from the CD “Voices of the Swamp”, Naturesound Studio, Ithaca, NY, USA) were used as stimuli. Each stimulus had a duration of 500 ms followed by 1500 ms of silence. All stimuli were tapered at beginning and end with a 5 ms cosine window. The data acquisition took place in a single-walled sound isolation chamber (IAC, Bronx, NY, USA) with sounds presented free field (RP2/ES1, Tucker-Davis, Alachua, FL, USA).

Individual units²⁹ were isolated by a semi-automatic algorithm (*KlustaKwik*³⁰) followed by manual clustering (*Klusters*³¹). Post-stimulus time histograms (PSTH) of spike counts per each unit for all 1734 stimulation onsets were estimated using a bin size of 3.2 ms. To ensure an accurate estimation of PSTHs only units with a

²⁹The term “unit” in the text refers to a single entity, which was segregated from the recorded data, and is expected to represent a single neuron.

³⁰<http://klustakwik.sourceforge.net>.

³¹<http://klusters.sourceforge.net>.

mean firing rate higher than 2 Hz were selected for further analysis, leaving us with a total of 105 units.

Since the segregation of individual units out of the extracellular recordings is carried out without taking the respective stimulus condition into account, i.e., in unsupervised fashion (in ML terminology), it does not guarantee that the activity of any particular unit can be easily attributed to some set of stimulus conditions. From the stimulus-wise descriptive statistics of the units presented in the top plots of **Figure 5** it is difficult to state that the activity of any particular unit at some moment in time is specific for a given stimulus. Furthermore, due to the inter-trial variance in the spike counts, it is even more difficult to reliably assess what stimulus condition any particular trial belongs to. Hence, the purpose of the PyMVPA analysis was to complement the results of the unsupervised clustering with a characterization of all extracted units in terms of their specificity to any given stimulus at any given time.

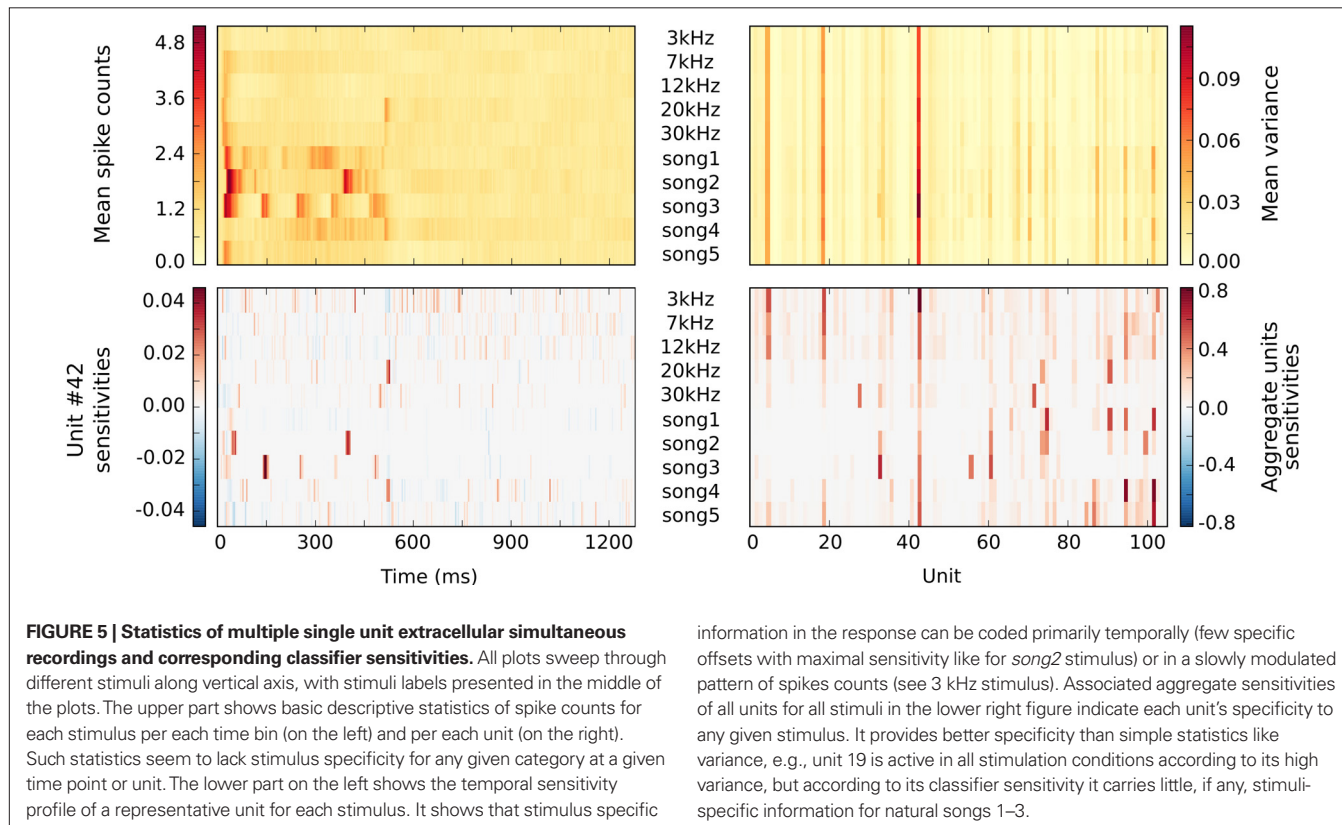
The analysis pipeline was similar to the one used for EEG, MEG, and fMRI data. We ran a standard eightfold cross-validation procedure for an SMLR classifier, which achieved a mean of 77.57% accuracy estimate across all 10 types of stimuli. This generalization accuracy is well above chance (10%) for all stimulus categories and allows one to conclude that the neuronal population activity pattern at the recording site carries a differential signal across all 10 stimuli. Misclassifications mostly occurred for low-frequency stimuli. Pure tones with 3 and 7 kHz were more often confused with each other than tones with a larger frequency

difference (see **Figure 6**), which suggests a high similarity in the spiking patterns for these stimuli. We could further speculate that this neuronal population is more tuned towards the processing of higher frequency tones.

Besides being able to label yet unseen trials with high accuracy, the trained classifier can readily provide its sensitivity estimates for each unit, time bin, and stimulus condition (see bottom plots of **Figure 5**). Temporal sensitivity profiles of any particular unit (see unit #42 profiles in lower left plot of **Figure 5**) can reveal that the stimulus specific information is contained in spike times relative to stimulus onset or can be represented as slowly modulated pattern of spike counts (see 3 kHz stimuli). An aggregate sensitivity (in this case the sum of absolute sensitivities) across all time-bins provides a summary statistic of any unit's sensitivity to a given stimulus condition (see lower right plot of **Figure 5**). In contrast to a simple variance measure, it provides an easier way to associate any given unit to a set of stimulus conditions. Additionally, it can identify units which might lack a substantial amount of variance, but nevertheless carry a stimulus-specific signal (e.g. unit #28 and 30 kHz stimulus).

CONCLUSIONS

In this article we presented PyMVPA, a data analysis framework especially tailored to neural data from a wide range of acquisition modalities. PyMVPA provides ML techniques as core functionality, addressing recent trends in neuroscience research. To illustrate the generalizability of the PyMVPA analysis pipeline we provided



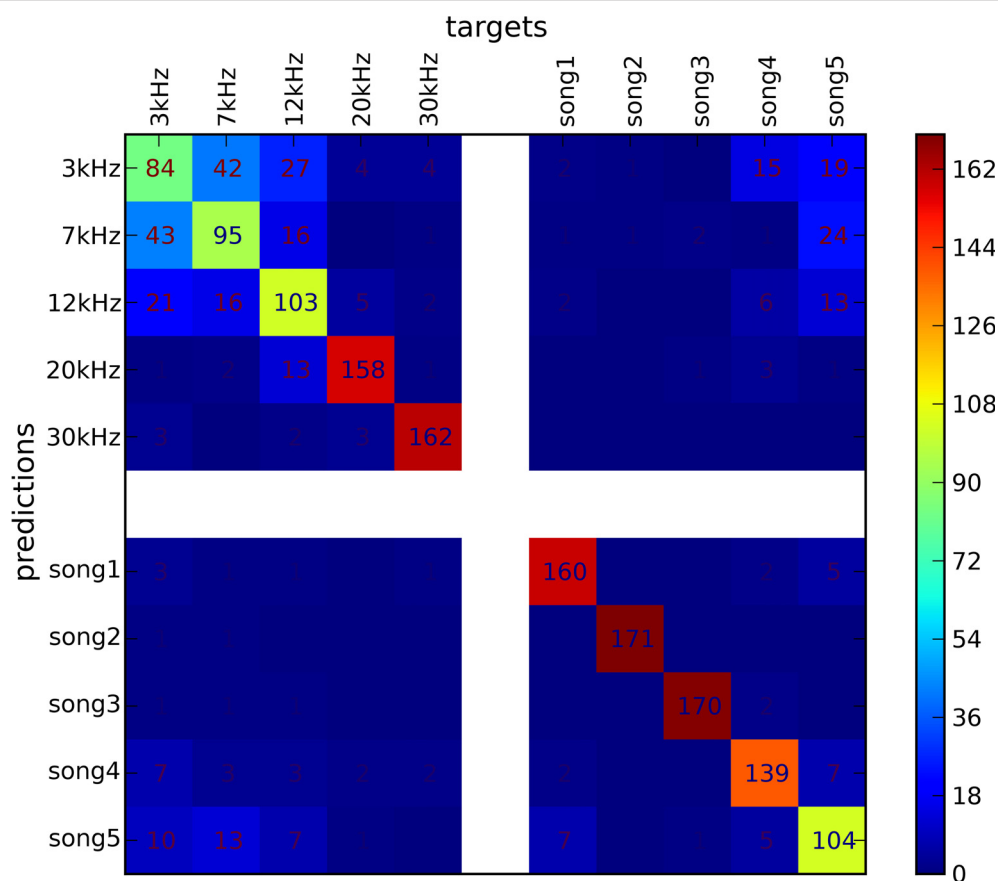


FIGURE 6 | Confusion matrix of SMLR classifier predictions of stimulus conditions from multiple unit recordings. The classifier was trained to discriminate between stimuli of five pure tones and five natural sounds. Elements of the matrix (numeric values and color-mapped visualization) show the number of trials which were correctly (diagonal) or incorrectly

(off-diagonal) classified by a SMLR classifier during an eightfold cross-validation procedure. The results suggest a high similarity in the spiking patterns for stimuli of low-frequency pure tones, which lead the classifier to confuse them more often, whenever responses to natural sound stimuli and high-frequency tones were hardly ever confused with each other.

example analyses of data from EEG, MEG, fMRI and extracellular recordings.

The framework presented here is Python-based, sophisticated, free and open-source software. Its intended audience is threefold. First, there are *neuroscience researchers* interested in testing ML algorithms on neural data, e.g., people working on brain-computer interfaces (BCI, see Birbaumer and Cohen, 2007; Lebedev and Nicolelis, 2006). PyMVPA provides researchers with the ability to execute complex analysis tasks in very concise code. Second, it is also designed for *ML researchers* interested in testing new ML algorithms on neural data. PyMVPA offers a highly-modularized architecture designed to minimize the effort of adding new algorithms. Moreover, the availability of neuroscience-related code-examples (like the ones presented in this article) and datasets greatly reduces the time to get actual results. Finally, PyMVPA is welcoming *code contributors* from both neuroscience and ML communities interested in improving or adding modality-specific functions or new algorithms. PyMVPA offers a community-based development model together with a distributed version control system and extensive reference documentation.

FUTURE WORK

PyMVPA does not aim to provide all possible ML analysis algorithms, and it will likely not come close, even in the future. Given that PyMVPA is tailored towards the high-dimensional problems found in neuroscience, it currently provides many of the most common algorithms tuned for this target. Still, as the neuroscience and ML communities unite, new and promising algorithms are constantly emerging and being added to PyMVPA. Beyond the inclusion of new ML algorithms, there are numerous plans for future enhancements to PyMVPA.

Because the current use of ML techniques in neuroscience is mainly limited to the application of only basic algorithms to neural data, one of the next, most intriguing, new directions of PyMVPA will be to provide *custom* workflows designed for specific neuroscience modalities. An example of such a custom workflow is the analysis of fMRI data from experiments with event-related designs, where multiple fMRI volumes after the onset of the event compose a single sample within a dataset provided to the ML methods for processing. Combining multiple volumes into a single sample obviates the need to provide a hemodynamic response function because the important features can be extracted independently for each voxel.

In addition, PyMVPA has yet to confront the problem of model selection. Currently, only Gaussian process regression has the ability to select hyper-parameters of the model. Uniform model selection for ML methods within PyMVPA is planned for the next major release of the project. It will provide the facility to automatically search for the best set of parameters for each classifier without sacrificing unbiased estimates of the generalization performance.

SUPPLEMENTAL MATERIAL

The Supplemental Materials (e.g., source code) for this article can be found online at <http://www.frontiersin.org/neuroinformatics/paper/10.3389/neuro.11/003.2009>.

REFERENCES

- Beckmann, C. F., and Smith, S. M. (2005). Tensorial extensions of independent component analysis for multisubject fMRI analysis. *Neuroimage* 25, 294–311.
- Birbaumer, N., and Cohen, L. G. (2007). Brain-computer interfaces: communication and restoration of movement in paralysis. *J. Physiol.* 579, 621–636.
- Busch, N. A., Herrmann, C. S., Müller, M. M., Lenz, D., and Gruber, T. (2006). A cross-laboratory study of event-related gamma activity in a standard object recognition paradigm. *Neuroimage* 33, 1169–1177.
- Detre, G., Polyn, S. M., Moore, C., Natu, V., Singer, B., Cohen, J., Haxby, J. V., and Norman, K. A. (2006). The Multi-Voxel Pattern Analysis (MVPA) Toolbox. Poster presented at the Annual Meeting of the Organization for Human Brain Mapping (Florence, Italy). Available at: <http://www.csmbm.princeton.edu/mvpa>.
- Eads, D. (2008). Hcluster: Hierarchical Clustering for SciPy. Available at: <http://scipy-cluster.googlecode.com/>.
- Efron, B., and Tibshirani, R. (1993). An Introduction to the Bootstrap. New York, NY: Chapman & Hall/CRC.
- Flitney, D., Webster, M., Patenaude, B., Seidman, L., Goldstein, J., Tordesillas Gutierrez, D., Eickhoff, S., Amunts, K., Zilles, K., Lancaster, J., Haselgrove, C., Kennedy, D., Jenkinson, M., and Smith, S. (2007). Anatomical Brain Atlases and Their Application in the FSLView Visualisation Tool. Thirteenth Annual Meeting of the Organization for Human Brain Mapping, Chicago, IL, USA.
- Fründ, I., Busch, N. A., Schadow, J., Gruber, T., Körner, U., and Herrmann, C. S. (2008). Time pressure modulates electrophysiological correlates of early visual processing. *PLoS ONE* 3, e1675.
- Guyon, I., and Elisseeff, A. (2003). An introduction to variable and feature selection. *J. Mach. Learn. Res.* 3, 1157–1182.
- Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Mach. Learn.* 46, 389–422.
- Hanke, M., Halchenko, Y. O., Sederberg, P. B., Hanson, S. J., Haxby, J. V., and Pollmann, S. (2009). PyMVPA: A Python toolbox for multivariate pattern analysis of fMRI data. *Neuroinformatics*. doi: 10.1007/s12021-008-9041-y.
- Hanke, M., Halchenko, Y. O., Sederberg, P. B., and Hughes, J. M. (2008). The PyMVPA Manual. Available at: http://www.py_mvpa.org/PyMVPA-Manual.pdf.
- Hanson, S., Matsuka, T., and Haxby, J. (2004). Combinatorial codes in ventral temporal lobe for object recognition: Haxby (2001). revisited: is there a “face” area? *Neuroimage* 23, 156–166.
- Hanson, S. J., and Halchenko, Y. O. (2008). Brain reading using full brain support vector machines for object recognition: there is no “face” identification area. *Neural Comput.* 20, 486–503.
- Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J., and Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. *Science* 293, 2425–2430.
- Haynes, J.-D., and Rees, G. (2006). Decoding mental states from brain activity in humans. *Nat. Rev. Neurosci.* 7, 523–534.
- Haynes, J.-D., Sakai, K., Rees, G., Gilbert, S., Frith, C., and Passingham, R. E. (2007). Reading hidden intentions in the human brain. *Curr. Biol.* 17, 323–328.
- Herrmann, C. S., Lenz, D., Junge, S., Busch, N. A., and Maess, B. (2004). Memory-matches evoke human gamma-responses. *BMC Neurosci.* 5, 13.
- Jenkinson, M., Bannister, P., Brady, J., and Smith, S. (2002). Improved optimisation for the robust and accurate linear registration and motion correction of brain images. *Neuroimage* 17, 825–841.
- Kamitani, Y., and Tong, F. (2005). Decoding the visual and subjective contents of the human brain. *Nat. Neurosci.* 8, 679–685.
- Kriegeskorte, N., and Bandettini, P. (2007). Analyzing for information, not activation, to exploit high-resolution fMRI. *Neuroimage* 38, 649–662.
- Kriegeskorte, N., Goebel, R., and Bandettini, P. (2006). Information-based functional brain mapping. *Proc. Natl. Acad. Sci. U.S.A.* 103, 3863–3868.
- Krishnapuram, B., Carin, L., Figueiredo, M. A., and Hartemink, A. J. (2005). Sparse multinomial logistic regression: fast algorithms and generalization bounds. *IEEE Trans. Pattern Anal. Mach. Intell.* 27, 957–968.
- Lachaux, J.-P., George, N., Tallon-Baudry, C., Martinerie, J., Hugueville, L., Minotti, L., Kahane, P., and Renault, B. (2005). The many faces of the gamma band response to complex visual stimuli. *Neuroimage* 25, 491–501.
- Lebedev, M. A., and Nicolelis, M. A. L. (2006). Brain-machine interfaces: past, present and future. *Trends Neurosci.* 29, 536–546.
- Makeig, S., Debener, S., Onton, J., and Delorme, A. (2004). Mining event-related brain dynamics. *Trends Cogn. Sci.* 8, 204–210.
- Norman, K. A., Polyn, S. M., Detre, G. J., and Haxby, J. V. (2006). Beyond mind-reading: multi-voxel pattern analysis of fMRI data. *Trends Cogn. Sci.* 10, 424–430.
- O’Toole, A. J., Jiang, F., Abdi, H., Penard, N., Dunlop, J. P., and Parent, M. A. (2007). Theoretical, statistical, and practical perspectives on pattern-based classification approaches to the analysis of functional neuroimaging data. *J. Cogn. Neurosci.* 19, 1735–1752.
- Pereira, F., Mitchell, T., and Botvinick, M. (in press). Machine learning classifiers and fMRI: a tutorial overview. doi: 10.1016/j.neuroimage.2008.11.007.
- Rasmussen, C. E., and Williams, C. K. (2006). Gaussian Processes for Machine Learning. Cambridge, MA: MIT Press.
- Rieger, J. W., Braun, C., Bülthoff, H. H., and Gegenfurtner, K. R. (2005). The dynamics of visual pattern masking in natural scene processing: a magnetoencephalography study. *J. Vis.* 5, 275–286.
- Rieger, J. W., Reichert, C., Gegenfurtner, K. R., Noesselt, T., Braun, C., Heinze, H.-J., Kruse, R., and Hinrichs, H. (2008). Predicting the recognition of natural scenes from single trial MEG recordings of brain activity. *Neuroimage* 42, 1056–1068.
- Sally, S. L., and Kelly, J. B. (1988). Organization of auditory cortex in the albino rat: sound frequency. *J. Neurophysiol.* 59, 1627–1638.
- Smith, S. M., Jenkinson, M., Woolrich, M. W., Beckmann, C. F., Behrens, T. E. J., Johansen-Berg, H., Bannister, P. R., De Luca, M., Drobnjak, I., Flitney, D. E., Niazy, R. K., Saunders, J., Vickers, J., Zhang, Y., De Stefano, N., Brady, J. M., and Matthews, P. M. (2004). Advances in functional and structural MR image analysis and implementation as FSL. *Neuroimage* 23, 208–219.

- Sonnenburg, S., Braun, M., Ong, C. S., Bengio, S., Bottou, L., Holmes, G., LeCun, Y., Müller, K.-R., Pereira, F., Rasmussen, C. E., Rätsch, G., Schölkopf, B., Smola, A., Vincent, P., Weston, J., and Williamson, R. (2007). The need for open source software in machine learning. *J. Mach. Learn. Res.* 8, 2443–2466.
- Sun, Y. (2007). Iterative RELIEF for feature weighting: algorithms, theories and applications. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 1035–1051.
- Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. New York, Springer.
- Veropoulos, K., Campbell, C., and Cristianini, N. (1999). Controlling the Sensitivity of Support Vector Machines. *Proceedings of the International Joint Conference on AI*. Stockholm, Sweden.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 14 September 2008; paper pending published: 20 October 2008; accepted: 20 January 2009; published online: 04 February 2009.
- Citation: Hanke M, Halchenko YO, Sederberg PB, Olivetti E, Fründ I, Rieger JW, Herrmann CS, Haxby JV, Hanson SJ and Pollmann S (2009) PyMVPA: a unifying approach to the analysis of neuroscientific data. *Front. Neuroinform.* (2009) 3:3. doi: 10.3389/neuro.11.003.2009
- Copyright © 2009 Hanke, Halchenko, Sederberg, Olivetti, Fründ, Rieger, Herrmann, Haxby, Hanson and Pollmann. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



PyNEST: A convenient interface to the NEST simulator

Jochen Martin Eppler^{1,2*}, Moritz Helias^{2†}, Eilif Muller³, Markus Diesmann^{2,4,5} and Marc-Oliver Gewaltig^{1,2}

¹ Honda Research Institute Europe GmbH, Offenbach, Germany

² Bernstein Center for Computational Neuroscience, Albert-Ludwig University, Freiburg, Germany

³ Laboratory for Computational Neuroscience, Swiss Federal Institute of Technology, EPFL, Lausanne, Switzerland

⁴ Theoretical Neuroscience Group, RIKEN Brain Science Institute, Wako City, Japan

⁵ Brain and Neural Systems Team, Computational Science Research Program, RIKEN, Wako City, Japan

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Upinder S. Bhalla, National Center for
Biological Sciences, India

Terrence C. Stewart, Carleton
University, Canada

*Correspondence:

Jochen Martin Eppler, Honda
Research Institute Europe GmbH,
Carl-Legien-Str. 30, 63073 Offenbach
am Main, Germany.
e-mail: eppler@biologie.uni-freiburg.de

[†]Eppler and Helias contributed equally
to this work.

The neural simulation tool NEST (<http://www.nest-initiative.org>) is a simulator for heterogeneous networks of point neurons or neurons with a small number of compartments. It aims at simulations of large neural systems with more than 10^4 neurons and 10^7 to 10^9 synapses. NEST is implemented in C++ and can be used on a large range of architectures from single-core laptops over multi-core desktop computers to super-computers with thousands of processor cores. Python (<http://www.python.org>) is a modern programming language that has recently received considerable attention in Computational Neuroscience. Python is easy to learn and has many extension modules for scientific computing (e.g. <http://www.scipy.org>). In this contribution we describe PyNEST, the new user interface to NEST. PyNEST combines NEST's efficient simulation kernel with the simplicity and flexibility of Python. Compared to NEST's native simulation language SLI, PyNEST makes it easier to set up simulations, generate stimuli, and analyze simulation results. We describe how PyNEST connects NEST and Python and how it is implemented. With a number of examples, we illustrate how it is used.

Keywords: Python, modeling, integrate-and-fire neuron, large-scale simulation, scientific computing, networks, programming

INTRODUCTION

The first user interface for NEST (Gewaltig and Diesmann, 2007; Plesser et al., 2007) was the simulation language SLI, a stack-based language derived from PostScript (Adobe Systems Inc., 1999). However, programming in SLI turned out to be difficult to learn and users asked for a more convenient programming language for NEST.

When we decided to use Python as the new simulation language, it was almost unknown in Computational Neuroscience. In fact, Matlab (MathWorks, 2002) was far more common, both for simulations and for analysis. Other simulators, like e.g. CSIM (Natschläger, 2003), already used Matlab as their interface language. Thus, Matlab would have been a natural choice for NEST as well.

Python has a number of advantages over commercial software like Matlab and other free scripting languages like Tcl/Tk (Ousterhout, 1994). First, Python is installed by default on all Linux and Mac-OS based computers. Second, Python is stable, portable, and supported by a large and active developer community, and has a long history in scientific fields outside the neurosciences (Dubois, 2007). Third, Python is a powerful interactive programming language with a surprisingly concise and readable syntax. It supports many programming paradigms such as object-oriented and functional programming. Through packages like NumPy (<http://www.numpy.org>) and SciPy (<http://www.scipy.org>), Python supports scientific computing and visualization à la Matlab. Finally, a number of neuroscience laboratories meanwhile use Python for simulation and analysis, which further supports our choice.

Python is powerful at steering other applications and provides a well documented interface (API) to link applications to Python

(van Rossum, 2008). To do so, it is common to map the application's functions and data structures to Python classes and functions. This approach has the advantage that the coupling between the application and Python is as tight as possible. But there is also a drawback: Whenever a new feature is implemented in the application, the interface to Python must be changed as well.

On many high-performance computers Python is not available and we have to preserve NEST's native simulation language SLI. In order to avoid two different interfaces, one to Python and one to SLI, we decided to deviate from the standard way of coupling applications to Python. Rather than using NEST's classes, we use NEST's simulation language as the interface: Python sends data and SLI commands to NEST and NEST responds with Python data structures.

Exchanging data between Python and NEST is easy since all important data types in NEST have equivalents in Python. Executing NEST commands from Python is also straightforward: Python only needs to send a string with commands to NEST, and NEST will execute them. With this approach, we only need to maintain one binary interface to the simulation kernel instead of two: Each new feature of the simulation kernel only needs to be mapped to SLI and immediately becomes accessible in PyNEST without changing its binary interface. This generic interpreter interface allows us to program PyNEST's high-level API in Python. This is an advantage, because programming in Python is more productive than programming in C++ (Prechelt, 2000). Python is also more expressive: A given number of lines of Python code achieve much more than the same number of lines in C++ (McConnell, 2004).

NEST users benefit from the increased productivity. They can now take advantage of the large number of extension modules for Python. NumPy is the Python interface to the BLAS libraries, the same libraries which power Matlab. Matplotlib (<http://matplotlib.sourceforge.net>) provides many routines to plot scientific data in publication quality. Many other packages exist to analyze and visualize data. Thus, PyNEST allows users to combine simulation, data analysis, and visualization in a single programming language.

In the Section “Using PyNEST”, we introduce the basic modeling concepts of NEST. With a number of PyNEST code examples, we illustrate how simulations are defined and how the results are analyzed and plotted. In the Section “The Interface Between Python and NEST”, we describe in detail how we bind NEST to the Python interpreter. In the Section “Discussion”, we discuss our implementation and analyze its performance. The complete API reference for PyNEST is contained in Appendix A. In Appendix B we illustrate advanced PyNEST features, using a large scale model.

USING PyNEST

A neural network in NEST consists of two basic element types: Nodes and connections. Nodes are either neurons, devices or subnetworks. Devices are used to stimulate neurons or to record from them. Nodes can be arranged in subnetworks to build hierarchical networks like layers, columns, and areas. After starting NEST, there is one empty subnetwork, the so-called *root node*. New nodes are created with the command `Create()`, which takes the model name and optionally the number of nodes as arguments and returns a list of handles to the new nodes. These handles are integer numbers, called *ids*. Most PyNEST functions expect or return a list of ids (see Appendix A). Thus it is easy to apply functions to large sets of nodes with a single function call.

Nodes are connected using `Connect()`. Connections have a configurable delay and weight. The weight can be static or dynamic, as for example in the case of spike timing dependent plasticity (STDP; Morrison et al., 2008). Different types of nodes and connections have different parameters and state variables. To avoid the problem of *fat interfaces* (Stroustrup, 1997), we use *dictionaries* with the functions `GetStatus()` and `SetStatus()` for the inspection and manipulation of an element's configuration. The properties of the simulation kernel are controlled through the commands `GetKernelStatus()` and `SetKernelStatus()`. PyNEST contains the submodules *raster_plot* and *voltage_trace* to visualize spike activity and membrane potential traces. They use Matplotlib internally and are good templates for new visualization functions. However, it is not our intention to develop PyNEST into a toolbox for the analysis of neuroscience data; we follow the modularity concept of Python and leave this task to others (e.g. NeuroTools, <http://www.neuralensemble.org/NeuroTools>).

EXAMPLE

We illustrate the key features of PyNEST with a simulation of a neuron receiving input from an excitatory and an inhibitory population of neurons (modified from Gewaltig and Diesmann, 2007). Each presynaptic population is modeled by a Poisson generator, which generates a unique Poisson spike train for each target. The simulation adjusts the firing rate of the inhibitory input population such that the neurons of the excitatory population and the target neuron fire at the same rate.

First, we import all necessary modules for simulation, analysis and plotting.

```
1 from nest import *
2 from scipy.optimize import bisect
3 import nest.voltage_trace as plot
```

Second, the parameters for the simulation are set.

```
4 t_sim = 100000.0 # [ms] simulation time
5 n_ex = 16000 # size of exc. population
6 n_in = 4000 # size of inh. population
7 r_ex = 5.0 # [Hz] rate of exc. neurons
8 epsc = 45.0 # [pA] amplitude of exc.
# synaptic currents
9 ipsc = -45.0 # [pA] amplitude of inh.
# synaptic currents
10 d = 1.0 # [ms] synaptic delay
11 lower = 5.0 # [Hz] lower bound of the
# search interval
12 upper = 25.0 # [Hz] upper bound of the
# search interval
13 prec = 0.05 # accuracy goal (in percent
# of inhibitory rate)
```

Third, the nodes are created using `Create()`. Its arguments are the name of the neuron or device model and optionally the number of nodes to create. If the number is not specified, a single node is created. `Create()` returns a list of ids for the new nodes, which we store in variables for later reference.

```
19 neuron = Create("iaf_neuron")
20 noise = Create("poisson_generator", 2)
21 voltmeter = Create("voltmeter")
22 spikedetector = Create("spike_detector")
```

Fourth, the excitatory Poisson generator (`noise[0]`) and the voltmeter are configured using `SetStatus()`, which expects a list of node handles and a list of parameter dictionaries. The rate of the inhibitory Poisson generator is set in line 32. For the neuron and the spike detector we use the default parameters.

```
23 SetStatus([noise[0]], [{"rate": n_ex*r_ex}])
24 SetStatus(voltmeter, [{"interval": 1000.0,
25 "withgid": True}])
```

Fifth, the neuron is connected to the spike detector and the voltmeter, as are the two Poisson generators to the neuron:

```
26 Connect(neuron, spikedetector)
27 Connect(voltmeter, neuron)
28 ConvergentConnect(noise, neuron,
29 [epsc, ipsc], [d, d])
```

The command `Connect()` has different variants. Plain `Connect()` (line 26 and 27) just takes the handles of pre- and postsynaptic nodes and uses the default values for weight and delay. `ConvergentConnect()` (line 28) takes four arguments: A list of presynaptic nodes, a list of postsynaptic nodes, and lists of weights and delays. It connects all presynaptic nodes to each postsynaptic node. All variants of the `Connect()` command reflect the direction of signal flow in the simulation kernel rather than the physical process of inserting an electrode into a neuron. For example, neurons send their spikes to a spike detector, thus the neuron is the

first argument to `Connect()` in line 26. By contrast, a voltmeter polls the membrane potential of a neuron in regular intervals, thus the voltmeter is the first argument of `Connect()` in line 27. The documentation of each model explains the types of events it can send and receive.

To determine the optimal rate of the neurons in the inhibitory population, the network is simulated several times for different values of the inhibitory rate while measuring the rate of the target neuron. This is done until the rate of the inhibitory neurons is determined up to a given relative precision (`prec`), such that the target neuron fires at the same rate as the neurons in the excitatory population. The algorithm is implemented in two steps:

First, the function `output_rate()` is defined to measure the firing rate of the target neuron for a given rate of the inhibitory neurons.

```
30 def output_rate(guess):
31     rate = float(abs(n_in*guess))
32     SetStatus([noise[1]], [{"rate": rate}])
33     SetStatus(spikedetector, [{"n_events": 0}])
34     Simulate(t_sim)
35     n_events = GetStatus(spikedetector,
36                         "n_events")[0]
37     r_target = n_events*1000.0/t_sim
38     print "r_in=%.4f Hz," % guess,
39     print "r_target=%.3f Hz" % r_target
40     return r_target
```

The function takes the firing rate of the inhibitory neurons as an argument. It scales the rate with the size of the inhibitory population (line 31) and configures the inhibitory Poisson generator (`noise[1]`) accordingly (line 32). In line 33, the spike-counter of the spike detector is reset to zero. Line 34 simulates the network using `Simulate()`, which takes the desired simulation time in milliseconds and advances the network state by this amount of time. During the simulation, the spike detector counts the spikes of the target neuron and the total number is read out at the end of the simulation period (line 35). The return value of `output_rate()` is an estimate of the firing rate of the target neuron in Hz.

Second, we determine the optimal firing rate of the neurons of the inhibitory population using the bisection method.

```
41 print "Desired target rate: %.2f Hz" % r_ex
42 r = bisect(lambda x: output_rate(x)-r_ex,
43           lower, upper, rtol=prec)
44 print "Resulting inhibitory rate: %.4f" % r
```

The SciPy function `bisect()` takes four arguments: First a function whose zero crossing is to be determined. Here, the firing rate of the target neuron should equal the firing rate of the neurons of the excitatory population. Thus we define an anonymous function (using `lambda`) that returns the difference between the actual rate of the target neuron and the rate of the excitatory Poisson generator, given a rate for the inhibitory neurons. The next two arguments are the lower and upper bound of the interval in which to search for the zero crossing. The fourth argument of `bisect()` is the desired relative precision of the zero crossing.

Finally, we plot the target neuron's membrane potential as a function of time.

```
45 plot.from_device(voltmeter, timeunit="s")
```

A transcript of the simulation session and the resulting plot are shown in **Figure 1**.

PYNEST ON MULTI-CORE PROCESSORS AND CLUSTERS

NEST has built-in support for parallel and distributed computing (Morrison et al., 2005; Plesser et al., 2007): On multi-core processors, NEST uses POSIX threads (Lewis and Berg, 1997), on computer clusters, NEST uses the Message Passing Interface (MPI; Message Passing Interface Forum, 1994). Nodes and connections are assigned automatically to threads and processes, i.e. the same script can be executed single-threaded, multi-threaded, distributed over multiple processes, or using a combination of both methods. This naturally carries over to PyNEST: To use multiple threads for the simulation, the desired number has to be set prior to the creation of nodes and connections. Note that the network setup is carried out by a single thread, as only a single instance of the Python interpreter exists

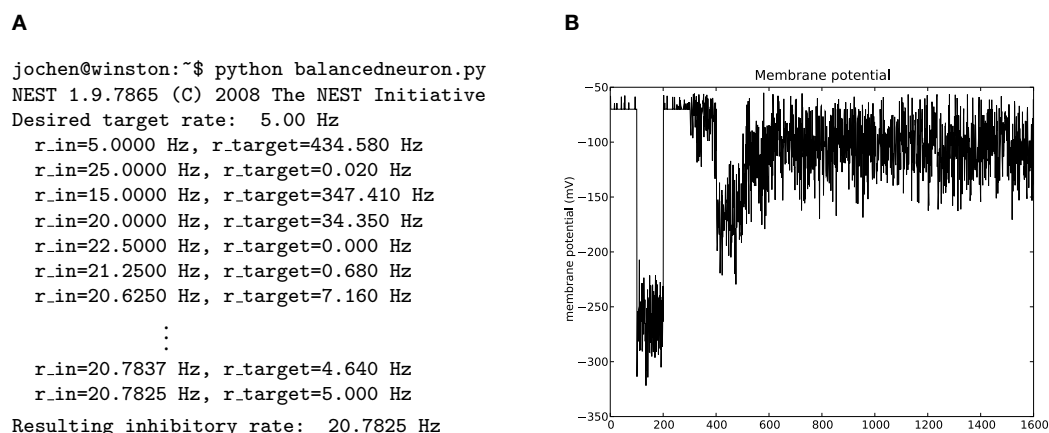


FIGURE 1 | Results of the example simulation. (A) The transcript of the simulation session shows the intermediate results of `r_target` as `bisect()` searches for the optimal rate. **(B)** The membrane potential of the target neuron

as a function of time. Repeated adjustment of the spike rate of the inhibitory population by `bisect()` results in a convergence of the mean membrane potential to -112 mV, corresponding to an output spike rate of 5.0 Hz.

in each process. Only the simulation takes advantage of multiple threads. Distributed simulations can be run via the `mpirun` command of the respective MPI implementation. Where, for SLI, one would execute `mpirun -np n nest simulation.sli` to distribute a simulation onto `n` processes, one has to call `mpirun -np n python simulation.py` to get the same result with PyNEST. In the distributed case, `n` Python interpreters run in parallel and execute the same simulation script. This means that both network setup and simulation are parallelized. With third-party tools like IPython (<http://ipython.scipy.org>) or MPI for Python (<http://mpi4py.scipy.org>), it is possible to use PyNEST interactively even in distributed scenarios. For a more elaborate documentation of parallel and distributed simulations with NEST, see the NEST user manual (<http://www.nest-initiative.org>).

THE INTERFACE BETWEEN PYTHON AND NEST

NEST's built-in simulation language (SLI) is a stack-based language in which functions expect their arguments on an operand stack to which they also return their results. This means that in every expression, the arguments must be entered before the command that uses them (*reverse polish notation*). For many new users, SLI is difficult to learn and hard to read. This is especially true for math: The simple expression $\alpha = t \cdot e^{-t/\tau}$ has to be written as `/alpha t t neg tau div exp mul def` in SLI. But SLI is also a high-level language where functions can be assembled at run time, stored in variables and passed as arguments to other functions (functional programming; Finkel, 1996). Powerful indexing operators like `Part` and functional operators like `Map`, together with data types like heterogeneous arrays and dictionaries, allow a compact and expressive formulation of algorithms.

Stack-based languages are often used as intermediate languages in compilers and interpreters (Aho et al., 1988). This inspired us to couple NEST and Python using SLI as an intermediate language.

THE PyNEST LOW-LEVEL INTERFACE

The low-level API of PyNEST is implemented in C/C++ using the Python C-API (van Rossum, 2008). It exposes only three functions to Python, and has private routines for converting between SLI data types and their Python equivalents. The exposed functions are:

1. `sli_push(py_object)`, which converts the Python object `py_object` to the corresponding SLI data type and pushes it onto SLI's operand stack.
2. `sli_pop()`, which removes the top element from SLI's operand stack and returns it as a Python object.
3. `sli_run(slicommand)`, which uses NEST's simulation language interpreter to execute the string `slicommand`. If the command requires arguments, they have to be present on SLI's operand stack or must be part of `slicommand`. After the command is executed, its return values will be on the interpreter's operand stack.

Since these functions provide full access to the simulation language interpreter, we can now control NEST's simulation kernel without explicit Python bindings for all NEST functions. This interface also provides a natural way to execute legacy SLI code

from within a PyNEST script by just using the command `sli_run("(legacy.sli) run")`. However, it does not provide any benefits over plain SLI from a syntactic point of view: All simulation specific code still has to be written in SLI. This problem is solved by a set of high-level functions.

THE PyNEST HIGH-LEVEL INTERFACE

To allow the researcher to define, run and evaluate NEST simulations using only Python, PyNEST offers convenient wrappers for the most important functions of NEST. These wrappers are implemented on top of the low-level API and execute appropriate SLI expressions. Thus, at the level of PyNEST, SLI is invisible to the user. Each high-level function consists essentially of three parts:

1. The arguments of the function are put on SLI's operand stack.
2. One or more SLI commands are executed to perform the desired action in NEST.
3. The results (if any) are fetched from the operand stack and returned as Python objects.

A concrete example of the procedure is given in the following listing, which shows the implementation of `Create()`:

```
1 def Create(model, n=1):
2     sli_run("/%s" % model)
3     sli_push(n)
4     sli_run("CreateMany")
5     lastid = sli_pop()
6     return range(lastid - n + 1, lastid + 1)
```

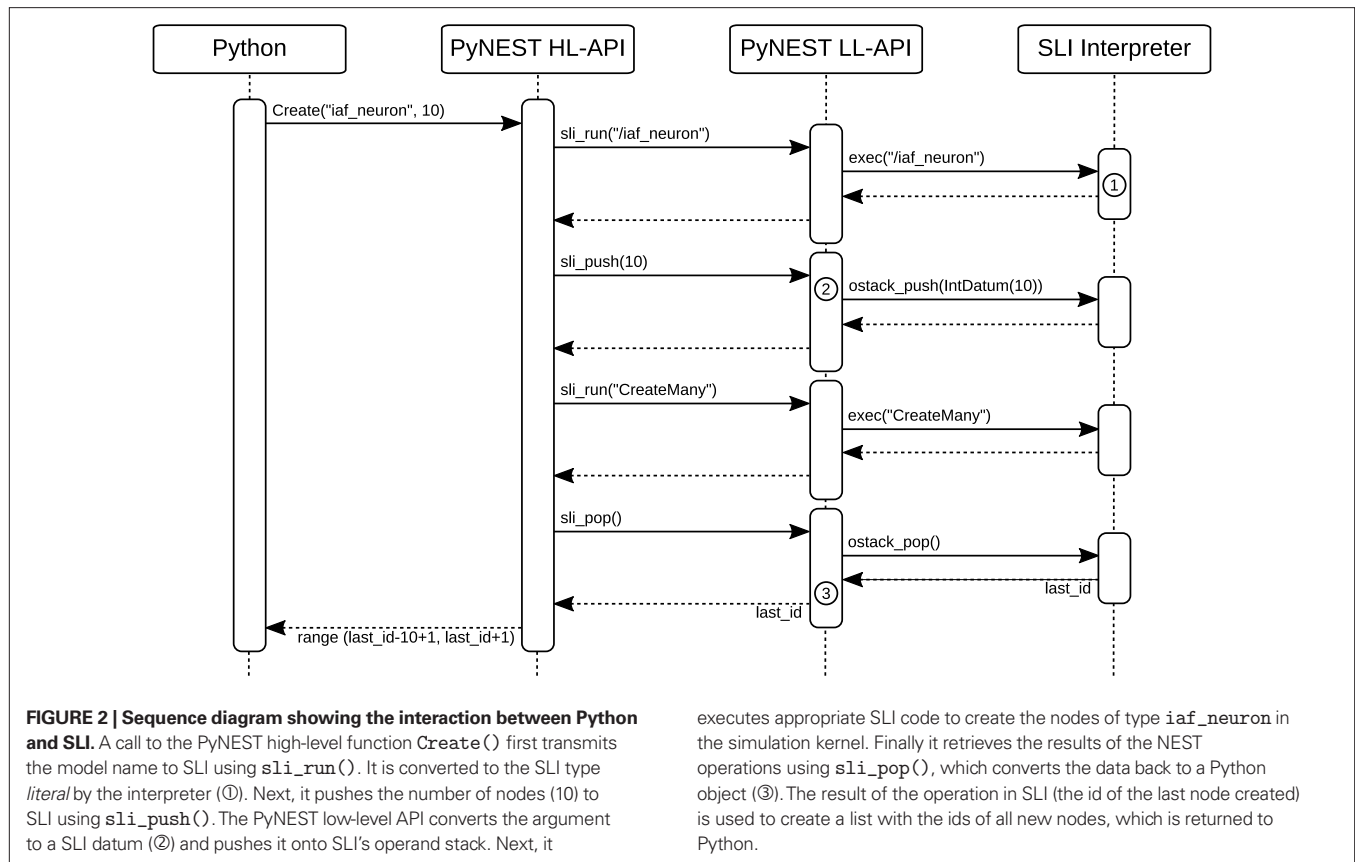
In line 2, we first transfer the model name to NEST. Model names in NEST have to be of type *literal*, a special symbol type that is not available in Python. Because of this, we cannot use `sli_push()` for the data transfer, but have to use `sli_run()`, which executes a given command string instead of just pushing it onto SLI's stack. The command string consists of a slash followed by the model name, which is interpreted as a literal by SLI. Line 3 uses `sli_push()` to transmit the number of nodes (`n`) to SLI. The nodes are then created by `CreateMany` in line 4, which expects the model name and number of nodes on SLI's operand stack and puts the id of the last created node back onto the stack. The id is retrieved in line 5 via `sli_pop()`. To be consistent with the convention that all PyNEST functions work with lists of nodes, we build a list of all created nodes' ids, which is returned in line 6.

A sequence diagram of the interaction between the different software layers of PyNEST is shown in **Figure 2** for a call to the `Create()` function.

DATA CONVERSION

From Python to SLI

The data conversion between Python and SLI exploits the fact that most data types in SLI have an equivalent type in Python. The function `sli_push()` calls `PyObjectToDatum()` to convert a Python object `py_object` to the corresponding SLI data type (see ② in **Figure 2**). `PyObjectToDatum()` determines the type of `py_object` in a cascade of type checks (e.g. `PyInt_Check()`, `PyString_Check()`, `PyFloatCheck()`) as described by van Rossum (2008). If a type check succeeds, the Python object is used to create a new



SLI Datum of the respective type. `PyObjectToDatum()` is called recursively on the elements of lists and dictionaries. The listing below shows how this technique is used for the conversion of the Python type `float` and for NumPy arrays of doubles:

```

1 Datum* PyObjectToDatum(PyObject *py_object)
2 {
3     if (PyFloat_Check(py_object)) //float?
4     {
5         return new DoubleDatum(PyFloat_AsDouble(
6             py_object));
7     }
8
9     if (PyArray_Check(py_object)) //NumPy array?
10    {
11        int size = PyArray_Size(py_object);
12        PyArrayObject *array;
13        array = (PyArrayObject*) py_object;
14        assert(array != 0);
15        switch (array->descr->type_num)
16        {
17            case PyArray_DOUBLE:
18            {
19                double *begin = (double*) array->data;
20                return new DoubleVectorDatum(
21                    new std::vector<double>(
22                        begin, begin+size));
23            }
24            //cases for NumPy arrays of other types
25        }
26    }
27 }

```

```

26     }
27     //checks for other supported Python types
28 }

```

From SLI to Python

To convert a SLI data type to the corresponding Python type, we can avoid the cascade of type checks, since all SLI data types are derived from a common base class, called `Datum`. The C++ textbook solution would add a pure virtual conversion function `convert()` to the class `Datum`. Each derived class (e.g. `DoubleDatum`, `DoubleVectorDatum`) then overloads this function to implement its own conversion to the corresponding Python type. This approach is shown for the SLI type `DoubleDatum` in the following listing. The function `get()` is implemented in each `Datum` and returns its data member.

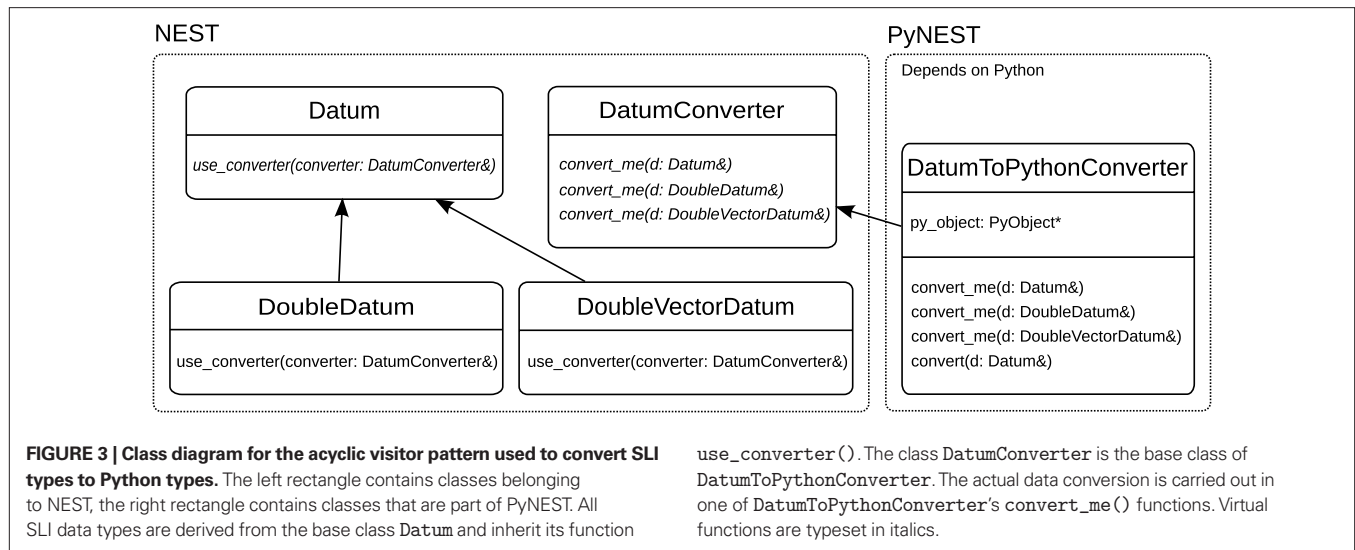
```

1 PyObject*
2 DoubleDatum::convert()
3 {
4     return PyFloat_FromDouble(get());
5 }

```

However, this solution would make SLI's type hierarchy (and thus NEST) depend on Python. To keep NEST independent of Python, we split the implementation in two parts: The first is Python-unspecific and resides in the NEST source code (Figure 3, left rectangle), the second is Python-specific and defined in the PyNEST source code (Figure 3, right rectangle).

We move the Python-specific conversion code from `convert()` to a new function `convert_me()`, which is then called by the



interface function `use_converter()`. This function is now independent of Python:

```
1 void
2 Datum::use_converter(DatumConverter& converter)
3 {
4     converter.convert_me(*this);
5 }
```

The function `use_converter()` is defined in the base class `Datum` and inherited by all derived classes. It calls the `convert_me()` function of `converter` that matches the type of the derived `Datum`. NEST's class `DatumConverter` is an abstract class that defines a pure virtual function `convert_me(T&)` for each SLI type `T`:

```
1 class DatumConverter
2 {
3 public:
4     virtual void convert_me(Datum&);
5     virtual void convert_me(DoubleDatum&)=0;
6     virtual void convert_me(DoubleVectorDatum&)=0;
7     //convert_me() function for other Datums
8 };
```

The Python-specific part of the conversion is encapsulated in the class `DatumToPythonConverter`, which derives from `DatumConverter` and implements the `convert_me()` functions to actually convert the SLI types to Python objects. `DatumToPythonConverter::convert_me()` takes a reference to the `Datum` as an argument and is overloaded for each SLI type. It stores the result of the conversion in the class variable `py_object`. An example for the conversion of `DoubleDatum` is given in the following listing:

```
1 void
2 DatumToPythonConverter::convert_me(
3     DoubleDatum& dd)
4 {
5     py_object = PyFloat_FromDouble(dd.get());
6 }
```

`use_converter()`. The class `DatumConverter` is the base class of `DatumToPythonConverter`. The actual data conversion is carried out in one of `DatumToPythonConverter`'s `convert_me()` functions. Virtual functions are typeset in italics.

`DatumToPythonConverter` also provides the function `convert()`, which converts a given `Datum d` to a Python object by calling `d.use_converter()` with itself as an argument. It is used in the implementation of `slipop()` (see ③ in Figure 2). After the call to `use_converter()`, the result of the conversion is available in the member variable `py_object`, and is returned to the caller:

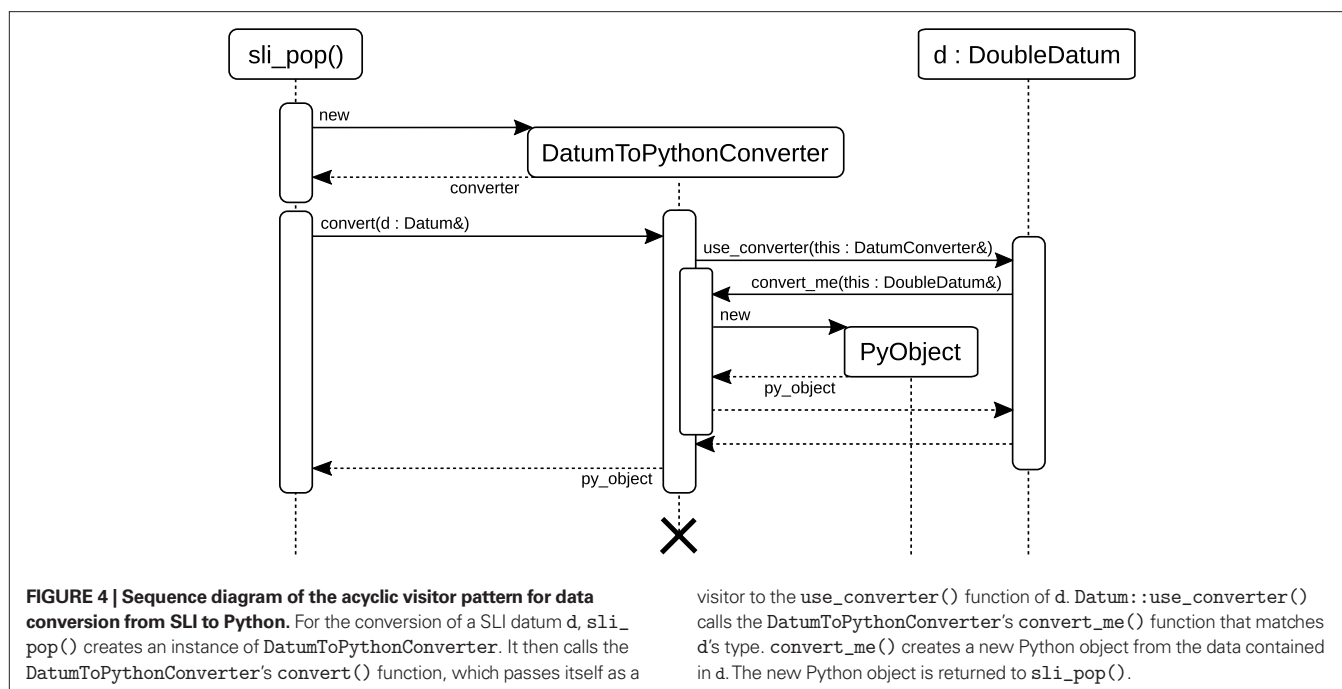
```
1 PyObject*
2 DatumToPythonConverter::convert(Datum& d)
3 {
4     d.use_converter(*this);
5     return py_object;
6 }
```

In the Computer Science literature, this method of decoupling different parts of program code is called the *acyclic visitor pattern* (Martin et al., 1998). Our implementation is based on Alexandrescu (2001).

As an example, the diagram in Figure 4 illustrates the sequence of events in `slipop()`: First, `slipop()` retrieves a SLI `Datum d` from the operand stack (not shown). Second, it creates an instance of `DatumToPythonConverter` and calls its `convert()` function, which then passes itself as visitor to the `use_converter()` function of `d`. `Datum::use_converter()` calls the `DatumToPythonConverter`'s `convert_me()` function that matches the type of `d`. The function `convert_me()` then creates a new Python object from the data in `d` and stores it in the `DatumToPythonConverter`'s member variable `py_object`, which is returned to `slipop()`.

NumPy support

To make PyNEST depend on NumPy only if it is available, we use conditional compilation based on the preprocessor macro `HAVE_NUMPY`, which is determined during the configuration of PyNEST prior to compilation. For example, the following listing shows the implementation of the `DatumToPythonConverter::convert_me()` function to convert homogeneous arrays of doubles from SLI to Python. If NumPy is available during compilation, its



homogeneous array type is used to store the data. Without NumPy, a Python list is used instead.

```

1 void
2 DatumToPythonConverter::convert_me(
3     DoubleVectorDatum& d)
4 {
5     int dims = d->size();
6     #ifdef HAVE_NUMPY
7     PyArrayObject* array;
8     array = (PyArrayObject*)
9         PyArray_FromDims(1, &dims, PyArray_DOUBLE);
10    std::copy(d->begin(), d->end(),
11        (double*) array->data);
12    py_object = (PyObject*) array;
13 #else
14    py_object = PyList_New(dims);
15    for(int i=0; i<dims; i++)
16        PyList_SetItem(py_object, i,
17            PyFloat_FromDouble((*d)[i]));
18 #endif
19 }

```

ERROR HANDLING

Error handling in NEST is implemented using C++ exceptions that are propagated up the calling hierarchy until a suitable error handler catches them. In this section, we describe how we extend this strategy to PyNEST.

PyNEST executes SLI code using *sli_run()* as described in the Section “The PyNEST High-Level Interface”. However, the high-level API does not call *sli_run()* directly, but rather through the wrapper function *catching_sr()*:

```

1 def catching_sr(cmd):
2     sli_run("{ " + cmd + " } runprotected")
3     if not sli_pop(): #cmd caused an error

```

```

4     errorname = sli_pop()
5     commandname = sli_pop()
6     raise NESTError("NEST error: " +
7         errorname + " in " +
8         commandname)

```

In line 2, *catching_sr()* converts the command string *cmd* to a SLI procedure by adding braces. It then calls the SLI command *runprotected* (see listing below), which executes the procedure in a stopped context (PostScript; Adobe Systems Inc., 1999). If an error occurs, *stopped* leaves the name of the failed command on the stack and returns true. In this case, *runprotected* extracts the name of the error from SLI's error dictionary, converts it to a string, and puts it back on the operand stack, followed by false to indicate the error condition to the caller. Otherwise, true is put on the stack. In case of an error, *catching_sr()* uses both the name of the command and the error to raise a Python exception (*NESTError*), which can be handled by the user's simulation code. The following listing shows the implementation of *runprotected*:

```

1 /runprotected
2 {
3     stopped dup
4     {
5         errordict /commandname get cvs
6         % tell NEST that the error was handled
7         errordict /newerror false put
8     } if
9     not
10 } def

```

Forwarding the original NEST errors to Python has the advantage that PyNEST functions do not have to check their arguments, because the underlying NEST functions already do. This makes the code of the high-level API more readable, while at the same time, errors are raised as Python exceptions without requiring additional

code. Moreover, this results in consistent error messages in NEST and PyNEST.

DISCUSSION

The previous sections describe the usage and implementation of PyNEST. Here we discuss consequences and limitations of the PyNEST implementation.

PERFORMANCE

The use of PyNEST entails a certain computational overhead over pure SLI-operated NEST. This overhead can be split into two main components:

1. Call overhead because of using SLI over direct access to the NEST kernel.
2. Data exchange between Python and NEST.

For most real-world simulations, the first is negligible, since the number of additional function calls is small. In practice, most overhead is caused by the second component, which we can reduce by minimizing the number of data conversions. For an illustration of the technique, see the following two listings that both add up a sequence of numbers in SLI. The first creates the sequence of numbers in Python, pushes them to SLI one after the other and lets SLI add them. Executing it takes approx. 15 s on a laptop with an Intel Core Duo processor at 1.83 GHz.

```
1 sli_push(0)
2 for i in range(1, 100001):
3     sli_push(i)
4     sli_run("add")
```

The second version computes the same result, but instead of creating the sequence in Python, it is created in SLI:

```
1 sli_run("0 1 1 100000 { add } for")
```

Although Python loops are about twice as fast as SLI loops, this version takes only 0.6 s, because of the reduced number of data conversions and, to a minor extent, the repeated parsing of the command string and the larger number of function calls in the first version.

The above technique is used in the implementation of the PyNEST high-level API wherever possible. The same technique is also applied for other loop-like commands (e.g. Map) that exist in both interpreters. However, it is important to note that the total run time of the simulation is often dominated by the actual creation and update of nodes and synapses, and by event delivery. These tasks take place inside of the optimized C++ code of NEST's simulation kernel, hence the choice between SLI or Python has no impact on performance.

INDEPENDENCE

One of the design decisions for PyNEST was to keep NEST independent of third-party software. This is important because NEST is used on architectures, where Python is not available or only available as a minimal installation. Moreover, since NEST is a long term project that has already seen several scripting languages and graphics libraries coming and going, we do not want to introduce a hard dependency on one or the other. The stand-alone version of NEST

can be compiled without any third-party libraries. Likewise, the implementation of PyNEST does not depend on anything except Python itself. The use of NumPy is recommended, but optional. The binary part of the interface is written by hand and does not depend on interface generators like SWIG (<http://www.swig.org>) or third-party libraries like Boost.Python (<http://www.boost.org>). In our opinion, this strategy is important for the long-term sustainability of our scientific software.

EXTENSIBILITY

NEST can never provide all models and functions needed by every researcher. Extensibility is hence important.

Due to the asymmetry of the PyNEST interface (see "Asymmetry of the Interface"), neuron models, devices and synapse models have to be implemented in C++, the language of the simulation kernel. However, new analysis functions and connection routines can be implemented in either Python, SLI or C++, depending on the performance required and the skills of the user. The implementation in Python is easy, but performance may be limited. However, this approach is safe, as the real functionality is performed by SLI code, which is often well tested. To improve the performance, the implementation can be translated to SLI. This requires knowledge of SLI in addition to Python. Migrating the function down to the C++ level yields the highest performance gain, but requires knowledge of C++ and the internals of the simulation kernel.

Since the user can choose between three languages, it is easy to extend PyNEST, while at the same time, it is possible to achieve high performance if necessary. The hierarchy of languages also provides abstraction layers, which make it possible to migrate the implementation of a function between the different languages, without affecting user code. The intermediate layer of SLI allows the decoupling of the development of the simulation kernel from the development of the PyNEST API. This is also helpful for developers of abstraction libraries like PyNN (Davison et al., 2008), who only need limited knowledge of the simulation kernel.

ASYMMETRY OF THE INTERFACE

Our implementation of PyNEST is asymmetric in that SLI code can be executed from Python, but NEST cannot respond, except for error handling and data exchange. Although this is sufficient to run NEST simulations from within a Python session, it could be beneficial to allow NEST to execute Python code: The user of PyNEST already knows the Python programming language, hence it might be easier to extend NEST in Python rather than to modify the C++ code of the simulation kernel. SciPy, NumPy and other packages provide well tested implementations of mathematical functions and numerical algorithms. Together with callback functions, these libraries would allow rapid prototyping of neuron and synapse models or to initialize parameters of neuron models or synapses according to complicated probability distributions: Python could be the middleware between NEST's simulation kernel and the numerical package. Using online feedback from the simulation, callback functions could also control simulations. Moreover, with a symmetric interface and appropriate Python modules it would be easier to add graphical user interfaces to NEST, along with online display of observables, and experiment management.

Different implementations of the symmetric interface are possible: One option is to pass callback functions from Python to NEST. Another option is to further exploit the idea that the “language is the protocol”. In the same way as PyNEST generates SLI code, NEST would emit code for Python. Already Harrison and McLennan (1998) mention this technique, and in experimental implementations it was used successfully to symmetrically couple NEST with Tcl/Tk (Diesmann and Gewaltig, 2002), Mathematica, Matlab and IDL. The fact that none of these interfaces is still maintained confirms the conclusions of the Section “Independence”.

LANGUAGE CONSIDERATIONS

At present, PyNEST maps NEST’s capabilities to Python. Further advances in the expressiveness of the language may be easier to achieve at the level of Python or above (e.g. PyNN; Davison et al., 2008) without a counterpart in SLI. An example for this is the support of units for physical quantities as available in SBML (Hucka et al., 2002) or Brian (Goodman and Brette, 2008).

More generally, the development of simulation tools has not kept up with the increasing complexity of network models. As a consequence the reliable documentation of simulation studies is challenging and laboratories notoriously have difficulties in reproducing published results (Djurfeldt and Lansner, 2007). One component of a solution is the ability to concisely formulate simulations in terms of the neuroscientific problem domain like connection topologies and probability distributions. At present little research has been carried out on the particular design of such a language (Davison et al., 2008; Nordlie et al., 2008), but a general purpose high-level language interface to the simulation engine is a first step towards this goal.

APPENDIX

A. PyNEST API REFERENCE

Models

Models(*mtype*="all", *sel*=None): Return a list of all available models (nodes and synapses). Use *mtype*="nodes" to only see node models, *mtype*="synapses" to only see synapse models. *sel* can be a string, used to filter the result list and only return models containing it.

GetDefaults(*model*): Return a dictionary with the default parameters of the given *model*, specified by a string.

SetDefaults(*model*, *params*): Set the default parameters of the given *model* to the values specified in the *params* dictionary.

GetStatus(*model*, *keys*=None): Return a dictionary with status information for the given *model*. If *keys* is given, a value is returned instead. *keys* may also be a list, in which case a list of values is returned.

CopyModel(*existing*, *new*, *params*=None): Create a new model by copying an existing one. Default parameters can be given as *params*, or else are taken from *existing*.

Nodes

Create(*model*, *n*=1, *params*=None): Create *n* instances of type *model* in the current subnetwork. Parameters for the new nodes can be given as *params* (a single dictionary, or a list of dictionaries with size *n*). If omitted, the *model*’s defaults are used.

GetStatus(*nodes*, *keys*=None): Return a list of parameter dictionaries for the given list of nodes. If *keys* is given, a list

of values is returned instead. *keys* may also be a list, in which case the returned list contains lists of values.

SetStatus(*nodes*, *params*, *val*=None): Set the parameters of the given nodes to *params*, which may be a single dictionary, or a list of dictionaries of the same size as *nodes*. If *val* is given, *params* has to be the name of a property, which is set to *val* on the nodes. *val* can be a single value, or a list of the same size as *nodes*.

Connections

Connect(*pre*, *post*, *params*=None, *delay*=None, *model*="static_synapse"): Make one-to-one connections of type *model* between the nodes in *pre* and the nodes in *post*. *pre* and *post* have to be lists of the same length. If *params* is given (as a dictionary or as a list of dictionaries with the same size as *pre* and *post*), they are used as parameters for the connections. If *params* is given as a single float, or as a list of floats of the same size as *pre* and *post*, it is interpreted as weight. In this case, *delay* also has to be given (as a float, or as a list of floats with the same size as *pre* and *post*).

ConvergentConnect(*pre*, *post*, *weight*=None, *delay*=None, *model*="static_synapse"): Connect all nodes in *pre* to each node in *post* with connections of type *model*. If *weight* is given, *delay* also has to be given. Both can be specified as a float, or as a list of floats with the same size as *pre*.

RandomConvergentConnect(*pre*, *post*, *n*, *weight*=None, *delay*=None, *model*="static_synapse"): Connect *n* randomly selected nodes from *pre* to each node in *post* with connections of type *model*. Presynaptic nodes are drawn independently for each postsynaptic node. If *weight* is given, *delay* also has to be given. Both can be specified as a float, or as a list of floats of size *n*.

DivergentConnect(*pre*, *post*, *weight*=None, *delay*=None, *model*="static_synapse"): Connect each node in *pre* to all nodes in *post* with connections of type *model*. If *weight* is given, *delay* also has to be given. Both can be specified as a float, or as a list of floats with the same size as *post*.

RandomDivergentConnect(*pre*, *post*, *n*, *weight*=None, *delay*=None, *model*="static_synapse"): Connect each node in *pre* to *n* randomly selected nodes from *post* with connections of type *model*. If *weight* is given, *delay* also has to be given. Both can be specified as a float, or as a list of floats of size *n*.

Structured networks

CurrentSubnet(): Return the id of the current subnetwork.

ChangeSubnet(*subnet*): Make *subnet* the current subnetwork.

GetLeaves(*subnet*): Return the ids of all nodes under *subnet* that are not subnetworks.

GetNodes(*subnet*): Return the complete list of *subnet*’s children (including subnetworks).

GetNetwork(*subnet*, *depth*): Return a nested list of *subnet*’s children up to *depth* (including subnetworks).

LayoutNetwork(*model*, *shape*, *label*=None, *customdict*=None): Create a subnetwork of shape *shape* that contains nodes of type *model*. *label* is an optional name for the subnetwork. If present, *customdict* is set as custom dictionary of

the subnetwork, which can be used by the user to store custom information.

BeginSubnet(label=None, customdict=None): Create a new subnetwork and change into it. *label* is an optional name for the subnetwork. If present, *customdict* is set as custom dictionary of the subnetwork, which can be used by the user to store custom information.

EndSubnet(): Change to the parent subnetwork and return the id of the subnetwork just left.

Simulation control

Simulate(t): Simulate the network for *t* milliseconds.

ResetKernel(): Reset the simulation kernel. This will destroy the network as well as all custom models created with `CopyModel()`. The parameters of built-in models are reset to their defaults. Calling this function is equivalent to restarting NEST.

ResetNetwork(): Reset all nodes and connections to the defaults of their respective model.

SetKernelStatus(params): Set the parameters of the simulation kernel to the ones given in *params*.

GetKernelStatus(): Return a dictionary with the parameters of the simulation kernel.

PrintNetwork(depth=1, subnet=None): Print the network tree up to *depth*, starting at *subnet*. If *subnet* is omitted, the current subnetwork is used instead.

B. ADVANCED EXAMPLE

In the Section “Using PyNEST”, we introduced the main features of PyNEST with a short example. This section contains a simulation of a balanced random network of 10,000 excitatory and 2,500 inhibitory integrate-and-fire neurons as described in Brunel (2000). We start with importing the required modules.

```
1 from nest import *
2 import nest.raster_plot as plot
3 import time
```

We store the current time at the start of the simulation.

```
4 startbuild = time.time()
```

Next, we use `SetKernelStatus()` to set the temporal resolution for the simulation to 0.1 ms.

```
5 SetKernelStatus({"resolution": 0.1})
```

We define variables for the simulation duration, the network size and the number of neurons to be recorded.

```
6 simtime = 500.0 # [ms] Simulation time
7 NE = 10000 # number of exc. neurons
8 NI = 2500 # number of inh. neurons
9 N_rec = 50 # record from 50 neurons
```

The following are the parameters of the integrate-and-fire neuron that deviate from the defaults.

```
10 tauMem = 20.0 # [ms] membrane time constant
11 theta = 20.0 # [mV] threshold for firing
12 t_ref = 2.0 # [ms] refractory period
13 E_L = 0.0 # [mV] resting potential
```

The synaptic delay and weights and the number of afferent synapses per neuron are assigned to variables. By choosing the relative

strength of inhibitory connections to be $|J_{in}| / |J_{ex}| = g = 5.0$, the network is in the inhibition-dominated regime.

```
14 delay = 1.5 # [ms] synaptic delay
15 J_ex = 0.1 # [mV] exc. synaptic strength
16 g = 5.0 # ratio between inh. and exc.
17 J_in = -g * J_ex # [mV] inh. synaptic strength
18 epsilon = 0.1 # connection probability
19 CE = int(epsilon * NE) # exc. synapses/neuron
20 CI = int(epsilon * NI) # inh. synapses/neuron
```

To reproduce Figure 8C from Brunel (2000), we choose parameters for asynchronous, irregular firing: v_0 denotes the external Poisson rate which results in a mean free membrane potential equal to the threshold. We set the rate of the external Poisson input to $v_{ext} = \eta v_0 = 2v_0$.

```
21 eta = 2.0 # fraction of ext. input
22 nu_th = theta / (J_ex * tauMem) # [kHz] ext. rate
23 nu_ext = eta * nu_th # [kHz] exc. ext. rate
24 p_rate = 1000.0 * nu_ext # [Hz] ext. Poisson rate
```

In the next step we set up the populations of excitatory (*nodes_ex*) and inhibitory (*nodes_in*) neurons. The neurons of both pools have identical parameters, which are configured for the model with `SetDefaults()`, before creating instances with `Create()`.

```
25 print "Creating network nodes ..."
26 SetDefaults("iaf_psc_delta", {"C_m": tauMem,
27                               "tau_m": tauMem,
28                               "t_ref": t_ref,
29                               "E_L": E_L,
30                               "V_th": theta})
31 nodes_ex = Create("iaf_psc_delta", NE)
32 nodes_in = Create("iaf_psc_delta", NI)
33 nodes = nodes_ex + nodes_in
```

Next, a Poisson spike generator (*noise*) is created and its rate is set. We use it to provide external excitatory input to the network.

```
34 noise = Create("poisson_generator",
35               params={"rate": p_rate})
```

The next paragraph creates the devices for recording spikes from the excitatory and inhibitory population. The spike detectors are configured to record the spike times and the id of the sending neuron to a file.

```
36 SetDefaults("spike_detector", {"withtime": True,
37                                "withgid": True,
38                                "to_file": True})
39 espikes = Create("spike_detector")
40 ispikes = Create("spike_detector")
```

Next, we use `CopyModel()` to create copies of the synapse model “static_synapse”, which are used for the excitatory and inhibitory connections.

```
41 SetDefaults("static_synapse", {"delay": delay})
42 CopyModel("static_synapse", "excitatory",
43           {"weight": J_ex})
44 CopyModel("static_synapse", "inhibitory",
45           {"weight": J_in})
```

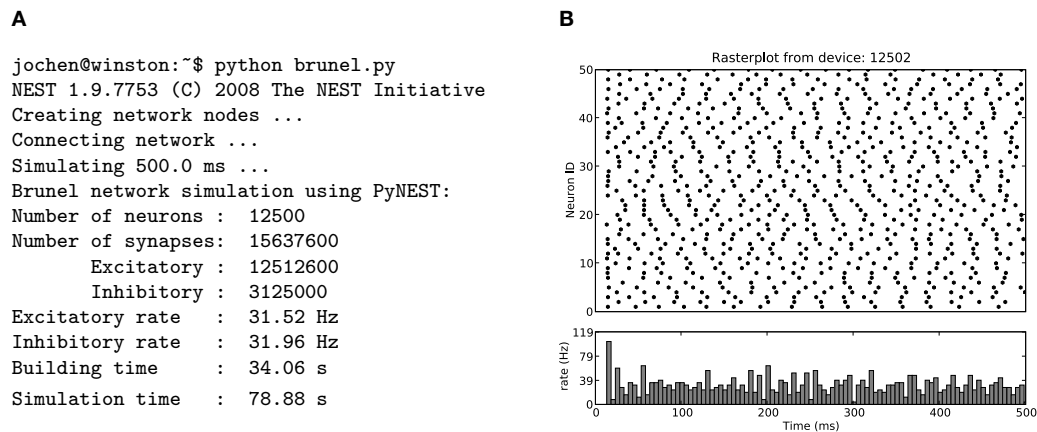


FIGURE 5 | Results of the balanced random network simulation. (A) The transcript of the simulation session shows the output during network setup and

the summary printed at the end of the simulation. **(B)** Spike raster (top) and spike time histogram (bottom) of the N_{rec} recorded excitatory neurons.

The following code connects neurons and devices. `DivergentConnect()` connects one source node with each of the given target nodes and is used to connect the Poisson generator (noise) to the excitatory and the inhibitory neurons (nodes). `ConvergentConnect()` is used to connect the first N_{rec} excitatory and inhibitory neurons to the corresponding spike detectors.

```
46 print "Connecting network ..."
47 DivergentConnect(noise, nodes,
48                 model="excitatory")
49 ConvergentConnect(nodes_ex[:N_rec], espikes,
50                 model="excitatory")
51 ConvergentConnect(nodes_in[:N_rec], ispikes,
52                 model="excitatory")
```

The following lines connect the neurons with each other. The function `RandomConvergentConnect()` draws CE presynaptic neurons randomly from the given list (first argument) and connects them to each postsynaptic neuron (second argument). The presynaptic neurons are drawn repeatedly and independent for each postsynaptic neuron.

```
53 RandomConvergentConnect(nodes_ex, nodes, CE,
54                         model="excitatory")
55 RandomConvergentConnect(nodes_in, nodes, CI,
56                         model="inhibitory")
```

To calculate the duration of the network setup later, we again store the current time.

```
57 endbuild = time.time()
```

We use `Simulate()` to run the simulation.

```
58 print "Simulating", simtime, "ms ..."
59 Simulate(simtime)
```

Again, we store the time to calculate the runtime of the simulation later.

```
60 endsimulate = time.time()
```

The following code calculates the mean firing rate of the excitatory and the inhibitory neurons, determines the total number of

synapses, and the time needed to set up the network and to simulate it. The firing rates are calculated from the total number of events received by the spike detectors. The total number of synapses is available from the status dictionary of the respective synapse models.

```
61 events_ex = GetStatus(espikes, "n_events")[0]
62 rate_ex    = event_ex/simtime*1000.0/N_rec
63 events_in = GetStatus(ispikes, "n_events")[0]
64 rate_in    = events_in/simtime*1000.0/N_rec
65 synapses_ex = GetStatus("excitatory",
66                        "num_connections")
67 synapses_in = GetStatus("inhibitory",
68                        "num_connections")
69 synapses    = synapses_ex+synapses_in
70 build_time  = endbuild-startbuild
71 sim_time    = endsimulate-endbuild
```

The next lines print a summary with network and runtime statistics.

```
72 print "Brunel network simulation using PyNEST:"
73 print "Number of neurons :", len(nodes)
74 print "Number of synapses:", synapses
75 print "    Excitatory   :", synapses_ex
76 print "    Inhibitory   :", synapses_in
77 print "Excitatory rate  : %.2f Hz" % rate_ex
78 print "Inhibitory rate  : %.2f Hz" % rate_in
79 print "Building time    : %.2f s" % build_time
80 print "Simulation time  : %.2f s" % sim_time
```

Finally, `nest.raster_plot` is used to visualize the spikes of the N_{rec} selected excitatory neurons, similar to Figure 8C of Brunel (2000).

```
81 plot.from_device(espikes, hist=True)
```

The resulting plot is shown in **Figure 5** together with a transcript of the simulation session. The simulation was run on a laptop with an Intel Core Duo processor at 1.83 GHz and 1.5 GB of RAM.

ACKNOWLEDGMENTS

We are grateful to our colleagues in the NEST Initiative and the FACETS project for stimulating discussions, in particular to Hans

Ekkehard Plesser for drawing our attention to the visitor pattern. Partially funded by DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grant 15879 (FACETS), and “The Next-Generation Integrated

Simulation of Living Matter” project, part of the Development and Use of the Next-Generation Supercomputer Project of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

REFERENCES

- Adobe Systems Inc. (1999). Postscript Language Reference Manual, third edn. Reading, MA, Addison-Wesley.
- Aho, A. V., Sethi, R., and Ullman, J. D. (1988). Compilers, Principles, Techniques, and Tools. Reading, MA, Addison-Wesley.
- Alexandrescu, A. (2001). Modern C++ Design. Boston, Addison-Wesley.
- Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208.
- Davison, A., Brüderle, D., Eppler, J. M., Kremkow, J., Müller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinformatics* 2. doi: 10.3389/neuro.11.011.2008.
- Diesmann, M., and Gewaltig, M.-O. (2002). NEST: an environment for neural systems simulations. In *Forschung und wissenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, Vol. 58 of GWDG-Bericht, T. Plesser and V. Macho, eds (Gottingen, Ges. für Wiss. Datenverarbeitung), pp. 43–70.
- Djurfeldt, M., and Lansner, A. (2007). Workshop report: 1st INCF workshop on large-scale modeling of the nervous system. *Nature Precedings*, doi: 10.1038/npre.2007.262.1.
- Dubois, P. F. (2007). Guest editor's introduction: Python: batteries included. *Comput. Sci. Eng.* 9, 7–9.
- Finkel, R. A. (1996). Advanced Programming Languages. Menlo Park, CA, Addison-Wesley.
- Gewaltig, M.-O., and Diesmann, M. (2007). NEST (Neural Simulation Tool). *Scholarpedia* 2, 1430.
- Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinformatics* 2. doi: 10.3389/neuro.11.005.2008.
- Harrison, M., and McLennan, M. (1998). Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk. Reading, MA, Addison-Wesley.
- 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. et al. (2002). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531.
- Lewis, B., and Berg, D. J. (1997). Multithreaded Programming With PThreads. Upper Saddle River: Sun Microsystems Press.
- Martin, R. C., Riehle, D., and Buschmann, F. (eds) (1998). Pattern Languages of Program Design 3. Reading, MA, Addison-Wesley.
- MathWorks (2002). MATLAB The Language of Technical Computing: Using MATLAB. Natick, MA, 3 Apple Hill Drive.
- McConnell, S. (2004). Code Complete: A practical Handbook of Software Construction. 2nd edn. Redmond, WA, Microsoft Press.
- Message Passing Interface Forum (1994). MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230.
- Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike-timing. *Biol. Cybern.* 98, 459–478.
- Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.
- Natschläger, T. (2003). CSIM: A Neural Circuit Simulator. Technical report.
- Nordlie, E., Plesser, H. E., and Gewaltig, M.-O. (2008). Towards reproducible descriptions of neuronal network models. Volume Conference Abstract: Neuroinformatics 2008. doi: 10.3389/conf.neuro.11.2008.01.086.
- Ousterhout, J. K. (1994). Tcl and the Tk Toolkit. Professional Computing. Reading Massachusetts: Addison-Wesley.
- Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In Euro-Par 2007: Parallel Processing, Volume 4641 of Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bouge, and T. Priol, eds (Berlin, Springer-Verlag), pp. 672–681.
- Prechelt, L. (2000). An empirical comparison of seven programming languages. *COMPUTER* 33, 23–29.
- Stroustrup, B. (1997). The C++ Programming Language, 3rd edn. New York, Addison-Wesley.
- van Rossum, G. (2008). Python/C API Reference Manual. Available at: <http://docs.python.org/api/api.html>.

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

Received: 14 September 2008; paper pending published: 29 September 2008; accepted: 30 December 2008; published online: 29 January 2009.

Citation: Eppler JM, Helias M, Müller E, Diesmann M and Gewaltig M-O (2009) PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* (2009) 2:12. doi: 10.3389/neuro.11.012.2008

Copyright © 2009 Eppler, Helias, Müller, Diesmann and Gewaltig. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



NEURON and Python

Michael L. Hines¹, Andrew P. Davison^{2*} and Eilif Muller³

¹ Computer Science, Yale University, New Haven, CT, USA

² Unité de Neurosciences Intégratives et Computationnelles, CNRS, Gif sur Yvette, France

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

Edited by:

Rolf Kötter, Radboud University,
Nijmegen, The Netherlands

Reviewed by:

Felix Schürmann, Ecole Polytechnique
Fédérale de Lausanne, Switzerland

Volker Steuber, University of
Hertfordshire, UK

Arnd Roth, University College London,
UK

*Correspondence:

Andrew Davison, UNIC, Bât. 32/33,
CNRS, 1 Avenue de la Terrasse, 91198
Gif sur Yvette, France.

e-mail: andrew.davison@unic.cnrs-gif.fr

The NEURON simulation program now allows Python to be used, alone or in combination with NEURON's traditional Hoc interpreter. Adding Python to NEURON has the immediate benefit of making available a very extensive suite of analysis tools written for engineering and science. It also catalyzes NEURON software development by offering users a modern programming tool that is recognized for its flexibility and power to create and maintain complex programs. At the same time, nothing is lost because all existing models written in Hoc, including graphical user interface tools, continue to work without change and are also available within the Python context. An example of the benefits of Python availability is the use of the `xm1` module in implementing NEURON's Import3D and CellBuild tools to read MorphML and NeuroML model specifications.

Keywords: Python, simulation environment, computational neuroscience

INTRODUCTION

The NEURON simulation environment has become widely used in the field of computational neuroscience, with more than 700 papers reporting work employing it as of April, 2008. In large part this is because of its flexibility and the fact that it is continually being extended to meet the evolving research needs of its user community. Experience shows that most of these needs have a software solution that has already been implemented elsewhere in the domain of scientific computing. The problem is one of interfacing an existing package with NEURON's interpreter. Some cases demand intimate knowledge of NEURON's internals and considerable effort; examples include network parallelization with MPI, and adoption of Sundials for adaptive integration. There are many more cases in which existing packages could potentially be employed by NEURON users. Few people, however, have the specialized expertise required to manually interface an existing software package and the creation of such interfaces is tedious. Instead of laborious piecemeal adoption of individual packages that requires intervention by a handful of experts, a better approach is to offer Python as an alternative interpreter so that a huge number of resources becomes available at the cost of only minimal interface code that most users can write for themselves.

Since 1984, the NEURON simulation environment has used the Hoc interpreter (Kernighan and Pike, 1984) for setup and control of neural simulations. Hoc has a syntax for expressions and control flow vaguely similar to the C language. Hoc is not exactly an interpreted language since, analogous to Pascal, Java, or Python, Hoc statements are first dynamically compiled to an internal stack machine representation using a yacc parser and then the stack machine statements are executed. A fundamental extension to Hoc syntax was made in the late 80's 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 90's, Hoc syntax was again extended to provide some limited support for classes and objects, that is, data encapsulation and polymorphism, but not inheritance.

Though Hoc has served well, continuing development and maintenance of a general programming language steals significant time and effort from neurobiology domain-specific improvements. Furthermore, Hoc has turned out to be an orphan language limited to NEURON users. What is desirable is a modern programming language such as Python, which provides expressive syntax, powerful debugging capabilities, and support for modularity, facilitating the construction and maintenance of complex programs. Python has proved its utility by giving rise to a large and diverse community of software developers who are making reusable tools that are easy to plug-in to the user's code, the so-called "batteries included" (Dubois, 2007). In the domain of scientific computing, some examples include Numpy (Oliphant, 2007) and Scipy (Jones et al., 2001) for core scientific functionality, Matplotlib (Hunter, 2007) for 2-D plotting, and IPython (Perez and Granger, 2007) for a convenient interactive environment.

There are three distinct ways to use NEURON with Python. One is to run the NEURON program with Python as the interpreter accepting interactive commands in the terminal window. Another is to run NEURON with Hoc as the interactive interpreter and access Python functionality through Hoc objects and function calls. These first two cases we will refer to as embedded Python. The third way is to dynamically import NEURON in a running Python or IPython instance, which we will refer to as using NEURON as an extension module for Python.

In the sections to follow, we describe the steps required to use NEURON with Python, from a user's point of view, and the techniques employed to enable NEURON and Python to work together, from a developer's point of view. We begin in Section "Getting

Started Using NEURON with Python” by describing how to install and run NEURON with Python. We then demonstrate how modeling is carried out using Python by comparing it side-by-side with Hoc syntax in Section “Writing NEURON Models in Python”. In Section “Using Python Code from Hoc”, we describe how Python can be accessed from the Hoc interpreter. In Section “Technical Aspects”, we discuss some technical aspects of the implementation of the Python-NEURON interaction. Finally, in Section “Importing MorphML Files — A Practical Example” we give a detailed, practical example, from the current NEURON distribution, of combining Python and Hoc.

The code listings in **Figures 1–3** are available for public download from the ModelDB model repository of the Senselab database, <http://senselab.med.yale.edu> (accession number 116491).

GETTING STARTED USING NEURON WITH PYTHON

INSTALLATION

NEURON works with Python on Windows, Mac OS X, Linux, and many other platforms such as the IBM Blue Gene/L/P and Cray XT3 supercomputers. Detailed installation information can be found at <http://www.neuron.yale.edu> by following the “Download and Install” link.

Binary installers are available for Windows, OS X and RPM-based Linux systems. The Windows installer contains a large portion of Cygwin Python 2.5. On OS X and Linux, the latest version of Python 2.3–2.5 previously or subsequently installed is dynamically loaded when NEURON is launched. The binary installers provide Python embedded in NEURON, but do not support using NEURON as an extension module for Python or IPython.

If you would like to use NEURON as an extension module for Python or IPython, if no installer for your platform exists, or if you need to customize the installation (e.g. enable parallel/MPI support, or change the location of binaries), you should instead get the source code for the standard distribution, also available from the above “Download and Install” link, and compile it for your machine. Further instructions for this are given in the Appendix.

BASIC USE

NEURON may be started without the graphical user interface (GUI) using `nrniv` or with the GUI using `nrngui`. To use Python as the interpreter, rather than Hoc, use the `-python` option:

```
$ nrniv -python
NEURON -- VERSION 7.0 (228: fbb244f333a9)
2008-11-25
Duke, Yale, and the BlueBrain Project --
Copyright 1984-2008
See http://www.neuron.yale.edu/credits.html

>>> from neuron import h
```

If there are any NEURON NMODL extension mechanisms (Hines and Carnevale, 2000) in the working directory, and they have been compiled with `nrnivmodl`, they will be loaded automatically.

Alternatively, you may wish to use NEURON as an extension to the normal Python interpreter, or to IPython (Prez and

Granger, 2007), a more interactive variant. To do so, you must build NEURON from source and install the NEURON shared library for Python, as described in the Appendix. In Python (or IPython) then, NEURON is started (and any NMODL mechanisms loaded) when you import `neuron`:

```
$ ipython
[...]

In [1]: from neuron import h
NEURON -- VERSION 7.0 (228: fbb244f333a9)
2008-11-25
Duke, Yale, and the BlueBrain Project --
Copyright 1984-2008
See http://www.neuron.yale.edu/credits.html
```

and the NEURON GUI is started by importing the `neuron.gui` module:

```
In [2]: from neuron import gui
```

The `h` object that we import from the `neuron` module is the principal interface to NEURON’s functionality. `h` is a `HocObject` instance, and has two main functions. First, it gives access to the top-level of the Hoc interpreter, e.g.:

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

Second, it makes any of the classes defined in Hoc available to Python:

```
>>> stim = h.IClamp(0.5, sec=h.soma)
```

Note that the `soma` section created through the Hoc interpreter appears in Python as a `Section` object. We can also create `Sections` directly in Python, e.g.

```
>>> dend = h.Section()
```

These two section objects are entirely equivalent, the only difference being that the name “`dend`” is not accessible within the Hoc interpreter. In addition to the `HocObject` class (and through it, any class defined in Hoc) and the `Section` class, the Python `neuron` module also provides the `Segment`, `Mechanism` and `RangeVariable` classes. More in-depth examples of using NEURON from Python are given in Section “Writing NEURON Models in Python”, while using Python code from Hoc is introduced in Section “Using Python Code from Hoc”.

STARTING PARALLEL NEURON

Assuming NEURON was built with parallel support as discussed in the Appendix, suitably parallelized Hoc scripts are started using the MPI job execution command, typically `mpiexec` (Hines and Carnevale, 2008) or the equivalent for your MPI implementation. When Python is used rather than Hoc, the same parallelism features are supported, with only slight changes in the execution model. Both embedded Python (`nrniv -python`) and NEURON as an extension module to Python are supported. MPI job execution for embedded Python is the same as standard NEURON/Hoc, except

```

from itertools import chain
from neuron import h
Section = h.Section

# ----- Model specification -----

# topology
soma = Section()           # create soma, apical, basilar, axon
apical = Section()
basilar = Section()
axon = Section()

apical.connect(soma, 1, 0) # connect apical(0), soma(1)
basilar.connect(soma, 0, 0) # connect basilar(0), soma(0)
axon.connect(soma, 0, 0)    # connect axon(0), soma(0)

# geometry

soma.L = 30                # soma {
soma.nseg = 1              #     L = 30
soma.diam = 30             #     nseg = 1
                           #     diam = 30
                           # }
                           # apical {
apical.L = 600             #     L = 600
apical.nseg = 23           #     nseg = 23
apical.diam = 1            #     diam = 1
                           # }
                           # basilar {
basilar.L = 200            #     L = 200
basilar.nseg = 5           #     nseg = 5
basilar.diam = 2           #     diam = 2
                           # }
                           # axon {
axon.L = 1000              #     L = 1000
axon.nseg = 37             #     nseg = 37
axon.diam = 1              #     diam = 1
                           # }

# biophysics
for sec in h.allsec():     # forall {
    sec.Ra = 100           #     Ra = 100
    sec.cm = 1             #     cm = 1
                           # }

soma.insert('hh')          # soma {
                           #     insert hh
                           # }
apical.insert('pas')       # apical {
                           #     insert pas
                           #     g_pas = 0.0002
                           #     e_pas = -65
                           # }
basilar.insert('pas')      # basilar {
                           #     insert pas
                           #     g_pas = 0.0002
                           #     e_pas = -65
                           # }
for seg in chain(apical, basilar): #
    seg.pas.g = 0.0002      #
    seg.pas.e = -65        #
                           #
                           # axon {
axon.insert('hh')          #     insert hh
                           # }

```

FIGURE 1 | Code listing for a simple model neuron: building the neuron. The Python code is on the left and the equivalent Hoc code on the right.

```

# ----- Instrumentation -----

# synaptic input
syn = h.AlphaSynapse(0.5, sec=soma)
syn.onset = 0.5
syn.gmax = 0.05
syn.e = 0

#
# objref syn
soma syn = new AlphaSynapse(0.5)
syn.onset = 0.5
syn.gmax = 0.05
syn.e = 0

#
# objref g
g = h.Graph()
g.size(0, 5, -80, 40)
g.addvar('v(0.5)', sec=soma)
#
# g = new Graph()
# g.size(0, 5, -80, 40)
# g.addvar("soma.v(0.5)")

# ----- Simulation control -----

h.dt = 0.025
tstop = 5
v_init = -65

def initialize():
    h.finitialize(v_init)
    h.fcurrent()

#
# dt = 0.025
# tstop = 5
# v_init = -65
#
# proc initialize() {
#     finitialize(v_init)
#     fcurrent()
# }

def integrate():
    g.begin()
    while h.t < tstop:
        h.fadvance()
        g.plot(h.t)

#
# proc integrate() {
#     g.begin()
#     while (t < tstop) {
#         fadvance()
#         g.plot(t)
#     }
#     g.flush()
# }

def go():
    initialize()
    integrate()

#
# proc go() {
#     initialize()
#     integrate()
# }

go()
#
# go()

```

FIGURE 2 | Code listing for a simple model neuron (continued from Figure 1): instrumenting and running the model. The Python code is on the left and the equivalent Hoc code on the right.

that an extra -python command line option must be passed to nrniv:

```
$ mpiexec -np 4 nrniv -python -mpi nrn-7.0/\
src/nrnpython/examples/test1.py
```

```
numprocs=4
NEURON -- VERSION 7.0 (228: fbb244f333a9)
2008-11-25
```

```
Duke, Yale, and the BlueBrain Project --
Copyright 1984-2008
See http://www.neuron.yale.edu/credits.html
NEURON thinks I am 0 of 4
NEURON thinks I am 2 of 4
NEURON thinks I am 3 of 4
NEURON thinks I am 1 of 4
```

For users who prefer to use NEURON as an extension module to Python or IPython, execution is as follows:

```
$ mpiexec -np 4 python nrn-7.0/src/nrnpython/\
examples/test0.py
```

```
MPI_Initialized==true, enabling MPI
functionality.
```

```
numprocs=4
NEURON -- VERSION 7.0 (228: fbb244f333a9)
2008-11-25
Duke, Yale, and the BlueBrain Project --
Copyright 1984-2008
See http://www.neuron.yale.edu/credits.html
```

```
mpi4py thinks I am 2 of 4, NEURON thinks I am
2 of 4
mpi4py thinks I am 1 of 4, NEURON thinks I am
1 of 4
mpi4py thinks I am 3 of 4, NEURON thinks I am
3 of 4
mpi4py thinks I am 0 of 4, NEURON thinks I am
0 of 4
```

However, there is one important caveat: The NEURON extension module does not initialize MPI itself, but rather delegates this job to Python. To initialize MPI in Python, one must import a

```

from neuron import h

# create pre- and post-synaptic sections
pre = h.Section()
post = h.Section()

for sec in pre, post:
    sec.insert('hh')

# inject current in the pre-synaptic section
stim = h.IClamp(0.5, sec=pre)
stim.amp = 10.0
stim.delay = 5.0
stim.dur = 5.0

# create a synapse in the pre-synaptic section
syn = h.ExpSyn(0.5, sec=post)

# connect the pre-synaptic section to the
# synapse object
nc = h.NetCon(pre(0.5)._ref_v, syn)
nc.weight[0] = 2.0

vec = {}
for var in 'v_pre', 'v_post', 'i_syn', 't':
    vec[var] = h.Vector()

# record the membrane potentials and
# synaptic currents
vec['v_pre'].record(pre(0.5)._ref_v)
vec['v_post'].record(post(0.5)._ref_v)
vec['i_syn'].record(syn._ref_i)
vec['t'].record(h._ref_t)

# run the simulation
h.load_file("stdrun.hoc")
h.init()
h.tstop = 20.0
h.run()

# plot the results
import pylab
pylab.subplot(2,1,1)
pylab.plot(vec['t'], vec['v_pre'],
           vec['t'], vec['v_post'])
pylab.subplot(2,1,2)
pylab.plot(vec['t'], vec['i_syn'])

```

FIGURE 3 | Code listing demonstrating the use of ref and plotting.

Python MPI module, such as “MPI for Python” (mpi4py) (Dalcín et al., 2008), prior to importing neuron:

```

from mpi4py import MPI
from neuron import h

pc = h.ParallelContext()

s = "mpi4py thinks I am %d of %d,\
    NEURON thinks I am %d of %d\n"

cw = MPI.COMM_WORLD
print s % (cw.rank, cw.size, \
           pc.id(), pc.nhost())

pc.done()

```

The module mpi4py is available from the Python Package Index (<http://pypi.python.org>).

ONLINE HELP

For new users of NEURON with Python, a convenient starting place for help is Python online help, provided through the global function `help`, which takes one argument, the object on which you would like help:

```

>>> import neuron
>>> help(neuron)
Help on package neuron:

```

NAME

neuron

FILE

/usr/lib/python2.5/site-packages/neuron/
__init__.py

DESCRIPTION

neuron
=====

For empirically-based simulations of neurons and networks of neurons in Python.

This is the top-level module of the official python interface to the NEURON simulation environment (<http://www.neuron.yale.edu/neuron/>).

For a list of available names, try `dir(neuron)`.

[...]

For commonly used Hoc classes, such as `Vector`, `APCount`, `NetCon`, etc., helpful reminders of constructor arguments, attributes and units with Python syntax examples are available at the Python prompt:

```

>>> from neuron import h
>>> help(h.APCount)
NEURON+Python Online Help System
=====

```

```

class APCount
pointprocess
apc = APCount(segment)
apc.thresh --- mV
apc.n --
apc.time --- ms
apc.record(vector)

```

Description:

Counts the number of times the voltage at its location crosses a threshold voltage in the positive direction. `n` contains the count and `time` contains the time of last crossing.

[...]

In IPython, the `?` symbol is a quick shorthand roughly equivalent to online help:

```
In [3]: ? h.APCount

Type:          HocObject
Base Class:    <type 'hoc.HocObject'>
String Form:   <hoc.HocObject object at 0
               xb79022f0>
Namespace:    Interactive
Length:       0
Docstring:
    class APCount
        pointprocess
[...]

```

WRITING NEURON MODELS IN PYTHON

To show how a model neuron is implemented using Python, we repeat the example described in Chapter 6 of the NEURON Book (Carnevale and Hines, 2006), but using Python rather than Hoc. The code listing is given in **Figures 1 and 2**, and has Python code on the left and the equivalent Hoc code on the right.

There are only a few syntax and conceptual differences between the Python and Hoc versions, and we expect that Hoc users will have little difficulty transitioning to Python, should they wish to do so (Hoc will continue to be supported, of course). We now comment on the most significant differences.

First are the `import` statements, absent from the Hoc listing, although Hoc does have the `xopen()` function that has similar functionality. Since NEURON is now only one of potentially many modules living within the Python interpreter, it must live in its own namespace, so that the names of NEURON-specific classes and variables do not interfere with those from other modules. Of particular importance is the object `h`, which is the top-level Hoc interpreter, and gives access to Hoc classes, functions and variables.

While sections are created using the `create` keyword in Hoc, in Python we instantiate a `Section` object. Hence the important distinction in Hoc between sections and objects is removed: Everything in Python is an object. Similarly, the `connect` keyword in Hoc is replaced by a method call of the child section object in Python.

In NEURON, each cable section is made up one or more segments, and the diameter is a property of each segment. Hoc's shorthand, allowing the `diam` attribute to be set on all segments by setting it on the section is also available in Python. Inhomogeneous values for range variables such as `diam` can also be set on the specific `Segment` object, returned by calling the `Section` object as a function.

The `forall` keyword in Hoc, which iterates over all sections, is replaced by the `allsec()` method of the top-level Hoc interpreter object `h`. Here again we see, in setting the membrane capacitance `cm`, the Hoc and Python shorthands to set the value for all segments at once, without having to explicitly iterate over all `Segments`.

In instrumenting the model, we see that Python and Hoc objects have very similar behaviours. In general, all Hoc classes (`Vector`, `List`, `NetCon`, etc) are accessible within Python via the `h` object. Hoc object references must be declared using the `objref` keyword, and objects created using `new`, but once created, attribute access and method calls have near-identical syntax in Python and Hoc.

There are three major exceptions to this rule. First, many functions and methods act in the context of the 'currently-accessed section'. To support this in Python, these functions take a keyword argument `sec`. Second, certain method calls take Hoc expressions as arguments, so, for example, in adding the membrane potential of the soma section to the list of variables to plot, in Hoc we use `g.addvar("soma.v(0.5)")`, but in the Python version the variable `soma` does not exist on the Hoc side, and so we have to pass the `soma Section` object as the `sec` keyword argument so that the Hoc expression is evaluated in the context of that section. Third, a number of functions/methods take Hoc variable references (indicated by preceding the variable name with the `&` character) as arguments, the most important being `Vector.record(&var)` and `NetCon(&var, target)`. The equivalent syntax in Python is to precede the variable name with `_ref_`, e.g.: `Vector.record(_ref_var)`. For example, given 'pre' and 'post' `Section` objects and a dictionary of Hoc `Vector` objects addressed by a mnemonic string, recording the voltage at the centres of those sections is activated by the statements:

```
# record the membrane potentials and
# synaptic currents
vec['v_pre'].record(pre(0.5)._ref_v)
vec['v_post'].record(post(0.5)._ref_v)
vec['i_syn'].record(syn._ref_i)
vec['t'].record(h._ref_t)

```

Figure 3 shows the complete listing with the above fragment in context and also illustrates the ease with which NEURON code can be mixed with third-party code such as the powerful Pylab/Matplotlib plotting package (<http://matplotlib.sourceforge.net/>): NEURON `Vector` objects work just as well as Python lists or arrays as arguments to the `plot()` function.

USING USER-DEFINED MECHANISMS

One of NEURON's most powerful features is the ability to write new mechanisms using the NMODL language, and then compile these mechanisms into the executable or into dynamic libraries (DLLs). The standard behaviour of NEURON is to load any mechanisms that have been compiled in the working directory. It is also possible to load DLLs from elsewhere in the filesystem using the Hoc function `nrn_load_dll()`. This has the disadvantage that the full path to the shared library file must be provided, which can be hard to determine, since the file is within a hidden folder which itself is within a folder with a platform-specific name. To simplify this, the `neuron` Python module adds a function `load_mechanisms()`, which takes as an argument the path to the directory containing the NMODL source files, and searches for shared library files below this directory. Furthermore, in analogy to the `PYTHONPATH` environment variable which contains a list of paths to search for importable Python modules, if you have defined a `NRN_NMODL_PATH` environment variable, NEURON will search these paths for shared libraries and load them at import time.

USING USER-DEFINED CLASSES

One of the principal advantages of writing NEURON programs in Python rather than Hoc, especially for large, complex programs, is that Python is a fully object-oriented language, supporting

encapsulation, polymorphism and inheritance, whereas Hoc supports only encapsulation and a limited form of polymorphism.

Just as with built-in Hoc classes, access to attributes and methods of user-defined Hoc classes (using the `begintemplate/endtemplate` keywords) uses the same syntax in Python as in Hoc. For example, if we have the following user-defined Hoc class in the file `string.hoc`:

```
begintemplate String
  public s
  strdef s
  proc init() {
    s = $s1
  }
endtemplate String
```

then we can use it as follows:

```
>>> from neuron import h
>>> h.xopen("string.hoc")
>>> my_string = h.String("Hello")
>>> my_string.s
'Hello'
```

It is also possible to subclass both built-in and user-defined Hoc classes in Python, although with the restriction that multiple inheritance from Hoc-derived classes is not possible. Subclassing requires the use of the `hclass` class factory:

```
>>> from neuron import h, hclass
>>> class MyNetStim(hclass(h.NetStim)):
...     """NetStim that allows setting
...     parameters on creation."""
...
...     def __init__(self, start=50, noise=0,
...                   interval=10, number=10):
...         self.start = start
...         self.interval = interval
...         self.noise = noise
...         self.number = number
...
>>> stim = MyNetStim(start=0, noise=1)
>>> stim.noise
1.0
>>> class MyString(hclass(h.String)):
...     def repeat(self, n):
...         return self.s*n
...
>>> my_string = MyString("Hello")
>>> my_string.repeat(3)
'HelloHelloHello'
```

NUMERICAL DATA TRANSFER BETWEEN HOC AND PYTHON

The Hoc Vector object provides NEURON with a convenient and efficient container for storing and manipulating collections of numerical values, such as membrane potential traces or spike-times.

In Python, Hoc Vector objects expose iterator and indexing methods, such that they can be used in most cases where Numpy

(Oliphant, 2007), Scipy (Jones et al., 2001), and Matplotlib (Hunter, 2007), the most important scientific modules, accept lists.

To benefit from the elegant and expressive notation of Numpy for N-dimensional array manipulation, and from results computed using the large and growing repertoire of scientific packages available for Python, which largely return Numpy arrays, several optimized methods are available for the conversion of Hoc Vectors to and from Numpy arrays.

Transferring one-dimensional Numpy arrays and non-nested lists with float or integer items to Hoc Vectors is straightforward, as the Hoc Vector constructor accepts an array or list as an argument:

```
>>> v1 = h.Vector(a)
>>> v2 = h.Vector(1)
```

Transferring a Hoc Vector to an array or list is equally straightforward:

```
>>> a = array(v1)
>>> print a
[ 3.  2.  3.  2.]
>>> l = list(v2)
>>> print l
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
```

If you would like to transfer between an existing Numpy array and a Hoc Vector, there are the Hoc Vector “in-place” member functions `to_python` and `from_python`:

```
>>> v3 = h.Vector(len(a))
>>> v3.from_python(a)
>>> print list(v3)
[3.0, 2.0, 3.0, 2.0]
>>> b = zeros_like(a)
>>> v3.to_python(b)
>>> print b
[ 3.  2.  3.  2.]
```

USING PYTHON CODE FROM HOC

For interacting with Python, Hoc provides the `nrnpython()` function and the `PythonObject` class. `nrnpython()` takes as its one argument a string that can be any Python statement, e.g.:

```
oc> nrnpython("a = 3.14159")
oc> nrnpython("print a")
3.14159
```

`PythonObject` has two main uses. Creating an instance using `new` returns an object that encapsulates the top-level Python interpreter, e.g.

```
oc> objref py
oc> py = new PythonObject()
oc> py.b = "hello"
oc> nrnpython("print b")
hello
```

Strings and float/double values move back and forth between Python and Hoc (although Python integers become double values in

Hoc and remain doubles if they are passed back to Python). All other Python objects become instances of the `PythonObject` class:

```
oc> objref dict
oc> nrnpython("d = {'a':1, 'b':2, 'c':3}")
oc> dict = py.d
oc> print dict
PythonObject [12]
oc> print dict.__getitem__("c")
3.0
```

For objects (such as lists and tuples) that take integer indices or are callable as functions, there is a special method named `'_'` (underscore):

```
oc> objref lst
oc> nrnpython("c = [7, 8.0, 'nine']")
oc> lst = py.c
oc> for i = 0, lst.__len__() -1 { print lst._[i] }
7.0
8.0
nine
```

The only other trap for the unwary is that both single and double quotes are valid for string definitions in Python, but only double quotes are accepted by Hoc!

A detailed example of using Python from Hoc, and of the value of being able to access its large standard library, is given in Section “Importing MorphML Files — A Practical Example” for the case of importing 3D morphology from a MorphML file.

TECHNICAL ASPECTS

Tools for building Python extensions, such as `BOOST.Python` (Abrahams and Grosse-Kunstleve, 2003) or `SWIG` (Beazley, 1996) might have been useful in more expert hands. However, the ability of users to declare variables, objects, and classes in Hoc, the fact that many existing C++ classes and class methods were not generally meant to be directly visible to the user except through the intermediation of Hoc syntax, and the fact that the Hoc connection to the internal NEURON code was already reasonably uniform, of reasonable size, and understood by us in depth, suggested to us that a Python interface written using the Python C-API (<http://docs.python.org/c-api/>) that reused as much as possible the existing Hoc connection to internal data and functions would give us the general control we needed, and allow us to accomplish the project in reasonable time. It should be emphasized that this design decision to reuse a few of the C functions that manipulate the Hoc runtime stack neither hinders nor assists any future work on development of APIs for major NEURON components, such as the numerical solvers, which may be useful to other simulators. However, our interface implementation does provide a compact example of how an application can communicate with NEURON within a shared address space and therefore makes the the process of dynamically linking NEURON into a user application much simpler.

Since double precision variables, arrays, constant strings, functions, and objects have very similar syntax and semantics in Hoc and Python, a single `PyTypeObject` structure called `HocObjectType` associated with a `PyHocObject` structure for

a Python object instance containing Hoc Symbol and Object fields was sufficient to allow Python access to all these Hoc data-types. When a name is given to an attribute method of the `HocObjectType` (the reflexive self `PyHocObject` is also an argument to the method), the name is looked up in Hoc’s symbol table for the `PyHocObject` Hoc Object field, and the symbol along with the Hoc object calls the same function that the Hoc interpreter would call to resolve the attribute at runtime. The attribute, which is typically a number, string, or `HocObject`, is then wrapped in a Python object of the proper type and returned. Function calls from Python into Hoc consist of pushing the function arguments onto the Hoc runtime stack and, again, calling the same function the Hoc interpreter would call at runtime. Thus, Python statements involving `PyHocObject` objects end up generating and executing the same Hoc stack machine code at runtime that would be accomplished by the corresponding Hoc statement. It should be noted that a great deal of interpreter efficiency can be gained in loop body statements by factoring out as much as possible the precursor objects. For example:

```
from neuron import h
vec = h.Vector (1000000)
a = 0
for i in xrange (1000000):
    a += vec.x[i]
```

can be optimized by avoiding the repeated search for the attribute `x`:

```
vx = vec.x
for i in xrange (1000000):
    a += vx[i]
```

The former takes 1.3 s on a 3 GHz machine, while the latter takes 1.0 s.

A critical requirement was to have as natural a correspondence as possible in Python for the special Hoc syntax for Sections, position along a Section, membrane mechanisms, and Range Variables. This was achieved through the C++ definition of corresponding types in Python to create instances for: `NPYSecObj`, `NPYSegObj`, `NPYMechObj`, and `NPYRangeVar`. For example, the `NPYSegObj` segment (compartment) object points to the `NPYSecObj` of which it is a part, specifies its location, `x`, and also contains a field to help in iterating over the mechanisms that exist at that location. An `NPYRangeVar` has, in practice, required only a pointer to the compartment (`NPYSegObj`) where it exists and a pointer to its Hoc Symbol. A Section represents a continuous cable and evaluation of or assignment to a variable associated with a particular location always involves specifying both which Section and the relative arc length location ($0 \leq x \leq 1$) along the Section. Internally, NEURON employs a Section stack to determine the working Section and Hoc syntax provided three ways to specify the top of the Section stack. The Hoc `Section.variable(x)` syntax has a direct correspondence to the Python `Section(x).variable` syntax and the latter perhaps has more clarity. The Hoc `Section { Hoc statements }` syntax is unique to NEURON and for the Python side we were reduced to explicit management of the Section stack with `Section.push()` with an explicit `h.pop_section()` as the final statement. This gets tedious for single function calls and so in

Python we allow the keyword argument, `sec=Section`, to push and pop the Section during the scope of the Hoc function call. The Hoc `access Section` statement does not require a Python counterpart. However, the Python statement, `sec = h.cas()`, returns the top of the Section stack.

There were several cases of syntax mismatch which could only be overcome by the addition of new idioms. Hoc syntax does not allow an object to be treated as a function, so in Hoc we use `po._(...)`. Python does not allow call by reference arguments. Therefore, when a Hoc function called from Python requires a reference argument, the variable name must be prefixed by `'_ref_'`. Of course, such variables can only be Hoc variables but that is not a difficulty in practice since either the need is to pass a Hoc RangeVariable or the Python program can construct a Hoc variable for use in these cases. Since all numbers in Hoc are double precision, type errors are raised when Python expects an integer. For the case of array arguments, the Hoc-to-Python interface converts the doubles to integers automatically. Unfortunately, one cannot in general call the `__getitem__(int)` method explicitly but must use the `[expr]` Hoc syntax. If this becomes a problem in practice, it will be necessary to supply a set of cast functions that can be explicitly invoked by the user.

We have encountered only one problem with freeing object memory that has proved resistant to a solution. In some cases there is an ambiguity in regard to whether the Hoc or the Python side owns a reference to an object. When this situation occurs, a reference to the object is kept in a list for a deferred call to `Py_DECREF` when it is guaranteed that it is safe to do so.

Assignment of a constant value to a range variable in a Section is far more common than assignment of different values within the segments of a Section and Hoc provides a simple syntax for that case which avoids writing an explicit loop. The latest extension of the NEURON Python interface mimics that behavior in Python by interpreting `Section.RangeVariableName` in that fashion instead of raising an `"AttributeError"`. We are also considering extending the implicit iteration idea to `SectionLists` and `Cells` to allow not only assignment of constants but also application of inhomogeneous functions.

A list of the principal differences in syntax between Hoc and Python is given in **Table 1**.

IMPORTING MORPHML FILES — A PRACTICAL EXAMPLE

Our first serious use of the NEURON Python interface was to extend the Import3D GUI tool to read MorphML specification files. Import3D is structured around a graphical view of a list of `Import3d_Section` objects defined in Hoc. Among many method and field attributes, the principle data field of the `Import3d_Section` object is the raw `x, y, z, diam` information along an unbranched cable and a list index indicating the parent `Import3d_Section`. The list of `Import3d_Section` objects is constructed by various file reader objects that understand a specific file format such as Eutectic, SWC, or NeuroLucida versions 1 or 3. Since MorphML is an XML format, it was opportune to employ the XML reader module in the standard Python distribution.

The problem of parsing and analyzing the MorphML format is similar in difficulty to that for NeuroLucida V3 files. We divided

the problem into Hoc and Python code portions. In contrast to a file size of 1180 lines for the NeuroLucida V3 file reader, the `read_morphml.hoc` file size is 78 lines and the Python portion of the problem is carried out by `rdxml.py` with a file size of 370 lines. Since these files are located in the NEURON package default search path — `.../nrn/lib/hoc` for the `read_morphml.hoc` file and `.../nrn/lib/python` for the `rdxml.py` file — the MorphML reader extension works wherever the NEURON Python interface is installed.

The `read_morphml.hoc` file defines an `Import3d_MorphML` Hoc template (class) which interacts with `Import3d_GUI` in exactly the same manner as the other format readers.

When an `Import3d_MorphML` instance is created, the Python helper module we wrote to parse the input file is imported with `nrnpymodule("import rdxml")` and `p = new PythonObject()` is defined in order to allow access to Python functions.

The `proc input() {...}` procedure defines a sections list and populates it with `Import3dSection` objects indirectly via `p.rdxml.rdxml($s1, this)` which passes the filename selected earlier by the user along with a reference to the `Import3dMorphML` instance to allow callback from the Python code.

The

```
def rdxml(fname, ho) :
    xml.sax.parse(fname, MyContentHandler(ho))
```

module function calls the xml parser with the filename and a new instance of

```
class MyContentHandler(xml.sax.ContentHandler):
    def __init__(self, ho):
        self.i3d = ho
        ...
```

The reference to the `Import3d_MorphML` instance is stored by the initializer for later use at the end of parsing. During file reading there is no interaction between Hoc and Python, so let it suffice that the xml parsing style is, at the beginning and end of every xml element, to call the `MyContentHandler` methods

```
def startElement(self, name, attrs):
    if self.elements.has_key(name):
        if debug: print "startElement:", name
        self.elements[name](self, attrs)
    else :
        if debug:
            print "startElement unknown", name

def endElement(self, name):
    if self.elements.has_key('end'+name):
        self.elements['end' +name](self)
```

where the `elements` literal map associates all possible element names with a `MyContentHandler` method. E.g.

```
elements = {
    'neuroml':nothing,
    'morphml':nothing,
    ...
    'segments':segments,
    'endsegments':endsegments,
```


Table 1 | The principal differences in syntax between Hoc and Python.

Python	Hoc	Notes
obj()	obj._()	
obj[int]	obj._[int]	
obj[double]	obj._getitem__(double)	or <code>__setitem__</code>
obj['string']	obj._getitem__("string")	or <code>__setitem__</code>
f(_ref_var)	f(&var)	when storing a persistent pointer
f(h.ref(strvar))	f(strvar)	when f changes the string
f(h.ref(obj))	f(obj)	when f changes the reference
f(h.ref(var))	f(&var)	when f changes var (via <code>\$&1</code>)
sec = Section()	create sec	
sec.push() stmt h.pop_section()	sec { stmt }	
f(..., sec = section)	section { f(...) }	
child.connect(parent, px, cx)	connect child(cx), parent(px)	
sec.insert('mechname')	sec { insert mechname }	
sec(x).rangevar	sec.rangevar(x)	
for sec in h.allsec():	forall { }	includes <code>sec.push()</code> and <code>h.pop_section()</code> of currently accessed section.
for sec in h.seclist:	forsec seclist { }	
for seg in sec:	for (x, 0)	the value of x is <code>seg.x</code>
for seg in sec.allseg():	for (x)	
seg.hh.gnabar or seg.gnabar_hh	gnabar_hh(x)	
pp = PointProcess(x, sec=section)	sec { pp = new PointProcess(x) }	
for mech in seg:	No direct equivalent. Use <code>MechanismType</code>	
iteration	for iterator	Python supplies several styles of iteration and Hoc supplies an iterator idiom. Conversion from one to the other is done via explicit programming but Python cannot use a Hoc iterator directly. Nor can Hoc use generators except by calling the underlying <code>__next__()</code> method.

```
'segment':segment,
'proximal':proximal,
...
}
```

The methods construct Python lists of `Point`, `Cable`, etc, as well as maps associating identifiers with list indices. At the end of parsing, the `MyContentHandler` method

```
def endDocument(self):
    self.i3d.parsed(self)
```

is called by the xml parser.

At this point we find ourselves back in the Hoc world with an argument that references the `MyContentHandler`. Through that we can obtain the information saved by the `MyContentHandler` in various maps and lists and copy it into new `Import3d_Section` instances.

```
proc parsed() {...
    cables = $o1.cables_
    points = $o1.points_
    cableid2index = $o1.cableid2index_
    for i=0, cables.__len__() - 1 {
```

```
cab = cables._[i]
sec = new Import3d_Section(cab.first_,\
    cab.pcnt_)
sections.append(sec)
if (cab.parent_cable_id_ >= 0) {
    ip = cableid2index_[cab.parent_cable_id_]
    sec.parentsec = sections.object(ip)
    sec.parentx = cab.px_
}
...
```

Note the `'_'` idiom for accessing a Python list element since, in Hoc, `cables[i]` is syntax implying an object reference array created with `objref cables[n]`. Also, `cableid2index` is a Python map which associates the cable identifier read from the xml input file, with the proper element in the Python `cables` list.

DISCUSSION

Python makes available within NEURON a very extensive suite of analysis tools written for the general science and engineering communities. All existing models written in Hoc, including GUI tools, continue to work without change. All NEURON objects are accessible to Python via an instance of the `HocObject`. Within the Hoc

interpreter, all Python objects are accessible via the `PyObject`. Binary installation remains straightforward for the usage case of launching NEURON with Python embedded: The MS Windows installer contains a large subset of the 2.5 version of Python, and the Linux RPM and Mac OS X dmg installations will use the latest version of Python, if any, that is already present or subsequently installed. The usage case of launching Python, e.g. using IPython, and dynamically importing NEURON also works but presently requires the extra installation steps described in the Appendix. Numpy is not a prerequisite but, if present, copying of vectors between Numpy and NEURON is very efficient. The Python `xml` module is used in the present standard distribution to extend NEURON's Import3D and CellBuild tools to allow reading of MorphML (Crook et al., 2007) and NeuroML (Goddard et al., 2001) model specifications. The Hoc portion of the `xml` readers makes heavy use of Python maps and lists.

With the release of NEURON version 7.0, the Python interface has largely stabilized, and is ready for general use. We recommend that new users of NEURON and those already familiar with Python should use Python rather than Hoc to develop new models. Those with considerable expertise in Hoc but without Python knowledge are likely to be more productive by continuing to develop models with Hoc, but accessing Python's powerful data structures, large standard library and external numerical/plotting packages through `nrnpython()` and the `PyObject` class. 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.

Users are encouraged to submit bug reports and feature requests at the NEURON forum (<http://www.neuron.yale.edu/phpBB>) in the "NEURON+Python" sub-section, so that we can continue to improve the Python interface in response to users' experiences.

APPENDIX

Here we give detailed instructions for building and installing NEURON as a Python extension. Note that, as mentioned earlier, to use NEURON with Python embedded you can use one of the binary installers.

The following assumes a standard GNU build environment, and a bash shell. You will need both NEURON (`nrn-VERSION.tar.gz`) and InterViews (`iv-VERSION.tar.gz`) sources, available through the "Download and Install" link at <http://www.neuron.yale.edu>.

First, build and install InterViews:

```
$ N='pwd'
$ tar xzf iv-17.tar.gz
$ cd iv-17
```

REFERENCES

- | | | | |
|--|---|--|---|
| <p>Abrahams, D., and Grosse-Kunstleve, R. W. (2003). Building hybrid systems with Boost.Python. <i>C/C++ Users J.</i> 21. http://www.ddj.com/cpp/184401666.</p> <p>Beazley, D. M. (1996). SWIG: an easy to use tool for integrating scripting</p> | <p>languages with C and C++. In <i>TCLTK'96: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop</i>, 1996, (Monterey, CA, USENIX Association), pp. 129–139.</p> <p>Carnevale, N. T., and Hines, M. L. (2006). <i>The NEURON Book</i>. Cambridge, Cambridge University Press.</p> | <p>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. <i>Neuroinformatics</i> 5, 96–104.</p> <p>Dalcín, L., Paza, R., Stortia, M., and D'Elia, J. (2008). MPI for Python: performance improvements and</p> | <p>MPI-2 extensions. <i>J. Parallel Distrib. Comput.</i> 68, 655–662.</p> <p>Dubois, P. F. (2007). Python: batteries included. <i>IEEE Comput. Sci. Eng.</i> 9, 7–9.</p> <p>Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A.,</p> |
|--|---|--|---|

```
$ ./configure --prefix='pwd'
$ make
$ make install
```

Then build and install NEURON:

```
$ cd .
$ tar xzf nrn-7.0.tar.gz
$ cd nrn-7.0
$ ./configure --prefix='pwd' \
  --with-iv=$N/iv-17 --with-nrnpython
$ make
$ make install
```

Here, the “\” at the end of the fourth line, indicates it is continued on the fifth. If you want to run parallel NEURON (Hines et al., 2008; Migliore et al., 2006), add `--with-paranrn` to the `configure` options. This requires a version of MPI to be installed, for example MPICH2 (Gropp, 2002) or openMPI (Gabriel et al., 2004).

Now add the NEURON bin directory to your PATH:

```
$ export PATH=$N/nrn-7.0/i686/bin:$PATH
```

(Here `i686` will be different for different CPU architectures).

Now build and install the NEURON shared library for Python:

```
$ cd src/nrnpython
# python setup.py install
```

This command installs the neuron package to the Python site-packages directory, which usually requires root access. If you don't have root access, you can install it locally using `--prefix` to specify a location under your home directory:

```
$ python setup.py install \
  --prefix=$HOME/local
```

This will install the neuron package to `$HOME/local/lib/python/site-packages` under your home directory. You will then have to add this directory to the `PYTHONPATH` environment variable:

```
$ export PYTHONPATH=$PYTHONPATH:\
$HOME/local/lib/python/site-packages
```

ACKNOWLEDGEMENTS

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.

- Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall T. S. (2004). Open MPI: goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, D. Kranzlmüller, P. Kacsuk and J. Dongara, eds (Budapest, Springer), pp. 97–104.
- Goddard, N., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modeling in neuroscience. *Philos. Trans. R. Soc. B* 356, 1209–1228.
- Gropp, W. (2002). MPICH2: a new start for MPI implementations. In Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, D. Kranzlmüller, P. Kacsuk, J. Dongara and J. Volkert, eds (London, Springer-Verlag), p. 7.
- Hines, M., and Carnevale, N. (2008). Translating network models to parallel hardware in NEURON. *J. Neurosci. Methods* 169, 425–455.
- Hines, M. L., and Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Comput.* 12, 995–1007.
- Hines, M. L., Markram, H., and Schuermann, F. (2008). Fully implicit parallel simulation of single neurons. *J. Comput. Neurosci.* 25, 439–448.
- Hunter, J. D. (2007). Matplotlib: a 2D graphics environment. *IEEE Comput. Sci. Eng.* 9, 90–95.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: open source scientific tools for Python. URL <http://www.scipy.org/>.
- Kernighan, B., and Pike, R. (1984). The Unix Programming Environment. Englewood Cliffs, NJ, Prentice Hall.
- Migliore, M., Cannia, C., Lytton, W. W., Markram, H., Hines, and M. L. (2006). Parallel network simulations with NEURON. *J. Comput. Neurosci.* 21, 119–129.
- Oliphant, T. E. (2007). Python for scientific computing. *IEEE Comput. Sci. Eng.* 9, 10–20.
- Prez, F., and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *IEEE Comput. Sci. Eng.* 9, 21–29.
- 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: 24 September 2008; paper pending published: 21 October 2008; accepted: 05 January 2009; published online: 28 January 2009

Citation: Hines ML, Davison AP and Muller E (2009) NEURON and Python. *Front. Neuroinform.* (2009) 3:1. doi: 10.3389/neuro.11.001.2009

Copyright © 2009 Hines, Davison and Muller. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Python for large-scale electrophysiology

Martin Spacek^{1*}, Tim Blanche² and Nicholas Swindale¹

¹ Ophthalmology and Visual Sciences, University of British Columbia, Vancouver, BC, Canada

² Redwood Center for Theoretical Neuroscience, University of California, Berkeley, CA, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Michele Giugliano, Ecole Polytechnique
Fédérale de Lausanne, Switzerland
Gaute T. Einevoll, Norwegian University
of Life Sciences, Norway

*Correspondence:

Martin Spacek, Department of
Ophthalmology and Visual Sciences,
University of British Columbia, 2550
Willow Street, Vancouver, BC V5Z 3N9,
Canada.
e-mail: frontiers@mspacek.mm.st

Electrophysiology is increasingly moving towards highly parallel recording techniques which generate large data sets. We record extracellularly *in vivo* in cat and rat visual cortex with 54-channel silicon polytrodes, under time-locked visual stimulation, from localized neuronal populations within a cortical column. To help deal with the complexity of generating and analysing these data, we used the Python programming language to develop three software projects: one for temporally precise visual stimulus generation ("dimstim"); one for electrophysiological waveform visualization and spike sorting ("spyke"); and one for spike train and stimulus analysis ("neuropy"). All three are open source and available for download (<http://swindale.ecc.ubc.ca/code>). The requirements and solutions for these projects differed greatly, yet we found Python to be well suited for all three. Here we present our software as a showcase of the extensive capabilities of Python in neuroscience.

Keywords: Python, silicon polytrodes, primary visual cortex, in-vivo

INTRODUCTION

As systems neuroscience moves increasingly towards highly parallel physiological recording techniques, generation, management, and analysis of large complex data sets is becoming the norm. We are interested in the function of localized neuronal populations in visual cortex. The goal is to understand how neurons in visual cortex respond to visual stimuli, to the extent that the responses to arbitrary stimuli can be predicted. Accurate prediction will require an understanding of how these neurons interact with each other. Neurons in close proximity are more likely to show functionally interesting interactions, and insights into how such localized populations work may help guide understanding of other parts of cortex, or even the brain as a whole. To this end we need to record and analyse the simultaneous spiking behaviour of many neurons in response to a wide variety of visual stimuli.

We use 54-channel silicon polytrodes, in both rat and cat primary visual cortex, to extracellularly sample spiking activity constrained to roughly a cortical column (**Figure 1A**) (Blanche et al., 2005). Time-locked visual stimuli are presented to the animal while simultaneously recording from dozens of neurons (**Figure 1B**). Waveforms are recorded continuously at a rate of 2.7 MB/s for up to 90 min (~15 GB) at a time. A single animal experiment can last up to 3 days and generate hundreds of GB of data. Setting up our electrophysiology rig, with custom acquisition software written in Delphi (Blanche, 2005), was the first step. Although we had existing solutions in place for visual stimulation, waveform visualization and spike sorting, and spike train analysis, all three had limitations which were addressed by rewriting our software in Python.

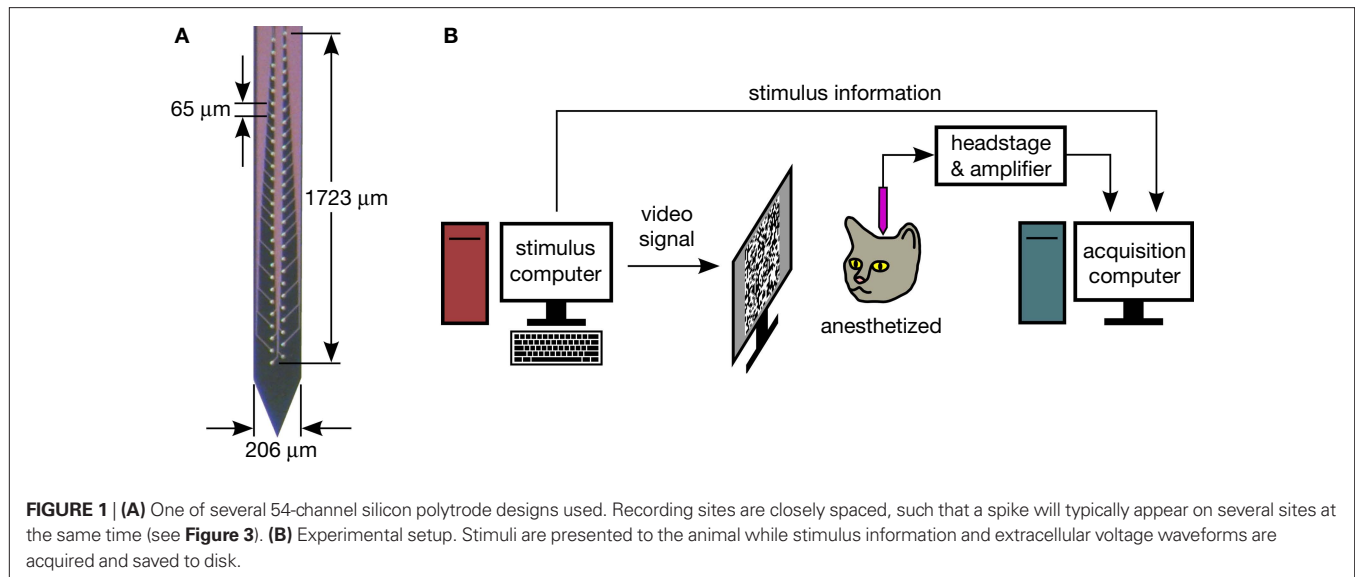
The first of those tackled was visual stimulation. After an extensive search for existing software, we discovered the "Vision Egg" (Straw, 2008), a Python library for generating stimuli. We chose the Vision Egg partly because of the language it was written in and written for: Python. We were thus introduced to Python via one of its many packages, and the experience was so positive that it

encouraged us to standardize on Python for the spike sorting and spike train analysis projects to follow. For one of us (M. Spacek), the switch to Python has made programming a much more enjoyable and productive experience, and has resulted in greatly improved programming skills.

The benefits of Python have been extolled at length elsewhere (Hetland, 2005; Langtangen, 2008; Lutz, 2006). Briefly, Python is a powerful, dynamically typed, interpreted language that "fits your brain", with syntax akin to "executable pseudocode". Python's clear, simple syntax is perhaps its biggest selling point. Some of its clarity stems from a philosophy to provide "one – and preferably only one – obvious way" to do a given task (Peters, 2004), making features easy to remember. Its clarity is also due to a strong adherence to object-oriented programming principles [Chapter 7 of Hetland (2005) is an excellent introduction]. In Python, nearly everything is an object, even numbers and functions. This means that everything has attributes and methods (methods are functions that are bound to and act on objects), and can thus be treated in a similar way. An object is an instance of a class. A class can inherit attributes and methods from other classes hierarchically, allowing for substantial code reuse, and therefore less code to maintain. Python code is succinct compared to most other languages: a lot can be accomplished in only a few lines. Finally, Python is free and open source, and encourages open source software development. This is partly due to its interpreted nature: the source code and executable are typically one and the same.

Python has a stable and feature-rich numeric library called NumPy¹ which provides an N-dimensional array object. NumPy arrays can be subjected to vectorized operations, most of which call static C functions, allowing them to run almost as fast as pure C code. Yet, these operations remain accessible from within succinct Python code. NumPy turns Python into an effective replacement

¹<http://numpy.org>



for MatLab (The MathWorks, Natick, MA, USA), and is used extensively by dimstim, spyke, and neuropy.

While all three projects presented here were written in Python, their use and implementation are very different. Dimstim is script based and is run from the system's command line. Spyke has a graphical user interface (GUI) and looks like a native application, while neuropy is typically accessed from the Python command line as a library. Here, we explore some of the features and benefits of Python and its many add-on packages for the electrophysiologist, by introducing our own three packages as detailed working examples.

DIMSTIM: VISUAL STIMULUS GENERATION

In our experiments, we needed a way to display and control a wide variety of stimuli with many different parameters, often shuffled with respect to each other in various ways. Since spike times are acquired at sub-millisecond temporal resolution, and since precise spike timing may play a role in neural coding (Mainen and Sejnowski, 1995; VanRullen and Thorpe, 2002), we also wanted high temporal precision in the stimulus. Our prior stimulus software was written in Fortran and ran under DOS with a 32-bit extender. It was written for the 8514/A graphics standard which has now lapsed. The last graphics cards to support it were limited in the size and speed of movie frames they could draw to screen. Moreover, these cards were limited to a screen refresh rate of 100 Hz at our desired resolution. We found significant artefactual phase-locking of responses in visual cortex at this frequency (Blanche, 2005), which has been a concern reported elsewhere (Williams et al., 2004; Wollman and Palmer, 1995). For these reasons, we needed a better solution.

Dimstim displays full-screen stimuli at a refresh rate of 200 Hz, providing precise control of the display at 5 ms intervals without frame drops. Stimuli include manually controlled, drifting, and flashed bars and gratings, sparse noise, and m-sequence noise (Golomb, 1967) and natural scene movies. Stimulus parameters can be shuffled with or without replacement, independently or in covariation with each other. Parameters include spatial location and phase, orientation, speed, duration, size, mask, contrast, brightness,

and spatial and temporal frequencies. Each stimulus session is fully specified by its own user-editable script. A copy of the script, and an index of the contents of the screen on each screen refresh, are sent to the acquisition computer, for simultaneous recording of stimulus and neuronal responses.

Dimstim relies heavily on the Vision Egg² library (Straw, 2008) to generate stimuli. The Vision Egg uses the well-established OpenGL³ graphics language, which thanks to the demands of video games, is now supported by all modern video cards on all major platforms. We currently use an Nvidia GeForce 7600 graphics card running under Windows XP. Stimuli are displayed on a 19" Iiyama HM903DTB and a 22" HM204DTA CRT monitor, two of only a handful of consumer monitors that are capable of 800 × 600 resolution at 200 Hz. Unfortunately, like most other CRTs, these particular models have now been discontinued, but used ones may still be available. Hopefully the timing of LCD monitors will improve such that they can replace CRTs for temporally precise stimulus control.

Multitasking operating systems (OSes) present a challenge for real-time control of the screen. Often, the OS will decide to delay an operation to maintain responsiveness in other areas. This can lead to frame drops, but can be mitigated by increasing the priority of the Python process. Setting the process and thread priorities to their maximum levels in the Vision Egg completely eliminated frame drops in Windows XP, but with the unfortunate loss of mouse and keyboard polling. In dimstim, this meant that the user had no way of interrupting the stimulus script, other than by resetting the computer. Moving to a computer with a dual core CPU alleviated this problem, as the maximum priority Python process was delegated to one core without interruption, while other OS tasks such as keyboard polling ran normally on the second core.

Dimstim communicates stimulus parameters on a frame-by-frame basis to the acquisition computer via a PCI digital output board (DT340, Data Translations, Marlboro, MA, USA), for

²<http://visionegg.org>

³<http://opengl.org>

simultaneous recording of stimulus timing alongside neuronal responses. Parameters are described by sending the row index of a large lookup table (“sweep table”) on every screen refresh. The sweep table contains all the combinations of the dynamic parameters, i.e. those stimulus parameters that can vary from one screen refresh to the next.

The digital output board is controlled by its driver’s C library. Because Python is written in C (other implementations also exist), it has a C application programming interface (API), and extensions to Python can be written in C. We wrote such an extension to interact with the board’s C library, but today this is no longer necessary. A new built-in Python module called “ctypes” now allows interaction with a C library on any platform directly from within Python code. This is much simpler, as it removes the need to both write and compile C extension code using Python’s somewhat tedious C API. If `dimstim` were rewritten today, `ctypes` would be the method of choice. `Dimstim` includes a demo (`oldda_demo.py`) of how to use `ctypes` to directly interact with Data Translations’ Open Layers data acquisition library. Libraries for cards from other vendors (such as National Instruments’ NI-DAQmx) can be similarly accessed.

Frame timing was tested with a photodiode placed on the monitor. The photodiode signal, along with the raster signal from the video card and the digital outputs from the stimulus computer, were all recorded simultaneously. We discovered that the contents of the screen always lagged by one screen refresh, due to OpenGL’s buffer swapping behaviour (Straw, 2008). This was corrected for by adding one frame time (5 ms) to the timestamp of the digitized raster signal in the acquisition system.

Gamma correction was used to ensure linear control of screen luminance. Several levels of uncorrected luminance were measured with a light meter (Minolta LS-100) and fit to a power law expression to determine the exponent corresponding to the gamma value of the screen (Blanche, 2005; Straw, 2008). Gamma correction can be set independently for each script, or globally across all scripts in `dimstim`’s config file.

Natural scene movies used by `dimstim` were filmed outdoors with an ordinary compact digital camera (Canon PowerShot SD200) with 320×240 resolution at 60 frames per second (fps). Unfortunately, this camera could record no more than 1 min of video at a time. To generate longer movies, multiple clips were filmed in succession, while keeping the camera as motionless as possible between the end of one clip and the start of the next. Concatenation of and conversion from multiple colour .avi files to a single uncompressed greyscale movie file was done using David McNab’s `y4m`⁴ package. Processed movies were displayed in `dimstim` with the same visual angle subtended by the camera, at 67 fps (three 5 ms screen refreshes per movie frame).

USAGE

`Dimstim`’s config file stores default values for a variety of generic parameters that apply to most stimuli. These parameters include spatial location, size, orientation offset, and temporal and spatial frequencies. For simplicity, all spatial parameters are specified in degrees of visual angle. The config file can be edited by hand, but

the typical procedure when optimizing parameters for the current neural population is to run a manually controlled bar or grating stimulus. For user convenience, the stimulus is shown simultaneously on two displays driven by two video outputs from the graphics card: one for the animal, and one for the user. The parameters of the manual stimulus are controlled in real-time with the mouse and keyboard. Once the user is satisfied, the parameters are saved to the config file. These can later be retrieved by an experiment script for use as default values.

An example script for a drifting sinusoidal grating experiment is shown in **Figure 2**. The script works in a bottom-up fashion. First, objects for storage of static and dynamic parameters are instantiated (“s” and “d” respectively, lines 5–6). To these are bound various different parameters as attributes (denoted by a “.”). In this example, most values are declared directly by the script, but two static parameters, grating orientation offset and gamma correction, are retrieved from their defaults in the config file, using the `dimstim` config parser object named “dc” (lines 15 and 23). Dynamic parameters, if assigned a list of multiple values, will iterate over those values over the course of the experiment. In this case, grating orientation, spatial frequency, and temporal frequency are all assigned multiple values (lines 28, 36, 38). The rest remain constant for the duration of the experiment. In order to describe their interdependence and shuffling, each multiple-value dynamic parameter must be declared as a “Variable” (lines 53–55). Variables with the same dimension value (“dim” keyword argument) covary with each other, and must therefore all have the same number of values and the same shuffle flag. Variables with different dimension values vary independently in a combinatorial fashion, with the lowest numbered dimension varying slowest, and the highest varying fastest. This is implemented by dynamically generating a string object containing Python code with the correct number of nested for loops (equalling the number of independent variables specified in the script), and then executing the contents of the string with Python’s `exec()` function (see the `dimstim.Core.SweepTable` class). Next, the number of times to cycle through all combinations, and the frequency at which to insert a blank screen sweep (for determining baseline firing rates) are specified in their own objects (lines 57–58). Finally, all these objects are passed together to the `Grating` class (which like all other `dimstim` stimuli, inherits from the `Experiment` class) to instantiate a `Grating` experiment object, and the experiment is run (lines 62–65). With 12 orientations, 6 spatial frequencies, and 4 temporal frequencies, this experiment has 288 unique parameter combinations, presented in shuffled order. Each is presented four times for a total of 1152 stimulus sweeps, lasting 4 s each, for a total experiment time of about 77 min (not including blank sweeps).

Before running, various checks are done to alert for any obvious errors in the user edited script. Then, a copy of the entire script is sent to the acquisition computer. This makes it possible to later reconstruct the sweep table for analysis, and even replay the entire experiment exactly, without the need for access to the original script on the stimulus computer. To ensure accurate timing, stimuli run only on the animal display, while the user display shows the system command line. In between experiments when no stimuli are running, a blank grey desktop is shown on the animal display. Scripts can be paused or cancelled using the keyboard.

⁴<http://freenet.org.nz/y4m>

```

1  from dimstim.Constants import dc # dimstim config parser
2  from dimstim.Core import StaticParams, DynamicParams, Variable, Variables, Runs, BlankSweeps
3  from dimstim.Grating import Grating
4
5  s = StaticParams() # stores static parameters
6  d = DynamicParams() # stores dynamic parameters
7
8  """Static parameters remain constant during the entire experiment"""
9
10 # pre-experiment duration to display blank screen (sec)
11 s.preexpSec = 1
12 # post-experiment duration to display blank screen (sec)
13 s.postexpSec = 1
14 # orientation offset (deg)
15 s.oriOff = dc.get("Manbar0", "oriOff") # retrieve from config file
16 # grating width (deg)
17 s.widthDeg = 60
18 # grating height (deg)
19 s.heightDeg = 60
20 # mask, one of: None, "gaussian", or "circle"
21 s.mask = None
22 # screen gamma: None, or single value, or RGB 3-tuple
23 s.gamma = dc.get("Screen", "gamma") # retrieve from config file
24
25 """Dynamic parameters can potentially vary from one sweep to the next"""
26
27 # grating orientation relative to oriOff (deg)
28 d.ori = range(0, 360, 30) # 0 to 330 deg in steps of 30
29 # grating x position relative to origin (deg)
30 d.xposDeg = 0
31 # grating y position relative to origin (deg)
32 d.yposDeg = 0
33 # mask diameter (deg), ignored if s.mask is None
34 d.diameterDeg = 0
35 # spatial frequency (cycles/deg)
36 d.sfreqCycDeg = [0.05, 0.1, 0.2, 0.4, 0.8, 1.6]
37 # temporal frequency (cycles/sec)
38 d.tfreqCycSec = [0.5, 1, 2, 5]
39 # grating phase to begin each sweep with (+/- deg)
40 d.phase0 = 0
41 # mean luminance (0-1)
42 d.ml = 0.5
43 # contrast (0-1), >> 1 gives square grating, < 0 reverses contrast
44 d.contrast = 1
45 # background brightness (0-1)
46 d.bgbrightness = 0.5
47 # sweep duration (sec)
48 d.sweepSec = 4
49 # post-sweep duration to display blank screen (sec)
50 d.postSweepSec = 0
51
52 vs = Variables() # stores Variable objects, used to generate sweep table
53 vs.ori = Variable(vals=d.ori, dim=0, shuffle=True)
54 vs.sfreqCycDeg = Variable(vals=d.sfreqCycDeg, dim=1, shuffle=True)
55 vs.tfreqCycSec = Variable(vals=d.tfreqCycSec, dim=2, shuffle=True)
56
57 runs = Runs(n=4, reshuffle=False) # step through sweep table 4 times in the same order
58 bs = BlankSweeps(T=20, sec=2, shuffle=False) # display blank sweep every 20 sweeps for 2 seconds
59
60 """Create the Grating experiment and run it"""
61
62 e = Grating(script=__file__, # this script's file name
63             static=s, dynamic=d, variables=vs,
64             runs=runs, blanksweeps=bs)
65 e.run()

```

FIGURE 2 | A dimstim script describing a drifting sinusoidal grating. Such scripts may be edited at will, and are the primary way the user interacts with dimstim. After some error checking, the script executes from the system's

command line, to which status messages are printed. Comments, denoted by # and """ in Python, are highlighted in red. Line numbers have been added for reference. See text for more details.

SPYKE: WAVEFORM VISUALIZATION AND SPIKE SORTING

Once neural waveform and stimulus data were saved to disk by our acquisition system (written in Delphi), we needed a way to retrieve the data for visualization and spike sorting. Our existing program for this, also written in Delphi, had some bugs and missing features. However, the Delphi environment required a license, the program would only run in Windows, and the code was more procedural than object-oriented. In particular, some of the code had blocks (if statements, for/while statements) that were nested many

layers deep, making it difficult to follow. “Flat is better than nested” (Peters, 2004) is another Python philosophy. Several short, shallow blocks of code are easier to understand and manage than one long deep block. We decided to start from scratch in Python.

Spyke has a cross-platform GUI with native widgets for data visualization and navigation, and spike sorting (**Figure 3**). Spike waveforms are displayed in two ways: spatially according to the polytrode channel layout (spike window), and vertically in chart form (chart window). Local field potential (LFP) waveforms are

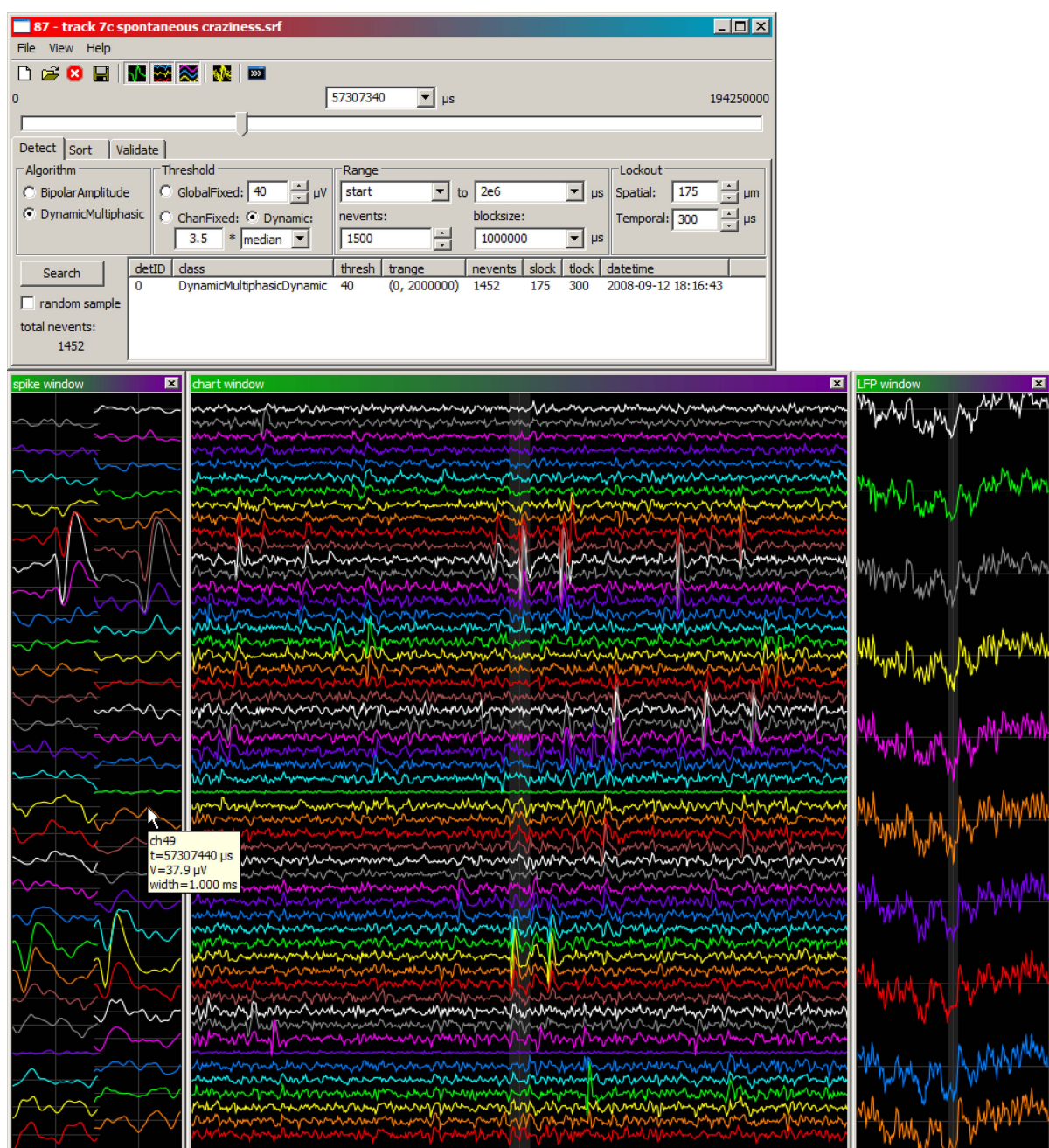


FIGURE 3 | Main spyke window (top), with data windows (bottom) showing high-pass waveforms in polytrode layout (left) and chart layout (middle). A third data window shows the low-pass LFP waveforms (right)

concurrently recorded from a subset of channels (colour coded). All data are centred on the same timepoint. The shaded region in the middle of both the chart and LFP windows represents the time range spanned by the window to its left.

also displayed vertically in chart form (LFP window). Polytrode channels are closely spaced (43–75 μm) over two or three columns (**Figure 1A**). A single spike can generate a signal on multiple channels, hence the need to visualize waveforms according to their polytrode channel layout. Channels are colour-coded to make them easy to distinguish and align across windows. Spyke looks and behaves like a native GUI application, with menus, buttons, and resizable windows. Navigation is mouse and keyboard based. A horizontal slider and combo box at the top of the main spyke window control file position in time. Left and right arrow keys, and page up and page down keys step through the data with single timepoint or 1 ms resolution respectively. Clicking on any data window (spike, chart, or LFP) centres all three windows on that timepoint. Holding CTRL and scrolling the mouse wheel over a data window zooms it in or out in time. Holding CTRL and clicking on a channel enables or disables it. Hovering the mouse over a data window displays a tooltip with the timestamp, channel, and voltage currently under the mouse cursor.

Spyke uses the wxPython⁵ library for its GUI. This is a Python interface to the wxWidgets C++ GUI library which generates widgets on Windows, Linux, and OSX. Now well over a decade old (Rappin and Dunn, 2006), wxPython is a stable library that has adapted to changing OSes. Widgets include everything from windows, menus, and buttons, to more complex list and tree controls. WxPython has a big advantage over other GUI libraries in its use of widgets that are native to the OS the program is running on, such that they look and behave identically to normally created widgets in that OS. WxGlade⁶ was used to visually lay out the GUI. Itself a wxPython based GUI application, wxGlade takes the programmer's visual layout and automatically generates the corresponding layout code in Python. This code can then be included in the programmer's own code base, typically by defining a class that inherits from the automatically generated code. Although wxGlade is not necessary for writing a GUI with wxPython, we found it much faster and easier than writing all of the layout code by hand.

Unfortunately, some widgets are inherently different on different OSes. Writing and testing a wxPython GUI on only one OS will therefore not guarantee perfect functionality on another. To do so would require checking for the current OS, and implementing certain things differently depending on the OS. Spyke does not currently do this, and has so far only been thoroughly tested in Windows. A cross-platform GUI library faces many challenges. Although wxPython is one of the best (Rappin and Dunn, 2006), it has bugs⁷ – some of them longstanding – that had to be worked around in spyke.

Although the widgets are handled by wxPython, waveforms are plotted using matplotlib⁸. Matplotlib is a 2D plotting library for Python that generates publication quality figures. It has two interfaces: one that mimics the familiar plotting commands of MatLab, and another that is much more object-oriented. Spyke embeds matplotlib figures within wxPython windows. Scaling of plots is handled automatically by matplotlib, such that when the wxPython window is resized by dragging its corner or edge, the plotted traces inside

resize accordingly. Another benefit of matplotlib is its antialiasing abilities, providing beautiful output with subpixel resolution. There is some performance penalty for using such a high level drawing library, but performance is fast enough on fairly ordinary hardware (Pentium M 1.6 GHz notebook), even when scrolling through 54 channels of data with thousands of data points on screen at a time. More importantly, matplotlib makes plotting very easy to do.

The data acquisition files are complex, with different types of data multiplexed throughout the file. On opening, the file must be parsed to determine the number and offset values of hundreds of thousands of records in the file. For multi GB files, this can take up to a few minutes. To deal with this, the parsing information is saved to disk for quicker future retrieval. This is done using Python's pickle module, which can take a snapshot of almost any Python object in memory, serialize it, and save it to disk as a "pickle". A pickle can then later be restored (unpickled) to memory as a live Python object, even on a different platform. In this case, a custom written File object containing all of the parse information is saved to disk as a .parse file of only a few MB in size. Restoring from the .parse file is about an order of magnitude faster than reparsing the entire acquisition file.

Segments of waveform data are loaded from the acquisition file, Nyquist interpolated, and sample-and-hold delay (SHD) corrected on the fly as needed (Blanche and Swindale, 2006). Interpolation is performed to improve spike detection, and Nyquist interpolation is the optimum method of reconstructing a bandwidth-limited signal at arbitrary resolution. To do so, a set of sinc function kernels is generated (one kernel per interpolated data point, each kernel with a different phase offset) and convolved with the data. For SHD correction, a different set of kernels is generated for each channel. Correcting for each channel's SHD requires appropriate modification of the phase offset of each kernel for that channel. For example, interpolating from 25 to 50 kHz with SHD correction requires two appropriately phase corrected kernels per channel. Each kernel is separately convolved with the data (using `numpy.convolve()`), and the resulting data points are interleaved to return the final interpolated waveform.

SPIKE SORTING

Spike sorting is done by template matching (Blanche, 2005). Event detection is the first step in generating the required multichannel spike templates. Two event detection methods are currently implemented. The "bipolar amplitude" method looks for simple threshold crossings of either polarity. The "dynamic multiphasic" method searches for two consecutive threshold crossings of opposite polarity within a defined period of time. The second crossing's threshold is dynamically set according to the amplitude of the first phase of the spike. For both methods, primary thresholds are calculated separately for each channel, based on the standard deviation or median noise level of either the entire recording or of a narrow sliding window thereof. Spatiotemporal detection lockouts prevent double triggering off of the same spike, while minimizing the chance of missed spikes.

Some algorithms, such as these event detection methods, cannot be easily vectorized and require a custom loop. Due to its dynamic typing and interpreted nature, long loops are slow to execute in Python. For the majority of software development, this is not an

⁵<http://wxpython.org>

⁶<http://wxglade.sf.net>

⁷See bugs #626 and #2307 at <http://trac.wxwidgets.org>

⁸<http://matplotlib.sf.net>

issue. Developer time is usually much more valuable than CPU time (Hetland, 2005), but numerically intensive software is the exception. Writing fast Python extensions in C has always been possible, but the C interface code required by Python's API is tedious to write, and writing in C eliminates the convenience of working in Python syntax. To get around this, the Cython⁹ package (a fork of the Pyrex package) specifies a sublanguage almost identical to Python, with some extra keywords to declare loop variables as static C types. After issuing the standard `python setup.py build` command, such code is automatically translated into an intermediary C file including all of the tedious interface code. This is subsequently compiled into object code and is accessible as a standard C extension module from within Python, just as a handwritten C extension would be. This yields the computational speed of C loops when needed, with the developmental speed, convenience and familiarity of Python syntax to implement them. Cython was used to write the custom loop that iterates over timepoints and channels for each of the event detection methods. For 25 kHz sampled waveform data on 54 channels, this amounts to 1.35 million iterations per second of data. On an average single-core notebook computer (Pentium M 1.6 GHz), this loop runs at about 5× real time.

The data is partitioned into blocks (typically 1 s long), and each is searched independently, allowing multiple core CPUs to be exploited. Search speed scales roughly proportionally with the number of cores available. Due to the “global interpreter lock” (GIL) in the C implementation of Python, multiple processes must typically be used instead of multiple threads to take advantage of multiple cores. Unfortunately, a process can require significantly more memory and more time to create than a thread. There are ways around the GIL, but the best solution for *spyke* is not yet clear.

Search options are controlled in the “detect” tab in *spyke*'s main window (Figure 3). Searches can be limited to specific time ranges in the file, in the number of events detected, and whether to search linearly or randomly. Random sampling is important to build up a temporally unbiased collection of detected events with which to build templates. Searching for the next or previous spike relative to the current timestamp can be done quickly using the keyboard. Searches are restricted to enabled channels, allowing for a targeted increase in the number of events belonging to a spatially localized template. This is useful for building up templates of neurons that rarely fire.

When a search completes, the sort window (Figure 4A) opens and is populated with any newly detected events. The user then visually sorts the detected events (typically only a fraction of all spikes in the recording) into templates corresponding to isolated neurons. This is accomplished by plotting spikes over top of each other. Any number of event or template mean waveforms can be overplotted with each other. Although the mouse may be used, keyboard commands are more efficient for toggling the display of events and templates, and moving events and keyboard focus around between the sorted template tree (left column) and unsorted event list (right column). The event list has sortable columns for event ID, maximum channel, timestamp, and match error. All the events in the list can be matched against the currently selected template, and

those match errors populate the error column. Sorting the event list by maximum channel or match error makes manual template generation much easier, because it clusters similar events close to each other in the unsorted event list.

Once templates have been generated, a full event detection is run across the whole recording, and the templates are matched against each detected event. Or, each template can be slid across the recording and matched against every timepoint in the recording (Blanche et al., 2005). Either way, matching to target and non-target spikes or noise generally yields a non-overlapping bimodal error distribution. For each template, a threshold is manually set at the trough between the two peaks in the distribution, and events whose match errors fall below this threshold are classified as spikes of that template.

At any point in the sorting process, the entire “Sort” session object, which among other information includes detected events, generated templates, and sorted spikes, can be saved to disk as a .sort file, again using Python's `pickle` module. Sort sessions can then be restored from disk and sorting can resume in *spyke*, or their sorted spike times can be used for spike train analysis (see *neuropy* section). Waveform data for detected events and sorted spikes is saved within the .sort file. This increases the file size, but allows for review of detected and sorted spikes without the need to access the original multi GB continuous data acquisition file.

Integrated into *spyke* is Patrick O'Brien's PyShell (Figure 4B), an enhanced Python command line that is part of the wxPython package. This permits live command line inspection and modification of all objects comprising *spyke*. This was, and continues to be, a very useful tool for testing existing features and for developing new ones. *Neuropy* (or almost any other Python package) can be imported and used directly from this command line. For example, spike sorting validation is not yet implemented in *spyke*'s GUI, but all of *neuropy*'s functionality including autocorrelograms (to check refractory periods) can be accessed by typing `import neuropy` in *spyke*'s PyShell.

NEUROPY: SPIKE TRAIN ANALYSIS

After spike sorting, we needed a way to analyse spike trains and their relation to stimuli. Our initial decision was to use MatLab for spike train analysis, and we soon developed a collection of MatLab scripts for the job, with one function per .m file. For example, one .m file would load each neuron's data from disk and return all of them in a cell array of structures. This was highly procedural instead of object-oriented. Furthermore, the code became difficult to manage as each additional function required an additional .m file. We were also faced with out of memory errors, limited GUI capabilities, and a high licensing cost.

Although MatLab's toolboxes are a major benefit, SciPy¹⁰ (Jones et al., 2001), an extensive Python library of scientific routines, provides most of the equivalent functionality. Much of SciPy is a wrapper for decades-old, highly tested and optimized Fortran code. Another package, `mlabwrap`¹¹, allows a licensed MatLab user to access all of MatLab's functionality, including all of its toolboxes, directly from within Python. Although in the end we did not need

⁹<http://cython.org>

¹⁰<http://scipy.org>

¹¹<http://mlabwrap.sf.net>

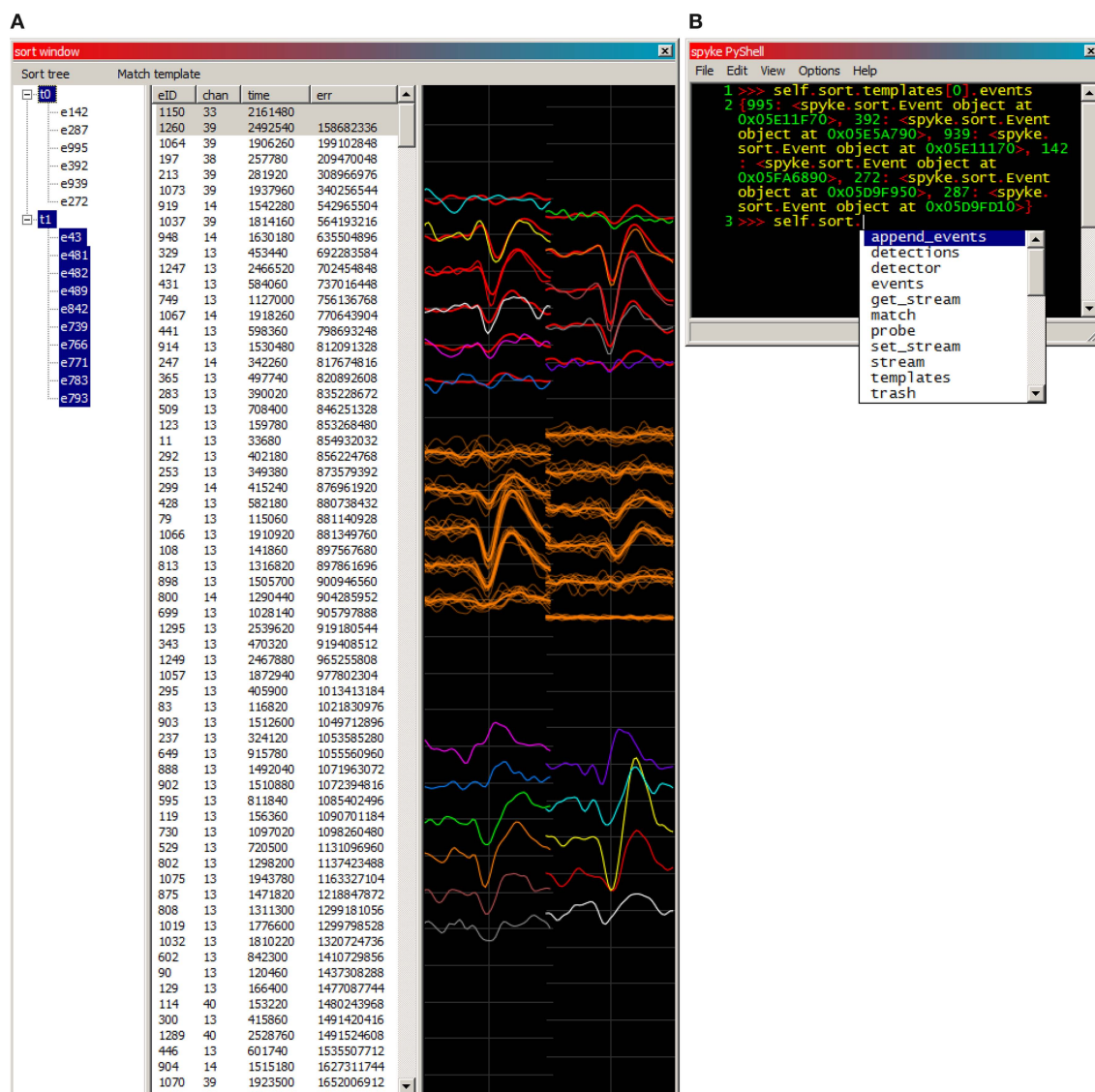


FIGURE 4 | (A) An example of spyke's sort window. Templates and their member spikes are represented in the tree (left), and unsorted detected events in the list (middle). Selecting a template or event in either the tree or the list plots its waveform (right). The tree currently has keyboard focus, making its selections more distinctly coloured than those of the list. Unsorted events have colour coded channels, while each template (and its member spikes) has a single identifying colour. Here, template 0 (red), a putative neuron near the top of the polytrode, has 6 member spikes, and its mean waveform is being overplotted with an unsorted event (#1260, multicoloured),

which fits quite well. Template 1 (orange) and all of its member spikes are plotted near the middle of the polytrode. Also plotted further down is another unsorted event (#1150, multicoloured), which obviously does not fit either template. The error values listed are from a match against template 0. **(B)** The integrated PyShell window exposes all of spyke's objects and functionality at the Python command line. Template 0's dictionary (a mapping from names to values) of its 6 member events is referenced and returned on lines 1–2. The "Sort" object's attributes and methods are displayed in a popup on line 3.

to use mlabwrap, its existence erased any remaining hesitations about switching to Python for analysis.

A data-centric object hierarchy (Figure 5A) quickly emerged as a natural way to organize neuropy. Each object in the hierarchy has an attribute that references its parent object, as well as all of its child objects. Specifically, "Data" is an abstract object from which all "Animals" are accessible. Each Animal has polytrode "Tracks", each Track has "Recordings", and each Recording has both "Sorts" (spike sorting sessions) and "Experiments" (which describe stimuli).

Finally, each Sort contains a number of "Neurons", one of whose attributes is a NumPy array of spike times.

Neuropy relies on a hierarchy of data folders on the disk with a fairly rigid naming scheme, such that animal, track, recording, experiment, and sort IDs can be extracted from file and folder names. This forces the user to keep sorted data organized. All objects have a unique ID under the scope of their parent, but not necessarily under the scope of their grandparent. All data can be loaded in at once by creating an instance of the Data class and then

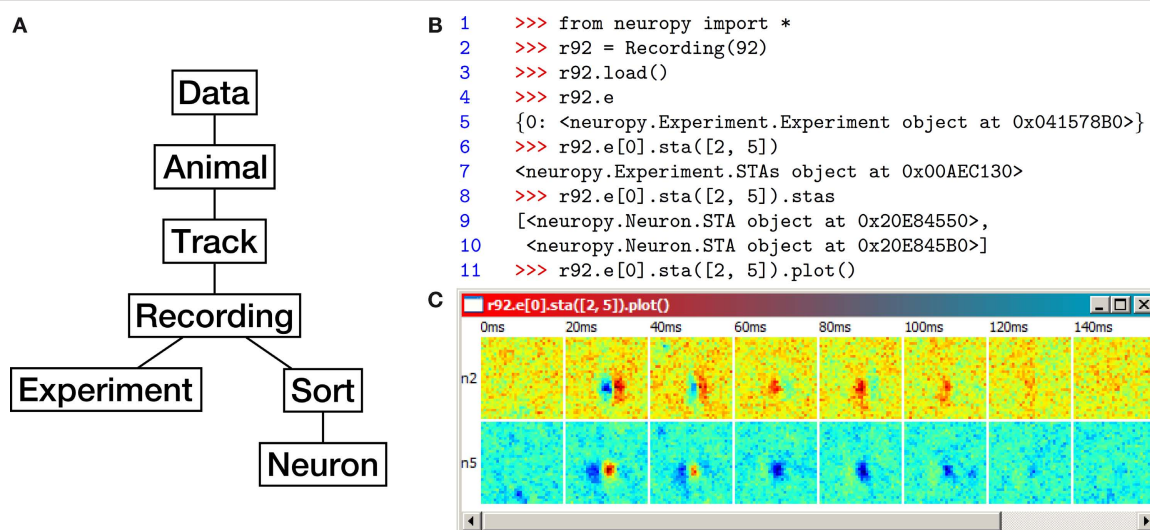


FIGURE 5 | (A) Neuropy's object hierarchy. **(B)** Example code using neuropy to plot the spike-triggered average (STA) of two neurons in response to an m-sequence noise movie (see text for details). **(C)** The resulting plot window.

Each row corresponds to a neuron, and each column corresponds to the STA within a fixed time range following the m-sequence white noise stimulus. ON responses are red, OFF responses are blue.

calling its `.load()` method. However, most often only a subset of data is needed, such as only the data from a given animal, track, or recording. For example, an object representing recording 92 from the default track of the default animal can be instantiated by typing `Recording(92)` at the command line. This recording's data can then be loaded from disk into the object by calling its `.load()` method. Default animal and track IDs can be modified from the command line. A recording loads the neurons from its default sort, which can also be modified.

Some analyses are written as simple methods of one of the data objects, but most have their own separate class which is instantiated by a data object's method call. Many analyses generate plots, some of them interactive (such as the population spike raster plot), again using matplotlib and wxPython. Currently implemented analyses include interspike interval histograms, instantaneous firing rates and their distributions, cross-correlograms and autocorrelograms, and spike-triggered averages (STAs) (Dayan and Abbott, 2001). More specialized analyses include binary codes of population spike trains, their correlation coefficient distributions, maximum entropy Ising modelling of such codes (using `scipy.maxent`), and several other related analyses (Schneidman et al., 2006; Shlens et al., 2006; Spacek et al., 2007). Because of the data-centric organization, new analyses are easy to add.

Neuropy is used interactively as a library from the Python command prompt, usually in an enhanced shell such as PyShell (Figure 4B) or the more widely used IPython¹². An example of neuropy use is shown in Figure 5B, which calculates and plots the STA of neurons 2 and 5 of the default animal and track. The STA estimates a neuron's spatiotemporal receptive field by averaging the stimulus (in this case, an m-sequence noise movie) at fixed time intervals preceding each spike. Recording 92 was recorded during m-sequence noise movie playback, and is used in this example.

Line 1 imports all of neuropy's functionality into the local namespace. Next, an object representing recording 92 is instantiated and bound to the name `r92` for convenience, and its data is loaded from disk (lines 2–3). Its dictionary of available experiments is requested and printed out (lines 4–5); only one experiment is available, with ID 0. STAs are calculated with respect to this experiment by calling its `.sta()` method and passing the IDs of the desired neurons (line 6). The calculated STAs are returned in an "STAs" object, which upon further inspection contains two "STA" objects, one per requested neuron (lines 7–10). Finally, the STAs object's `.plot()` method is called with default options, displaying the result for both neurons (Figure 5C).

Python's object orientation has benefits even at the command line. It allows the user to quickly discover what methods and attributes are available for any given object, eliminating the need to recall them from memory (Figure 4B). Instead of immediately returning the raw result or plotting it, most analyses in neuropy return an analysis object, which usually has `.calc()` and `.plot()` methods. The `.calc()` method is run automatically on instantiation, and the results are stored as attributes of the analysis object. Settings used to do the calculation are also stored as attributes. These can be modified, and `.calc()` can be called again to update the result attributes. Once satisfied with the calculation, the user can call the `.plot()` method. This can be done several times to generate different plots with different plot settings. Each time a new plot is generated, it does so from the existing results, saving on unnecessary recalculation time.

CONCLUSION

We have described Python packages for three tasks pertinent to systems neuroscience: visual stimulus generation, waveform visualization and spike sorting, and spike train analysis. Python allowed us to meet these software challenges with a level of performance not normally associated with a dynamically typed interpreted language. Performance challenges included time-critical display and

¹²<http://ipython.scipy.org>

communication of visual stimuli, parsing and streaming of multiplexed data from GB sized files, on the fly Nyquist interpolation and SHD correction, fast execution of non-vectorizable algorithms, and parallelization. Other challenges, whose solutions were simpler than in a statically compiled language, included a cross-platform native GUI, the storage and retrieval of relatively complex data structures to and from file (.parse and .sort files), and a command line environment for interactive data analysis.

Dimstim is the oldest of the three packages, and the most stable. Spyke is the most recent and remains under heavy development, while new analyses are added to neuropy as needed. As with most other Python packages, all three can be used alone or from within another Python module. All three depend on each other to a limited extent. Neuropy relies on the stimulus description and timing signals generated by dimstim, and on the spike sorting results from spyke. Spyke can use parts of neuropy for spike sorting validation. These three packages depend on many other open source packages, which themselves rely on yet other packages (e.g. the Vision Egg currently depends on PyOpenGL and PyGame). Modularity and code reuse is thus maximized across the community.

Because it greatly encourages object-oriented programming, Python code is easier to organize and reuse than MatLab code. This is important for scientific code which tends to continually evolve as new avenues are explored. Often, scientific code is quickly written

and bug-tested, used once or twice, and then forgotten about, with little chance of re-use outside of copying and pasting. Python has reduced this tendency for us. Its object orientation and excellent error handling have also helped to reduce bugs.

Finally, Python was chosen for these projects for its clear, succinct syntax. Dimstim, spyke, and neuropy have roughly 3000, 5000, and 4000 lines of code respectively (excluding comments and blank lines). Fewer lines make code maintenance easier, not just because there is less code to maintain, but also because each line is closer to all other lines, making it easier to navigate. Concise syntax also makes collaboration easier.

We encourage others in neuroscience to consider Python for their programming needs, and hope that our three examples (available at <http://swindale.ecc.ubc.ca/code>) may be of use to others, whether directly or otherwise. Rallying around a common open-source language may help foster efforts to increase sharing of data and code, efforts deemed necessary (Teeters et al., 2008) to push forward progress in systems neuroscience.

ACKNOWLEDGEMENTS

Keith Godfrey wrote dimstim's C extension to interface with the Data Translations board. Reza Lotun contributed code to early versions of spyke. Funding came from grants from the Canadian Institutes of Health Research, and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

- Blanche, T. J. (2005). Large scale neuronal recording. Ph.D. dissertation, University of British Columbia, Vancouver, BC.
- Blanche, T. J., Spacek, M. A., Hetke, J. F., and Swindale, N. V. (2005). Polytrodes: high-density silicon electrode arrays for large-scale multiunit recording. *J. Neurophysiol.* 93, 2987–3000.
- Blanche, T. J., and Swindale, N. V. (2006). Nyquist interpolation improves neuron yield in multiunit recordings. *J. Neurosci. Methods* 155, 81–91.
- Dayan, P., and Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA, MIT Press.
- Golomb, S. W. (1967). *Shift Register Sequences*. San Francisco, Holden-Day.
- Hetland, M. L. (2005). *Beginning Python: From Novice to Professional*. Berkeley, CA, Apress.
- Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: open source scientific tools for Python. <http://scipy.org>.
- Langtangen, H. P. (2008). *Python Scripting for Computational Science*, 3rd Edn. Berlin, Springer-Verlag.
- Lutz, M. (2006). *Programming Python*, 3rd Edn. Sebastopol, CA, O'Reilly.
- Mainen, Z. F., and Sejnowski, T. J. (1995). Reliability of spike timing in neocortical neurons. *Science* 268, 1503.
- Peters, T. (2004). The Zen of Python. <http://www.python.org/dev/peps/pep-0020>.
- Rappin, N., and Dunn, R. (2006). *wxPython in Action*. Greenwich, CT, Manning.
- Schneidman, E., Berry, M. J. II, Segev, R., and Bialek, W. (2006). Weak pairwise correlations imply strongly correlated network states in a neural population. *Nature* 440, 1007–1012.
- Shlens, J., Field, G. D., Gauthier, J. L., Grivich, M. I., Petrusca, D., Sher, A., Litke, A. M., and Chichilnisky, E. J. (2006). The structure of multi-neuron firing patterns in primate retina. *J. Neurosci.* 26, 8254–8266.
- Spacek, M. A., Blanche, T. J., Seamans, J. K., and Swindale, N. V. (2007). Accounting for network states in cortex: are (local) pairwise correlations sufficient? *Soc. Neurosci. Abstr.* 33, 790.1. <http://swindale.ecc.ubc.ca/Publications>.
- Straw, A. D. (2008). Vision Egg: an open-source library for realtime visual stimulus generation. *Front. Neuroinform.* 2, 4.
- Teeters, J. L., Harris, K. D., Millman, K. J., Olshausen, B. A., and Sommer, F. T. (2008). Data sharing for computational neuroscience. *Neuroinformatics* 6, 47–55.
- VanRullen, R., and Thorpe, S. J. (2002). Surfing a spike wave down the ventral stream. *Vis. Res.* 42, 2593–2615.
- Williams, P. E., Mechler, F., Gordon, J., Shapley, R., and Hawken, M. J. (2004). Entrainment to video displays in primary visual cortex of macaque and humans. *J. Neurosci.* 24, 8278–8288.
- Wollman, D. E., and Palmer, L. A. (1995). Phase locking of neuronal responses to the vertical refresh of computer display monitors in cat lateral geniculate nucleus and striate cortex. *J. Neurosci. Methods* 60, 107–113.

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

Received: 18 September 2008; paper pending published: 04 November 2008; accepted: 19 December 2008; published online: 28 January 2009.

Citation: Spacek M, Blanche T and Swindale N (2009) Python for large-scale electrophysiology. *Front. Neuroinform.* (2009) 2:9. doi: 10.3389/neuro.11.009.2008

Copyright © 2009 Spacek, Blanche and Swindale. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



PyNN: a common interface for neuronal network simulators

Andrew P. Davison^{1*}, Daniel Brüderle², Jochen Eppler^{3,4}, Jens Kremkow^{5,6}, Eilif Müller⁷, Dejan Pecevski⁸, Laurent Perrinet⁶ and Pierre Yger¹

¹ Unité de Neurosciences Intégratives et Computationnelles, CNRS, Gif sur Yvette, France

² Kirchhoff Institute for Physics, University of Heidelberg, Heidelberg, Germany

³ Honda Research Institute Europe GmbH, Offenbach, Germany

⁴ Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany

⁵ Neurobiology and Biophysics, Institute of Biology III, Albert-Ludwigs-University, Freiburg, Germany

⁶ Institut de Neurosciences Cognitives de la Méditerranée, CNRS, Marseille, France

⁷ Laboratory of Computational Neuroscience, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

⁸ Institute for Theoretical Computer Science, Graz University of Technology, Graz, Austria

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Graham Cummins, Montana State
University, USA

Fred Howell, Textensor Limited, UK

*Correspondence:

Andrew Davison, UNIC, Bât. 32/33,
CNRS, 1 Avenue de la Terrasse, 91198
Gif sur Yvette, France.
e-mail: andrew.davison@unic.cnrs-gif.fr

Computational neuroscience has produced a diversity of software for simulations of networks of spiking neurons, with both negative and positive consequences. On the one hand, each simulator uses its own programming or configuration language, leading to considerable difficulty in porting models from one simulator to another. This impedes communication between investigators and makes it harder to reproduce and build on the work of others. On the other hand, simulation results can be cross-checked between different simulators, giving greater confidence in their correctness, and each simulator has different optimizations, so the most appropriate simulator can be chosen for a given modelling task. A common programming interface to multiple simulators would reduce or eliminate the problems of simulator diversity while retaining the benefits. PyNN is such an interface, making it possible to write a simulation script once, using the Python programming language, and run it without modification on any supported simulator (currently NEURON, NEST, PCSIM, Brian and the Heidelberg VLSI neuromorphic hardware). PyNN increases the productivity of neuronal network modelling by providing high-level abstraction, by promoting code sharing and reuse, and by providing a foundation for simulator-agnostic analysis, visualization and data-management tools. PyNN increases the reliability of modelling studies by making it much easier to check results on multiple simulators. PyNN is open-source software and is available from <http://neuralensemble.org/PyNN>.

Keywords: Python, interoperability, large-scale models, simulation, parallel computing, reproducibility, computational neuroscience, translation

INTRODUCTION

Science rests upon the three pillars of open communication, reproducibility of results and building upon what has gone before. In these respects, computational neuroscience ought to be in a good position, since computers by design excel at repeating the same task without variation, as many times as desired: reproducibility of computational results ought, then, to be a trivial task. Similarly, the Internet enables almost instantaneous transmission of research materials, i.e. source code, between labs.

However, in practice this theoretical ease of reproducibility and communication is seldom achieved outside of a single lab and a time frame of a few months or years. While a given scientist may easily be able to reproduce a result obtained a few months ago, precisely reproducing a result obtained several years ago is likely to be rather more difficult, and the general experience seems to be that reproducing the results of others is both difficult and time consuming: very many published papers lack sufficient detail to rebuild a model from scratch, and typographic errors are common.

Having available the source code of the model greatly improves the situation, but here still there are numerous barriers to reproducibility and to building upon previously published models. One is that source code can rapidly go out of date as computer architectures,

compiler standards and simulators develop. Another is that model source code is often not written with reuse and extension in mind, and so considerable rewriting to modularize the code is necessary. Probably the most important barrier is that code written for one simulator is not compatible with any other simulator.

Although many computational models in neuroscience are written from the ground up in a general purpose programming language such as C++ or Fortran, probably the majority use a special purpose simulator that allows models to be expressed in terms of neuroscience-specific concepts such as neurons, ion channels, synapses; the simulator takes care of translating these concepts into a system of equations and of numerically solving the equations. A large number of such simulators are available (reviewed in Brette et al., 2007), mostly as open-source software, and each has its own programming language, configuration syntax and/or graphical interface, which creates considerable difficulty in translating models from one simulator to another, or even in understanding someone else's code, with obvious negative consequences for communication between investigators, reproducibility of others' models and building on existing models.

However, the diversity of simulators also has a number of positive consequences: (i) it allows cross-checking – the probability of two

different simulators having the same bugs or hidden assumptions is very small; (ii) each simulator has a different balance between efficiency (how fast the simulations run), flexibility (how easy it is to add new functionality; the range of models that can be simulated), scalability (for parallel, distributed computation on clusters or supercomputers), and ease of use, so the most appropriate can be chosen for a given task.

Addressing the problems associated with an ecosystem of multiple simulators while retaining the benefits would greatly increase the ease of reproducibility of computational models in neuroscience and hence make it easier to verify the validity of published models and to build upon previous work.

There are at least two possible (and complementary) approaches to this. One is to enable direct, efficient communication between different simulators at run-time, allowing different components of a model to be simulated on different simulators (Ekeberg and Djurfeldt, 2008). This approach addresses the problem of building a model from diverse components, but still leaves the problem of having to use different programming languages, and does not enable straightforward cross-checking. The other approach is to develop a system for model specification that is simulator-independent. Translation then only has to be done once for each simulator and not once for each model.

Here we can take advantage of the recent, rapid emergence of the Python programming language as an alternative interface to several of the more widely-used simulators. Thus, for example, both NEURON and NEST may be controlled either via their original, native interpreter (Hoc and SLI, respectively) or via Python. More recent simulators (e.g. PCSIM, Brian) have Python as the only available scripting language. This widespread adoption of Python is probably due to a number of factors, including the powerful data structures, clean and expressive syntax, extensive library, maturity of tools for numerical analysis and visualization (allowing use of a single language for the entire modelling workflow from simulation to analysis to graphing), and the ease-of-use of Python as a glue language which allows computation-intensive code written in a low-level language such as C to be transparently accessed within high-level Python code.

Python alone does not address the translation problem (although it does make the translation process easier, since at least simple data structures such as lists and arrays are the same for each simulator), since neuroscience-specific concepts are still expressed differently. However, it is now possible to define a simulator-independent Python interface for neuronal network simulators and to implement automatic translation to any Python-enabled simulator. We have designed and implemented such an interface, PyNN (pronounced “pine”). In this paper we describe its design, concepts, implementation and use. We do not attempt here to provide a complete user guide – this may be found online at <http://neuralensemble.org/PyNN>.

DESIGN GOALS

When designing and implementing a common simulator interface, the following goals should be taken into account. These are the goals we have kept in mind when designing and implementing the PyNN interface, but they are equally applicable to any other such interface.

Write the code for a model once, run it on any supported simulator or hardware device *without modification*. This is the primary design goal for PyNN.

Support a high-level of abstraction. For example, it is often preferable to deal with a single object representing a population of neurons than to deal with all the individual neurons directly. Each single neuron can be accessed when necessary, but in many cases the population is the more useful abstraction. The advantages of this approach are that (i) it is easier to maintain a conceptual idea of the model, without being distracted by implementation details, and (ii) the internal implementation of an object can be optimized for speed, parallelization or memory requirements without changing the interface presented to the user.

Support any feature provided by at least two supported simulators. The aim is to strike a balance between supporting all features of all simulators (unfeasible) and supporting only the subset of features common to all simulators (overly restrictive).

Allow mixing of PyNN and native simulator code. PyNN should not limit the range of models that can be implemented. Following the two-simulator rule, above, there will be things that are possible in one simulator and not in any other. Although a model implementation consisting of 100% PyNN is the best scenario for running on multiple simulators, an implementation with 50% PyNN code will be easier to convert between simulators than one with no PyNN code.

Facilitate porting of models between simulators. PyNN changes the process of porting a model between simulators from all-or-nothing, in which the validity of the translated model cannot be tested until the entire translation is complete, to an incremental approach, in which the native code is gradually replaced by simulator-independent code. At each stage, the hybrid code remains runnable, and so it is straightforward to verify that the model behaviour has not been changed.

Minimize dependencies, to make installation as simple as possible and maximize flexibility. There are no visualization and few data analysis tools built-in to PyNN, which means the user can use any such tools they wish.

Present a consistent interface on output as well as on input. The formats used for simulation outputs are consistent across simulator back-ends, making it a stable base upon which to build more complex systems of simulation control, data-analysis and visualization.

Prioritize compatibility over optimizations, but allow compatibility-breaking optimizations to be selected by a deliberate choice of the user (e.g. the `compatible_output` flag of the various `print()` methods is `True` by default, but can be set to `False` to get potentially-faster writing of data to file).

API Versioning. The PyNN API will inevitably evolve over time, as more simulators are supported and to take account of the preferences of the community of users. To ensure backwards compatibility, the API should be versioned so that the user can indicate which version was used for a particular implementation. Note that the examples given in this paper use version 0.4 of the API.

Transparent parallelization. Code that runs on a single processor should run on multiple processors (using MPI) without changes.

Some of these goals are somewhat contradictory: for example, having a high level of abstraction and making porting easy.

Reconciling this particular pair of goals has led to the presence in PyNN of both a high-level, object-oriented interface and a low-level, procedural interface that is more similar to the interface of many existing simulators. These will be discussed further below.

USAGE EXAMPLES

Before describing in detail the concepts underlying the PyNN interface, we will work through some examples of how it is used in practice: first a simple example using the low-level, procedural interface and then a more complex example using the high-level, object-oriented interface.

For the simple example, we will build a network consisting of a single integrate-and-fire (IF) cell receiving spiking input from a Poisson process.

First, we choose which simulator to use by importing the relevant module from PyNN:

```
>>> from pyNN.neuron import *
```

If we wanted to use PCSIM, we would just import `pyNN.pcsim`, etc. Whichever simulator back-end we use, none of the code below would change.

Next we set global parameters of the simulator:

```
>>> setup(timestep=0.1, min_delay=2.0)
```

Now we create two cells: an IF neuron with synapses that respond to a spike with a step increase in synaptic conductance, which then decays exponentially, and a “spike source”, a simple cell that emits spikes at predetermined times but cannot receive input spikes.

```
>>> ifcell = create(IF_cond_exp,
...                 {'i_offset': 0.11,
...                 'tau_refrac': 3.0,
...                 'v_thresh': -51.0})
>>> times = map(float, range(5, 105, 10))
>>> source = create(SpikeSourceArray,
...                 {'spike_times': times})
```

Behind the scenes, the `create()` function translates the standard PyNN model name, `IF_cond_exp` in this case, into the model name used by the simulator, `Standard_IF` for NEURON, `iaf_cond_exp` for NEST, for example and also translates parameter names and units into simulator-specific names and units. To take one example, the `i_offset` parameter represents the amplitude of a constant current injected into the cell, and is given in nanoamps. The equivalent parameter of the NEST `iaf_cond_exp` model has the name `I_e` and units of picoamps, so PyNN both converts the name and multiplies the numerical value by 1000 when running with NEST. Standard cell models and automatic translation are discussed in more detail in the next section.

The `create()` function returns an ID object, which provides access to the parameters of the cell models, e.g.:

```
>>> ifcell.tau_refrac
3.0
>>> ifcell.tau_m = 12.5
>>> ifcell.get_parameters()
{'tau_refrac': 3.0, 'tau_m': 12.5,
 'e_rev_E': 0.0, 'i_offset': 0.11,
```

```
'cm': 1.0, 'e_rev_I': -70.0,
 'v_init': -65.0, 'v_thresh': -51.0,
 'tau_syn_E': 5.0, 'v_rest': -65.0,
 'tau_syn_I': 5.0, 'v_reset': -65.0}
```

Having created the cells, we connect them with the `connect()` function:

```
>>> connect(source, ifcell, weight=0.006,
...         synapse_type='excitatory', delay=2.0)
```

Now we tell the system what variable or variables to record, run the simulation and finish.

```
>>> record_v(ifcell, 'ifcell.dat')
>>> run(200.0)
>>> end()
```

The result of running the above model is shown in **Figure 1**, which also shows the degree of reproducibility obtainable between different simulators for such a simple network.

The low-level, procedural interface, using the `create()`, `connect()` and `record()` functions, is useful for simple models or when porting an existing model written in a different language that uses the create/connect idiom. For larger, more complex networks we have found that an object-oriented approach, with a higher-level of abstraction, is more effective, since it both clarifies the conceptual structure of the model, by hiding implementation details, and allows behind-the-scenes optimizations.

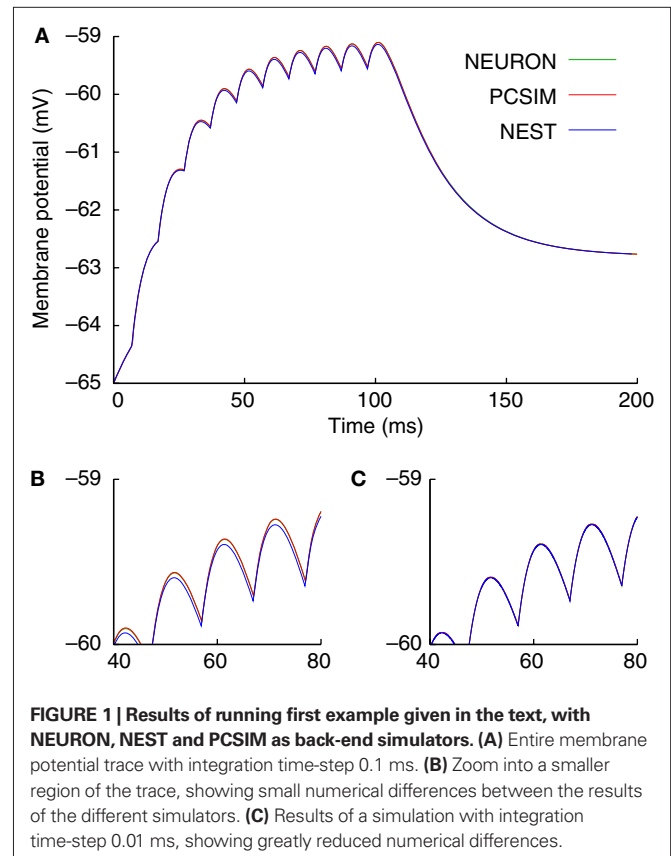


FIGURE 1 | Results of running first example given in the text, with NEURON, NEST and PCSIM as back-end simulators. (A) Entire membrane potential trace with integration time-step 0.1 ms. **(B)** Zoom into a smaller region of the trace, showing small numerical differences between the results of the different simulators. **(C)** Results of a simulation with integration time-step 0.01 ms, showing greatly reduced numerical differences.

To illustrate the high-level, object-oriented interface we turn now from the simple example of a few neurons to a more complex example: a network of several thousand excitatory and inhibitory neurons that displays self-sustained activity (based on the “CUBA” model of Vogels and Abbott (2005), and reproducing the benchmark model used in Brette et al. (2007)). This still is not a particularly complicated network, since it has only two cell types, no spatial structure and no heterogeneity of neuronal or connection properties, but in demonstrating how building such a network becomes trivial using PyNN we hope to convince the reader that building genuinely complex, structured and heterogeneous networks becomes manageable.

Again, we begin by choosing which simulator to use. We also import some classes from PyNN’s random module.

```
>>> from pyNN.nest2 import *
>>> from pyNN.random import (NumpyRNG,
...                           RandomDistribution)
```

We next specify the parameters of the neuron model (the same model and same parameters are used for both excitatory and inhibitory neurons).

```
>>> cell_params = {
...     'tau_m':      20.0, 'tau_syn_E':   5.0,
...     'cm':         0.2, 'tau_syn_I':  10.0,
...     'v_rest':    -49.0, 'v_reset':   -60.0,
...     'v_thresh': -50.0, 'tau_refrac':  5.0
... }
```

Parameters with dimensions of voltage are in millivolts, time in milliseconds and capacitance in nanofarads. The units convention is discussed further in the next section.

We now initialize the simulation, this time accepting the default values for the global parameters.

```
>>> setup()
```

Now, rather than creating each cell separately, we just create a Population object for each different type of cell:

```
>>> pE = Population(4000, IF_cond_exp,
...                 cell_params,
...                 label="Excitatory")
>>> pI = Population(1000, IF_cond_exp,
...                 cell_params,
...                 label="Inhibitory")
```

By default, all cells of a given Population are created with identical parameters, but these can be changed afterwards. Here we wish to randomize the value of the membrane potential at the start of the simulation to values between -50 and -70 mV.

```
>>> unif_distr = RandomDistribution('uniform',
...                                 [-50,-70])
>>> pE.randomInit(unif_distr)
>>> pI.randomInit(unif_distr)
```

`randomInit()` is a convenience method for randomizing the initial membrane potential. For the more general case of randomizing any cell parameter use `rset()`.

Just as individual neurons are encapsulated within Populations, connections between neurons are encapsulated within Projections. To create a Projection object, we need to specify how the neurons will be connected, either via an algorithm or via an explicit list. Different algorithms are encapsulated in different Connector classes, e.g. `FixedProbabilityConnector`, `AllToAllConnector`. An explicit list of connections can be provided via a `FromListConnector` or a `FromFileConnector`.

```
>>> FPC = FixedProbabilityConnector
>>> exc_conn = FPC(0.02, weights=0.004,
...                delays=0.1)
>>> inh_conn = FPC(0.02, weights=0.051,
...                delays=0.1)
```

Note that weights are in microsiemens and delays in milliseconds. Where the delay is not specified, the global minimum delay specified in the `setup()` function is used. Here we set all weights and delays of a Projection to the same value, but it is equally possible to pass the constructor a `RandomDistribution` object, as we did above for the initial membrane potential, or an explicit list of values.

To create a Projection, we need to specify the pre- and post-synaptic Populations, a Connector object, and a synapse type. The standard IF cells each have two synapse types, “excitatory” and “inhibitory”. User-defined models can use arbitrary names, e.g. “AMPA”, “NMDA”.

```
>>> e2e = Projection(pE, pE, exc_conn,
...                 target='excitatory')
>>> e2i = Projection(pE, pI, exc_conn,
...                 target='excitatory')
>>> i2e = Projection(pI, pE, inh_conn,
...                 target='inhibitory')
>>> i2i = Projection(pI, pI, inh_conn,
...                 target='inhibitory')
```

Having constructed the network, we now need to instrument it, using the `record()` (for recording spikes) and `record_v()` (membrane potential) methods of the Population objects. Here we choose to record spikes from 1000 of the excitatory neurons (chosen at random) and all of the inhibitory neurons, and to record the membrane potential of two specific excitatory neurons. We then run the simulation for 1000 ms.

```
>>> pE.record(1000)
>>> pI.record()
>>> pE.record_v([pE[0], pE[1]])
>>> run(1000.0)
```

After running the simulation, we can access the results or write them to file.

```
>>> pI.getSpikes()[:5]
array([[ 715. ,    1.5],
       [ 609. ,    1.6],
       [ 708. ,    1.7],
       [ 796. ,    1.7],
       [  34. ,    1.8]])
```

```
>>> pE.get_v()[:5]
array([[ 0. ,  0.1 , -55.073],
       [ 1. ,  0.1 , -50.163],
       [ 0. ,  0.2 , -55.098],
       [ 1. ,  0.2 , -50.212],
       [ 0. ,  0.3 , -55.122]])
>>> end()
```

The results of running simulations of the above network with two different simulator back-ends are shown in **Figure 2**.

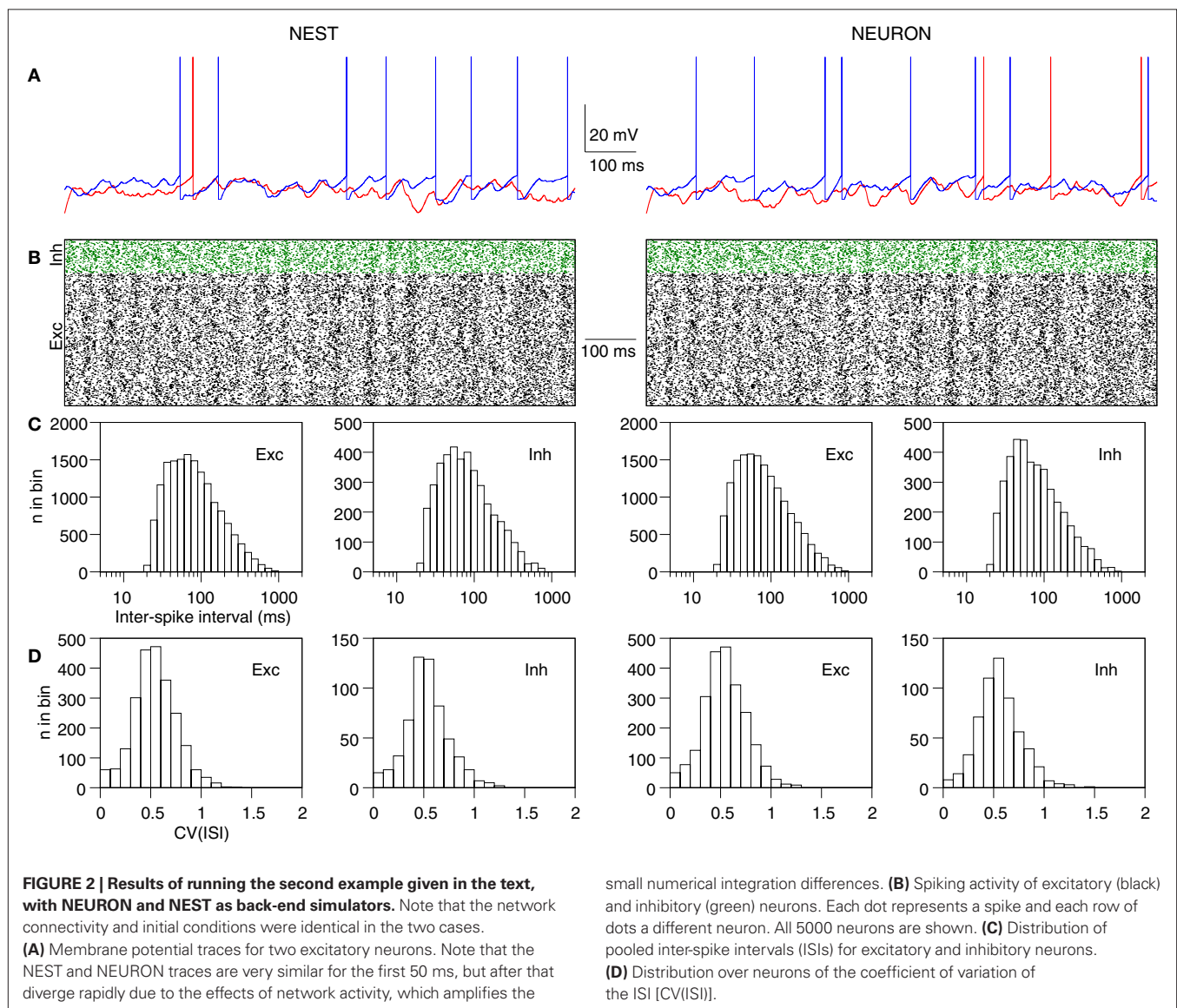
PRINCIPAL CONCEPTS

To achieve the goal of “write the code for a model once, run it on any supported simulator without modification” requires (i) a common interface, (ii) neuron and synapse models that are standardized across simulators, (iii) consistent handling of physical units, (iv) consistent handling of (pseudo-)random numbers. To achieve the twin goals of supporting a high-level of abstraction

and facilitating porting of models between simulators requires both an object-oriented and a procedural interface. The implementation of all these requirements is described in more depth in the following. We also illustrate the mixing of PyNN and native simulator code, and how PyNN can support features that are found in only a single simulator back-end, by describing support for multi-compartmental models.

STANDARD CELL MODELS

A fundamental concept in PyNN is the cell type – a given model of a neuron, representable by a set of equations, and comprising sub-threshold behaviour, spiking mechanism and post-synaptic response. The public interface of a cell type is mainly defined by its parameters. Different neurons of the same cell type may have very different behaviour if they have different values of the parameters. For example, the Izhikevich model (Izhikevich, 2003), can reproduce a wide range of spiking patterns, from fast-spiking through regular spiking to multiple types of bursting, depending on the



parameter values chosen. A cell type is therefore a model type rather than a biologically defined cell type (such as “Layer V pyramidal neuron”, for example).

When using a given simulator back-end, PyNN can work with any cell type that is supported by that simulator. In this case, the cell type is generally represented by a string, holding a model name that is meaningful for that simulator, e.g. “iaf_neuron” in NEST.

Of course, such a cell type will only work with one simulator. To create a model that will run on different simulators requires you to use one of PyNN’s built-in, standard cell models, each represented by a sub-class of the `StandardCell` class. The models provided by PyNN include various simple IF models, the Izhikevich-like adaptive exponential IF model (Brette and Gerstner, 2005), a single-compartment neuron with Hodgkin–Huxley sodium and potassium channels, and various models that emit spikes (e.g. according to a Poisson process) but cannot receive them.

The `StandardCell` class contains machinery for translating model names, parameter names and parameter units between PyNN standardized values and simulator-specific values. This is particularly useful when the underlying simulators use different unit systems or different parameterizations of the same set of equations, e.g. when one simulator expects the membrane time constant and another the membrane leak conductance. An example of the translations performed by PyNN is given in **Table 1**.

Currently, all the standard cell types are single-compartment or point neuron models, since PyNN currently supports only one simulator for multi-compartmental models (NEURON). Further details on using multi-compartmental models with PyNN’s NEURON back-end are given below. We plan in future to allow specifying multi-compartmental cell types using a NeuroML description (Crook et al., 2005).

UNITS

As is clear from the previous section, each simulator back-end has its own convention for which units to use for which physical quantities. The exception to this is Brian, which has a system for explicitly specifying units and for checking that equations are dimensionally consistent. In the future, we plan to adopt Brian’s system for PyNN, but for now we have chosen to use a convention, which is similar to

that of NEURON and NEST in that the units are those that tend to be used by experimental physiologists. An alternative would have been the convention used by PCSIM (and also by the GENESIS simulator) of using pure SI units with no prefixes. The advantage of the latter convention is that there is no need for checking equations for dimensional consistency. The disadvantage is that numerical values in such a system are often very large or very small, and hence the human intuition for reasonable and unreasonable parameter values is mostly lost.

Irrespective of the relative merits of different conventions, the most important thing is that PyNN now provides a single convention which is valid across simulators. In detail, the convention is as follows: voltage – mV, current – nA, conductance – μ S, time – ms, capacitance – nF.

STANDARD SYNAPSE MODELS

In PyNN, the shape and time-course of the elementary post-synaptic current or conductance change in response to a pre-synaptic spike are considered to be a part of the post-synaptic neuron model, while all other properties of a synaptic connection, notably its weight (the peak current or conductance of the synaptic response), delay (for point models, this implicitly includes axonal propagation, chemical transmission and dendritic propagation; more morphologically and/or biophysically detailed models may model explicitly some or all of these sources of delay), and short- and long-term plasticity, are considered to depend on both pre- and post-synaptic neurons, and so are encapsulated in the concept of “synapse type” that mirrors the “cell type” discussed above.

The default type of synaptic connection in PyNN is static, with fixed synaptic weights. To model dynamic synapses, for which the synaptic weight (and possibly other properties, such as rise-time) varies depending on the recent history of post- and/or pre-synaptic activity, we use the same idea as for neurons, of standardized, named models that have the same interface and behaviour across simulators, even if the underlying implementation may be very different.

Where the approach for dynamic synapses differs from that for neurons is that we attempt a greater degree of compositionality, i.e. we decompose models into a number of components, for

Table 1 | Comparison of parameter names and units for different implementations of a leaky integrate-and-fire model with a fixed firing threshold and current-based, alpha-function synapses. This model is called `IF_curr_alpha` in PyNN, `iaf_psc_alpha` in NEST, `LIFCurrAlphaNeuron` in PCSIM and `StandardIF` in NEURON (this is a model template distributed with PyNN and is not in the standard NEURON distribution). Manual conversion of names and units is straightforward but error-prone and time-consuming. PyNN takes care of such conversions transparently.

Parameter	PyNN		NEST		NEURON		PCSIM	
Resting membrane potential	<code>v_rest</code>	mV	<code>E_L</code>	mV	<code>v_rest</code>	mV	<code>Vresting</code>	V
Reset membrane potential	<code>v_reset</code>	mV	<code>V_reset</code>	mV	<code>v_reset</code>	mV	<code>Vreset</code>	V
Membrane capacitance	<code>cm</code>	nF	<code>C_m</code>	pF	<code>CM</code>	nF	<code>Cm</code>	F
Membrane time constant	<code>tau_m</code>	ms	<code>tau_m</code>	ms	<code>tau_m</code>	ms	<code>taum</code>	s
Refractory period	<code>tau_refrac</code>	ms	<code>t_ref</code>	ms	<code>t_refrac</code>	ms	<code>Trefrac</code>	s
Excitatory synaptic time constant	<code>tau_syn_E</code>	ms	<code>tau_syn_ex</code>	ms	<code>tau_e</code>	ms	<code>TauSynExc</code>	s
Inhibitory synaptic time constant	<code>tau_syn_I</code>	ms	<code>tau_syn_in</code>	ms	<code>tau_i</code>	ms	<code>TauSynInh</code>	s
Spike threshold	<code>v_thresh</code>	mV	<code>V_th</code>	mV	<code>v_thresh</code>	mV	<code>Vthresh</code>	V
Injected current amplitude	<code>i_offset</code>	nA	<code>I_e</code>	pA	<code>i_offset</code>	nA	<code>Iinject</code>	A

example for short-term and long-term dynamics, or for the timing-dependence and the weight-dependence of STDP rules, that can then be composed in different ways.

The advantage of this is that if we have n different models for component A and m models for component B, then we require only $n + m$ models rather than $n \times m$, which had advantages in terms of code-simplicity and in shorter model names. The disadvantage is that not all combinations may exist, if the underlying simulator implements composite models rather than using components itself: in this situation, PyNN checks whether a given composite model AB exists for a given simulator and raises an Exception if it does not. The composite approach may be extended to neuron models in future versions of the PyNN interface depending on the experience with composite synapse models.

Currently only a single model exists in PyNN for the short-term plasticity component, the Tsodyks–Markram model (Markram et al., 1998). For long-term plasticity there is a spike-timing-dependent plasticity STDP component, which itself is composed of separate timing-dependence and weight-dependence components.

LOW-LEVEL, PROCEDURAL INTERFACE

We refer to the procedural interface as “low-level” because it deals with a lower level of abstraction – individual neurons and individual synapses – than the object-oriented interface. The procedural interface consists of the functions `create()`, `connect()`, `set()`, `record()` (for recording spikes) and `record_v()` (for recording membrane potential). Each of these functions operates on, or returns, either individual cell ID objects or lists of such objects. As was described in the Usage Examples section, as well as being passed around as arguments, the ID object may be used for accessing/modifying the parameters of individual neurons, and takes care of parameter translation using the `StandardCell` mechanisms described above.

It is possible to some extent to mix the low-level and high-level interfaces. For example, it is possible to access individual neurons within a `Population` as ID objects and then use the `connect()` function to connect them, instead of using a `Projection` object.

Why have both a low-level and high-level interface? Having both is a potential source of confusion for users and is definitely a maintenance burden for developers. The main reason is to support the use of PyNN as a porting tool. The majority of neuronal network models using existing simulators use a procedural approach, and so conversion to PyNN is easier if PyNN supports the same approach. In addition, when developing a PyNN interface for a simulator, or for neuromorphic hardware, that deals primarily with individual cells and synaptic connections, it is easier to implement only the low-level interface, since the high-level interface can be built upon it.

HIGH-LEVEL, OBJECT-ORIENTED INTERFACE

Object-oriented programming has been used for many years in computer science as a method for reducing program complexity. As the ambition and scope of large-scale, biologically detailed neuronal network modelling increases, reducing program complexity will become more and more critical, as the limiting factor in computational neuroscience becomes the productivity of the programmer and not the capacity of the computer (Wilson, 2006). It is for this

reason that the preferred interface in PyNN for developing new models is an object-oriented one.

The object-oriented interface is built around three main classes:

Population – a group of cells all with the same cell type (model type). It is generally considered that the cells in a `Population` should all represent the same biological cell type, i.e. although parameter values may vary between cells in the group, all cells should have qualitatively the same firing response. This is not enforced, but is a good guideline to follow for producing understandable code. The `Population` class eliminates tedious iteration over lists of neurons and enables more efficient, array-based management of neuron properties.

Projection – the set of connections of a given synapse type between two `Populations`. Creating a `Projection` requires specifying the pre- and post-synaptic `Populations`, the synapse type, and the algorithm used to determine which neurons connect to which.

Connector – an encapsulation of the connection algorithm used in creating a `Projection`. Simple examples of such algorithms are “all-to-all”, “one-to-one” and “connect-each-pre-and-post-synaptic-cell-with-a-fixed-probability”. It is also possible to provide an explicit list of which cells are to be connected to which others. Each algorithm is defined within a subclass of the `Connector` class. PyNN contains a number of such classes, but it is fairly straightforward for a user to define their own algorithms.

In future development of PyNN, we plan to extend the interface to still higher-level abstractions, such as layers, cortical columns, brain areas and inter-areal projections. We also aim to use the high-level interface as a link between spiking network models and more abstract models that do not represent individual neurons, such as mean-field models.

RANDOM NUMBERS

The central nervous system contains many sources of noise, and activity patterns are often sufficiently complex, and possibly chaotic, to make a stochastic representation a reasonable model.

This can become a problem when comparing the behaviour of a given model run on different simulators, since random differences might obscure real inconsistencies between implementations of the model. Similarly, when performing distributed computations on parallel machines, the model behaviour should not depend on the number of processors used (Morrison et al., 2005), and random differences can conceal real differences between the parallel and serial implementations.

For these reasons, it is important to be able to use identical sequences of random numbers in different simulators, and to have the random number used at a particular point in the program execution be independent of which processor it is running on.

Another consideration is that simulations in most cases use only pseudo-random sequences, and low-quality random number generators (RNGs) may have correlations between different elements of the sequence that can significantly affect the qualitative behaviour of a network. Hence it is necessary to be able to test the simulation with different RNGs.

PyNN supports simulator-independent RNGs and use of different generators – currently any of the generators provided by

the numpy package or by the GNU Scientific Library (GSL) can be used.

This is done by wrapping the numpy and GSL RNGs in classes with a common interface. PyNN's `random` module contains the classes `NumpyRNG` and `GSLRNG`, which both have a single method, `next(n, distribution, parameters)`, which returns `n` random numbers from a distribution of type `distribution` with parameters `parameters`, e.g.

```
>>> from pyNN.random import NumpyRNG, GSLRNG
>>> rngN = NumpyRNG(seed=76847376)
>>> rngG = GSLRNG(seed=87548753)
>>> rngN.next()
0.91457981651574294
>>> rngG.next(5)
array([ 0.02518011, 0.79118205, 0.16679516,
...      0.1902914, 0.66204769])
>>> rngN.next(3, 'gamma', [2.0, 0.5])
array([ 0.48903019, 0.63129009, 0.70428452])
>>> rngG.next(distribution='uniform')
0.93618978746235371
```

Since all PyNN code that uses random numbers accesses the RNG classes only through this `next()` method, a user can substitute their own RNG simply by defining a wrapper class with such a method.

Since very often one wishes to use the same random distribution repeatedly, rather than changing distribution each time, the `random` module also provides the `RandomDistribution` class, which is initialized with the distribution name and parameters, and thereafter the `next()` method is simplified to take a single argument, the number of values to draw from the distribution, e.g.

```
>>> from pyNN.random import (NumpyRNG,
...                          RandomDistribution)
>>> rng = NumpyRNG(seed=8745753)
>>> gamma_distr = RandomDistribution('gamma',
...                                  [2.0, 0.5],
...                                  rng=rng)
>>> gamma_distr.next(3)
array([ 0.72682412, 0.82490159, 1.03882654])
```

Note that `NumpyRNG` and `GSLRNG` distributions may not have the same names, e.g. “normal” for `NumpyRNG` and “gaussian” for `GSLRNG`, and the arguments may also differ. One of our future plans is to extend the `random` module in order to harmonize names across RNGs.

MULTI-COMPARTMENTAL MODELS

PyNN currently supports only a single simulator, NEURON, that is suitable for many-compartment models. Given the principle of supporting simulator-independence only for features that are shared by at least two of the supported simulators, and given PyNN's focus on network modelling, PyNN does not provide an API for specifying simulator-independent multi-compartmental models. This is a possible future development – preliminary work has been done on a PyNN interface to the MOOSE simulator (Ray and Bhalla, 2008) – but a more likely path would be to make use

of the NeuroML standards for specifying multi-compartmental models. In this scenario, the filename of a NeuroML level 2 file, specifying a single cell type, would be passed as the `cellclass` argument to the PyNN `create()` function or `Population` constructor.

However, since native and PyNN code can be mixed, the `pyNN.neuron` module already supports simulations with multi-compartmental models. The pre-synaptic compartment whose voltage is watched to trigger synaptic transmission (e.g. axon terminal) can be specified using the `source` argument to the `Projection` constructor, and the post-synaptic mechanism specified with the `target` argument.

DEBUGGING

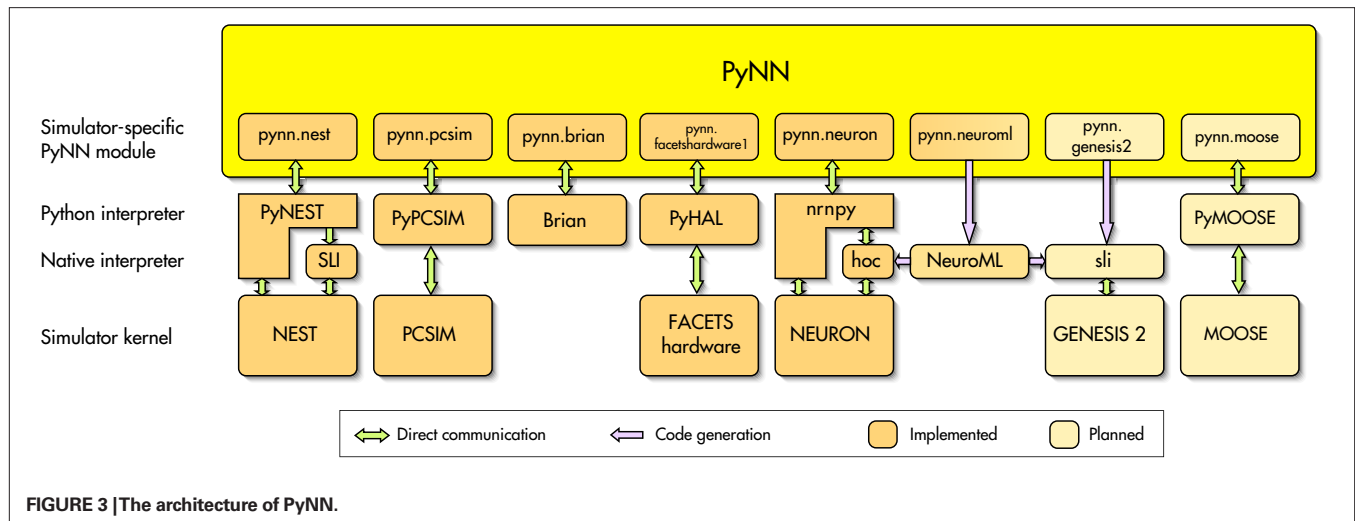
Should an error occur in a PyNN simulation, a good first step is to re-run it on another simulator back-end and so narrow down the source of the problem to one back-end in particular. Nevertheless, it has proven to be the case that the additional layers of abstraction provided by PyNN sometimes make it harder to track down sources of errors. To counterbalance this, PyNN traps errors coming from the simulator core and employs Python's introspection capabilities to provide additional information about the error context. For example, if an invalid parameter name is provided to a neuron model, the error message lists all the valid parameter names for that model. Furthermore, logging can be switched on via the `init_logging()` function in the `pyNN.utility` module, causing detailed information about what the system is doing to be written to file, a valuable resource for tracking down bugs.

IMPLEMENTATION

PyNN is both a definition of a common simulator interface and an implementation of this interface for each supported simulator. PyNN is implemented as a Python package containing a common module, which defines the API and contains functionality common to all simulator back-ends, a `random` module (described above), and a module for each simulator back-end, as shown in **Figure 3**. Each simulator module separately implements the API, although it can make use of much shared code in common. In most cases, the simulator modules have been implemented by, or in close collaboration with, the simulator developers.

PyNN currently fully supports the following simulators: NEURON (Carnevale and Hines, 2006; Hines and Carnevale, 1997; Hines et al., 2008), NEST (Eppler et al., 2008; Gewaltig and Diesmann, 2007), PCSIM (<http://www.lsm.tugraz.at/pcsim/>) and Brian (Goodman and Brette, 2008). Support for MOOSE (Ray and Bhalla, 2008) and for export in NeuroML format (Crook et al., 2005) is under development.

PyNN also supports the Heidelberg neuromorphic hardware system (Schemmel et al., 2007). This illustrates a major benefit of the existence of a common neuronal simulation interface: novel simulation or emulation systems do not need to develop their own programming interface, but can benefit from an existing one that guarantees interoperability with existing tools. Using PyNN as the interface to neuromorphic hardware systems provides the possibility of closing the gap between the two domains of numerical simulation and physical emulation, which have so far coexisted rather separately.



LIMITATIONS ON REPRODUCIBILITY

For a given model with a given parameter set run on a given version of a given simulator, it should be possible to exactly reproduce a simulation result, independent of computer architecture (except where this affects the precision of the floating-point representation) or operating system. For parallel systems, results should also be independent of how many threads or processes are used in the computation, although here exact quantitative reproduction is harder to achieve. Reproducibility across different versions of a given simulator is not essential provided the precise version used to generate a given result is specified, but it is of course highly desirable. When running a model on different simulators, exact reproduction is impossible to achieve, except in simple cases, due to round-off errors in floating point calculations. When validating a model implementation by running it on two or more simulators, therefore, what level of reproducibility is achievable, and how can we tell whether any differences are due to round-off error or to implementation errors?

To get a preliminary handle on this problem, we have compared the difference in model activity between two simulators to the difference due to two different initial conditions with the same simulator.

Our test case is the balanced random network, based on Vogels and Abbott (2005), whose implementation was shown above. The activity pattern of this network is very sensitive to initial conditions (chaotic or near-chaotic), and so we cannot use differences in the precise spike pattern to measure reproducibility: we are more interested in the statistical properties of the activity, and so we have chosen to take the distribution of inter-spike intervals (ISIs) of excitatory neurons (see **Figure 2C**) as a measure of network activity.

To measure the difference between the distributions from two different runs we use the Kolmogorov–Smirnov two-sample test. We ran the simulation ten times, each time with a different seed for the RNG used to generate the initial membrane potential distribution, with both NEURON and NEST back-ends. This gave values for the Kolmogorov–Smirnov D-statistic between 0.008 and 0.026 ($n \approx 19000$) with a mean of 0.015, with associated

p -values (probability that the two distributions are the same) between 6.3×10^{-5} and 0.68 with mean 0.15.

We then ran the simulation twenty times just on NEURON, each time with a different RNG seed, to give 10 pairs of distributions. In this case the D -values were in the range 0.007–0.026, mean 0.015, and the p -values in the range 2.8×10^{-5} to 0.77, mean 0.20.

In summary, the differences due to different simulators are in almost exactly the same range as those due to different initial conditions, suggesting that the differences between the simulators are indeed due to round-off errors and that there are not, therefore, any implementation errors in this case.

It is also interesting to note that in most cases the null hypothesis is supported, i.e. the distributions are the same, but that for some initial conditions there are highly significant differences between the ISI distributions. The ISI distribution may not therefore be the best measure for reproducibility in this case.

DISCUSSION

In this article we have presented PyNN, a Python-based common simulator interface, which allows simulator-independent model specification. PyNN is already in use in a number of research groups, and has been a key technology enabling improved communication between labs in a pan-European collaborative project with a major component of modelling and of neuromorphic hardware development (the FACETS project: <http://www.facets-project.org>).

By providing a standard simulation platform, PyNN also has the potential to act as the foundation for other, simulator agnostic but neuroscience-specific, tools such as analysis, visualization and data-management software.

PyNN is not the only project to address simulator-independent model specification and simulator interoperability (review in Cannon et al., 2007). neuroConstruct (Gleeson et al., 2007) is a tool to develop networks of morphologically-detailed neurons using a graphical user interface (GUI), that can generate code for both the NEURON and GENESIS simulators. A limitation with respect to PyNN is that since it uses code generation rather than a direct interface, neuroConstruct cannot receive information back from the simulator except by reading the data files it

generates. A second limitation is that features that are not available through the GUI cannot be incorporated in a model. The NeuroML standards (Crook et al., 2005, <http://www.neuroml.org>) are intended to provide an infrastructure for exchanging model specifications between groups in a simulator-independent way. Their scope includes much more detailed levels of modelling, e.g. membrane ion channels and detailed dendritic morphology, than are supported by PyNN. They have the advantage over PyNN of being language-independent, since specifications are written in XML, for which tools exist in all major programming languages. The major disadvantage of purely declarative specifications is lack of flexibility: if a concept or entity is not defined in the standard, it is not possible to specify models that use it, whereas with a procedural/imperative or mixed declarative-procedural specification such as is achievable with PyNN, arbitrary specifications are possible.

Although we emphasize here the differences between the GUI, pure-declarative, and programming-interface approaches to simulator-independent model specification, in fact they are highly complementary. Graphical interfaces are particularly good for beginners, for teaching, for giving high-level overviews of a system, and for integrating analysis and visualization tools. It would be very useful for neuroConstruct to be able to generate PyNN code, for example, in addition to code for NEURON and GENESIS. Declarative specifications reach the highest levels

of system-independence, for the range of concepts that are supported. They are also particularly suitable for transformation into human-readable formats and for automated GUI generation. As such, they seem to be best suited for domains in which the modelling approach is fairly stable, e.g. for describing neuron morphologies or non-stochastic ion channel models. In PyNN, we plan to support simulator-independent multi-compartmental models using NeuroML: in this scenario cell models would be specified in NeuroML while PyNN would be used for network specification and for simulation setup and control.

Our main priorities for future development of PyNN are to increase the number of supported simulators (simulator developers who are interested in PyNN support for their simulator are encouraged to contact us), improve the support for multi-compartmental modelling, and extend the interface towards higher-level abstractions, such as cortical columns and more abstract modelling approaches. PyNN is open source software (CeCILL licence, <http://www.cecill.info>) and has an open development model: anyone who wishes to contribute is welcome and invited to do so.

ACKNOWLEDGEMENTS

This work was supported by the European Union (FACETS project, FP6-2004-IST-FETPI-015879). Jens Kremkow is also supported by the German Federal Ministry of Education and Research (BMBF grant 01GQ0420 to BCCN, Freiburg).

REFERENCES

- Brette, R., and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.* 94, 3637–3642.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J., Diesmann, M., Morrison, A., Goodman P. H., Harris, F. Jr, Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A., El Boustani, S., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398.
- Cannon, R., Gewaltig, M., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., and De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.
- Carnevale, N. T., and Hines, M. L. (2006). The NEURON Book. Cambridge, University Press.
- Crook, S., Beeman, D., Gleeson, P., and Howell, F. (2005). XML for model specification in neuroscience: an introduction and workshop summary. *Brains Minds Media* 1, bmm228 (urn: nbn:de:0009-3-2282).
- Ekeberg, Ö., and Djurfeldt, M. (2008). MUSIC – multisimulation coordinator: request for comments. *Nat. Precedings* <http://dx.doi.org/10.1038/npre.2008.1830.1>.
- Eppler, J., Helias, M., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2, doi: 10.3389/neuron.11.012.2008.
- Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430.
- Gleeson, P., Steuber, V., and Silver, R. A. (2007). neuroConstruct: a tool for modeling networks of neurons in 3D space. *Neuron* 54, 219–235.
- Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2, doi: 10.3389/neuron.11.005.2008.
- Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209.
- Hines, M., Davison, A., and Muller, E. (2008). NEURON and Python. *Front. Neuroinform.* 2, doi: 10.3389/neuron.11.001.2009.
- Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Trans Neural Netw.* 14, 1569–1572.
- Markram, H., Wang, Y., and Tsodyks, M. (1998). Differential signaling via the same axon of neocortical pyramidal neurons. *Proc. Natl. Acad. Sci. USA* 95, 5323–5328.
- Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.
- Ray, S., and Bhalla, U. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2, doi: 10.3389/neuron.11.006.2008.
- Schemmel, J., Brüderle, D., Meier, K., and Ostendorf, B. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07). New Orleans, IEEE Press, pp. 3367–3370. doi: 10.1109/ISCAS.2007.378289.
- Vogels, T., and Abbott, L. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci.* 25, 10786–10795.
- Wilson, G. (2006). Where's the real bottleneck in scientific computing? *Am. Sci.* 94, 5–6.

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

Received: 21 September 2008; paper pending published: 21 October 2008; accepted: 22 December 2008; published online: 27 January 2009.

Citation: Davison AP, 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. Neuroinform.* (2009) 2:11. doi: 10.3389/neuro.11.011.2008
Copyright © 2009 Davison, Brüderle, Eppler, Kremkow, Muller, Pecevski, Perrinet and Yger. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Generating stimuli for neuroscience using PsychoPy

Jonathan W. Peirce*

Nottingham Visual Neuroscience, School of Psychology, University of Nottingham, Nottingham, UK

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Andrew D. Straw, California Institute of
Technology, USA

Peter Tass, Forschungszentrum Jülich,
Germany

***Correspondence:**

Jonathan Peirce, School of Psychology,
University of Nottingham, University
Park, Nottingham NG7 2RD, UK.
e-mail: jon@peirce.org.uk

PsychoPy is a software library written in Python, using OpenGL to generate very precise visual stimuli on standard personal computers. It is designed to allow the construction of as wide a variety of neuroscience experiments as possible, with the least effort. By writing scripts in standard Python syntax users can generate an enormous variety of visual and auditory stimuli and can interact with a wide range of external hardware (enabling its use in fMRI, EEG, MEG etc.). The structure of scripts is simple and intuitive. As a result, new experiments can be written very quickly, and trying to understand a previously written script is easy, even with minimal code comments. PsychoPy can also generate movies and image sequences to be used in demos or simulated neuroscience experiments. This paper describes the range of tools and stimuli that it provides and the environment in which experiments are conducted.

Keywords: Python, psychophysics, software, neuroscience, vision, fMRI, EEG, MEG

INTRODUCTION

The majority of experiments in modern neuroscience require the presentation of auditory or visual stimuli to subjects while a measure is taken of their ability to see, remember or interact with that stimulus, or of the brain activity that results from its presentation. As a result, neuroscience needs for tools that allow the accurate presentation of stimuli and collection of participant responses. Those tools should be as easy to use as possible to reduce the time spent constructing experiments, while being able to deliver as wide a variety of stimuli and experimental designs as possible to reduce the variety of software that a single scientist needs to learn to use. Additionally the ideal software package should be open-source, such that scientists can fully examine the code and know exactly what is being done “under the hood”, it should be platform independent and it should, of course, be free.

This article describes PsychoPy, an open-source software library that allows a very wide range of visual and auditory stimuli and a great variety of experimental designs to be generated within a very powerful script-driven framework based on Python. It is built entirely on open-source libraries and technologies, such that the user can, if they desire, examine all of the code that contributes to the stimuli they present. By leveraging the power of Python, and several existing cross-platform Python libraries, the software is fully platform independent and is being used in a number of labs worldwide on Windows, Mac OS X and Linux.

A previous publication (Peirce, 2007) describes the design philosophy and underlying mechanisms of PsychoPy and its relationship to other software packages, such as Vision Egg (Straw, 2008) and Psychophysics Toolbox (Brainard, 1997; Pelli, 1997). This paper focuses on its use, describing more of the variety of stimuli that the library can generate and present (images, dot arrays, text and movies), the environment in which experiments are developed and the latest developments and additions to the software.

MATERIALS AND METHODS

PsychoPy has been under active development since 2003 and, at time of writing, had reached version 0.95.2. The code is now

largely stable (it is largely backward-compatible between versions) and is sufficiently complete and bug-free that it is used as the standard means of conducting psychophysical and/or neuroimaging experiments in a number of labs worldwide. The software is still very much under development however; stimuli are still being added, code is still being optimised and the user interface is being refined constantly. There is a mailing list where users can report bugs, discuss improvements and get help in general use of the software.

PYTHON

One of the strengths of PsychoPy is its use of Python. The high-level functions and libraries available in Python make it an ideal language in which to develop such software. The platform independence that PsychoPy enjoys is based very much on the fact that it is based on pure Python code, using libraries such as *wxPython*, *pyglet* and *numpy* that have been written to be as platform independent as is technically possible. The fact that Python now has such a large user base means that there is a large community of excellent programmers developing libraries that PsychoPy can make use of. The fact that Python can be used in such a wide variety of ways (for example, in the author's own lab Python is used not only for stimulus presentation but also for data analysis, for the generation of publication-quality figures, for computational modelling and for various general purpose scripts to manipulate files) means that in many cases this is likely to be the only programming language that a scientist need learn, with the obvious benefits in time that result. By nature of its clean, readable, and powerful syntax combined with its free and open-source release model Python is clearly a very popular language that is continuously growing and developing further. Where Matlab has, in the past, benefited from its large user base and wide variety of applications to science, Python stands to benefit even more.

HARDWARE ACCELERATED GRAPHICS

One of the goals of PsychoPy was to generate stimuli in *real-time*, that is to update the character of a stimulus on a frame-by-frame basis as needed without losing temporal precision. For static stimuli this is an

unnecessary benefit, but for moving stimuli, where the alternative is to pre-compute a movie sequence it makes for much cleaner experimental code, with fewer delays (some experiments would previously require several seconds or even minutes before running where they computed the stimulus movies). The possibility of real-time stimulus manipulations also allows experiments to alter based on input from the participant such that, for example, a stimulus might be moved fluidly under mouse (or even eye-movement) control, or the next stimulus can be generated based on the previous response.

In order to achieve good temporal precision, while updating stimuli in real-time from an interpreted language like Python or Matlab, it has been essential to make good use of the hardware accelerated graphics capabilities of modern computers. Most modern machines have very powerful graphics processing units that can perform a lot of the calculations necessary to present stimuli at a precise point in space and time and to update that stimulus frequently. The OpenGL specification determines, fairly precisely, what a graphics card should do given various commands, such that platform independence is largely maintained (there are certain aspects, such as the synchronisation of drawing with the screen vertical refresh that are graphics card and/or platform dependent). PsychoPy 0.95 is fully compatible with the OpenGL 1.5 specification but makes use of further facilities that were added to OpenGL 2.0 on graphics cards and drivers where these are available. Nearly all modern graphics cards are capable of using OpenGL (although they may need updated drivers) and perfectly adequate cards from nVidia or ATI, that support the OpenGL 2.0 extensions, can be currently purchased and added to a desktop computer of any platform for roughly £30.

PLATFORM INDEPENDENCE

Platform independence is a particular goal of PsychoPy. Computer technologies change rapidly and the relative advantages of different platforms can vary equally quickly. Scientists should not need to learn a whole new set of tools just because they have decided to switch their main computer platform, and should be able to share code and experiments with colleagues using other platforms. Perfect independence is never possible because of hardware differences between computers. Some such differences are obvious; for example, Apple Macs have not supported parallel ports directly for several years so scripts using parallel port communication cannot work on those platforms. Other differences are subtle and unnoticed by most users. An example of this is that the OpenGL specification allows for the frame not to be cleared after a swap of the “front” and “back” buffers during a screen refresh, but does not specify whether the new back buffer is maintained from the previous back buffer (most useful for the continuity of drawing frames) or retrieved from the previous front buffer (as implied by the term “swapping” buffers). As a result, the behaviour is free to, and does, vary between manufacturers.

In the vast majority of cases, however, thanks to the hard work of the developers of libraries such as *pyglet*, *numpy* and *wxPython*, a PsychoPy script will run identically on all platforms.

RESULTS

INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

PsychoPy was developed as a Python package that could be imported from scripts needing to present stimuli. For new users of Python

that has certain disadvantages; users need to install Python and other dependent libraries separately, they need some form of text editor to write the scripts and they need to know where to find the text, including error messages, that scripts might output. Although none of these are difficult (and may seem obvious to an experienced programmer or user of command-line operating systems), they were impediments to new users, particularly from Windows and “traditional” Mac platforms. PsychoPy now comes with a built-in code editor (PsychoPyIDE), complete with code auto-completion, code folding and help tips. Scripts can be run directly from the editor and code output is directed to another window in the application (see **Figure 1**). When this output includes error messages these show up as URL-style links that take the user directly to the line on which the error occurred.

On Windows, installation is very straightforward using simple double-clickable installers. On Intel-based Apple Macintosh computers running OS X an application bundle is provided that contains its own copy of Python and all the dependent libraries. This has a number of advantages. The first is that it installs simply as a single application that can be dragged into the Applications folder (or other location) and can be removed equally easily by simply sending to the trash. As well as being easy to install by this method, distributing PsychoPy with its own copy of Python has two major advantages: PsychoPy’s developers know what libraries have been installed and that they are compatible and the user knows that it won’t interfere with any existing Python installation that they have (such as previous installs, or the Apple system Python). For more experienced Python users, who may wish to install to their own customised set of libraries, the standard Python-style methods of installing from source distributions are also available.

On Linux the dependencies can be installed simply from simple `apt-get` commands and PsychoPy is then easily installed from its source distribution.

MODULE STRUCTURE

As with most Python packages, PsychoPy contains a number of sub-modules, which can be imported relatively independently (some depend on each other) depending on the task at hand. This is useful in keeping related functions and classes together in meaningful units. For instance, the following will import modules useful in presenting visual and auditory stimuli and collecting responses (events) from the subject:

```
from psychopy import visual, core, event
```

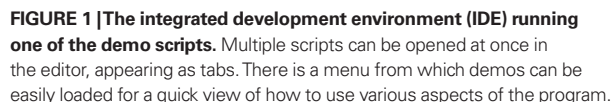
The main modules that can be imported from PsychoPy, and the main libraries that they depend upon are shown in **Figure 2**.

PRESENTING STIMULI

A subset of the available visual stimuli is shown as a screenshot in **Figure 3**.

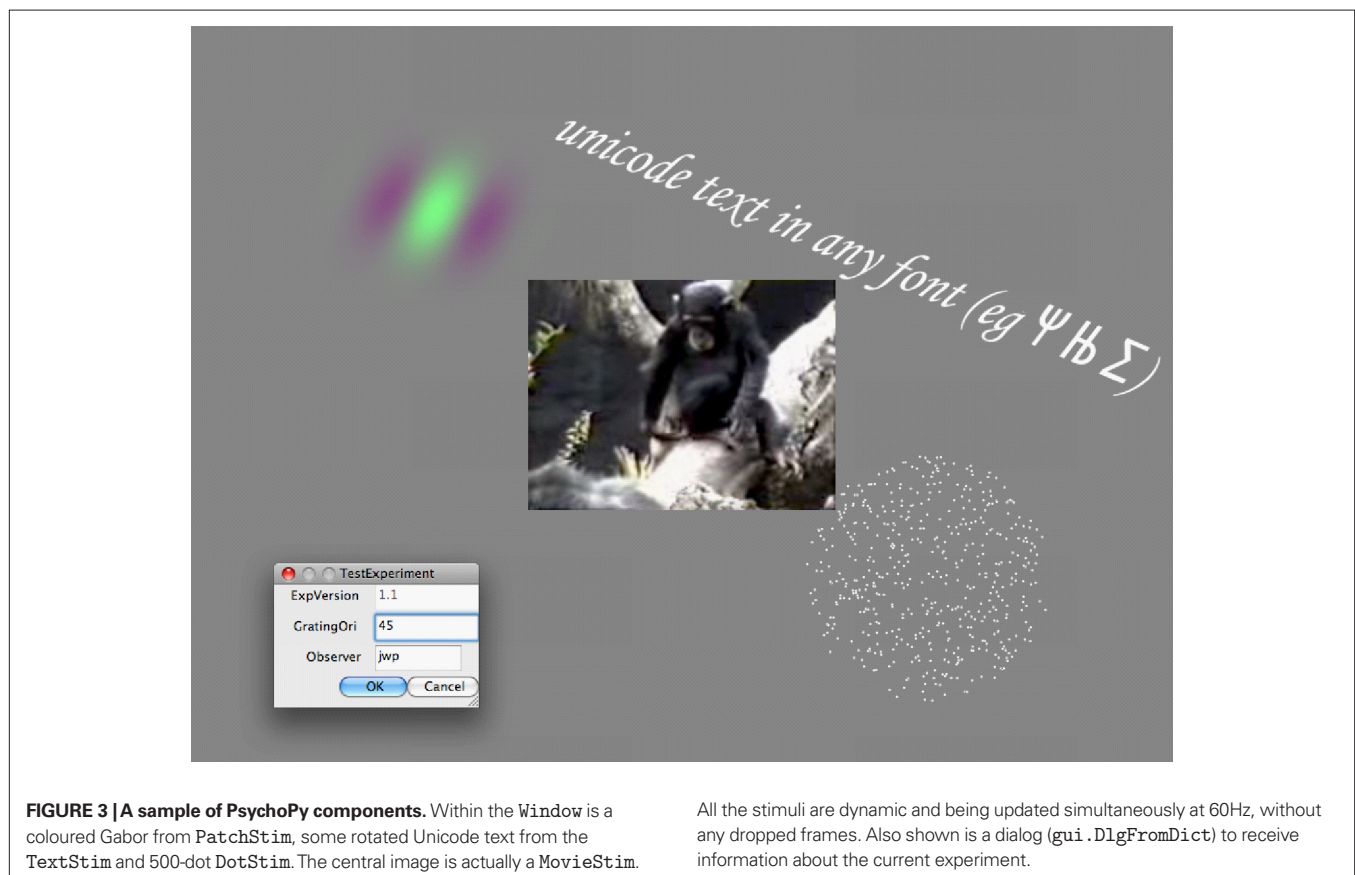
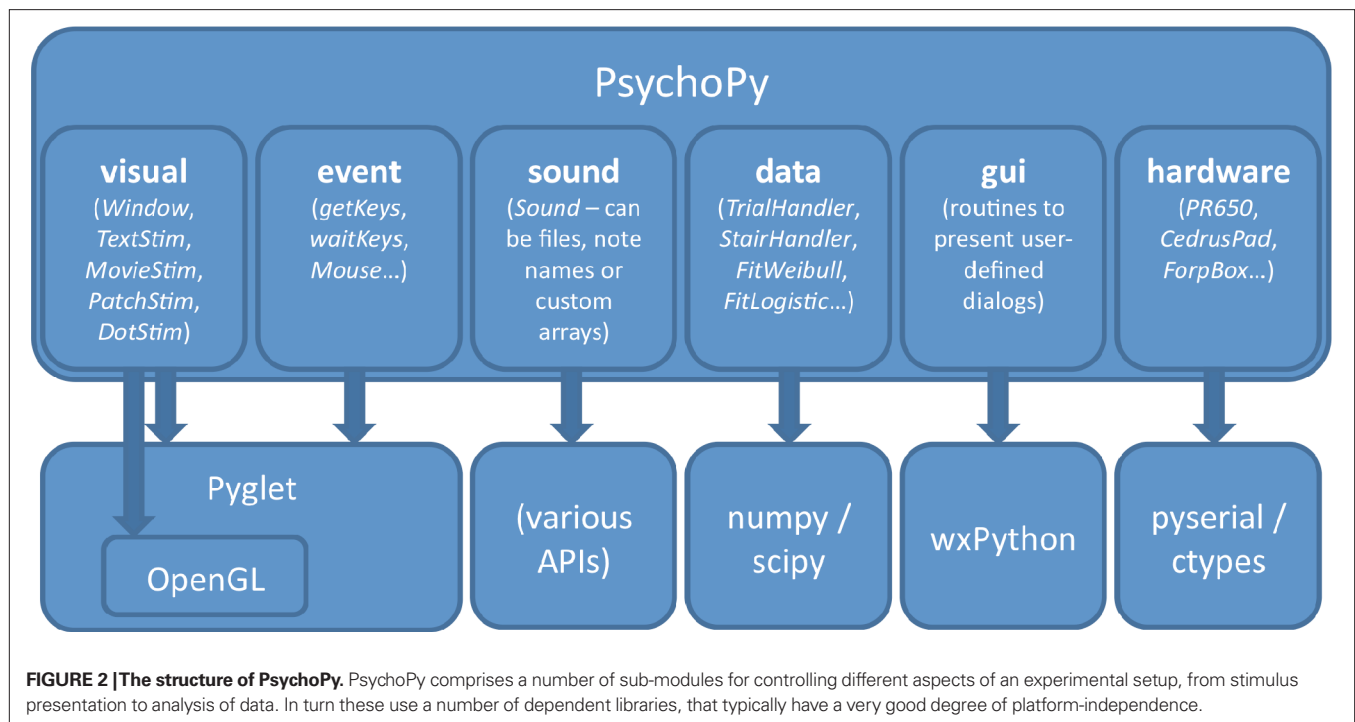
Windows

Most experiments begin with creating a window into which visual stimuli or instructions can be presented. In PsychoPy this can be achieved in a full screen mode or in a normal window, with the mouse either shown or hidden. Furthermore, multiple windows can be created at one time and these may be presented on any physical



60fps, [Esc] to quit

The most widely-used stimulus in PsychoPy is the PatchStim, used to control a visual patch on the screen. Patches can contain any bitmap-style data, including periodic textures (such as sinusoidal gratings or repetitive lines) or photographic images. These



also support alpha masks, which define the transparency of the stimulus across the patch and can therefore determine the shape, or “envelope” of the stimulus. These stimuli can be manipulated

in real-time in a wide variety of useful ways; the bitmaps can be rotated, have their phase shifted, change the number of cycles in either dimension etc.

As a result, PatchStim stimuli can be used to present a wide variety of image-based objects, either those used typically in visual psychophysics (gratings, Gabors etc...) or those in higher-level psychology and cognitive neuroscience studies (such as photographic images) or to create simple geometric shapes such as fixation points and arrows.

TextStim

Another common experimental requirement is the presentation of text to subjects, either as instructions or as actual stimuli. PsychoPy has a stimulus that provides simple access to clear, anti-aliased text in any true-type font available on the host system (obviously more can be installed). These stimuli are fully compatible with Unicode, so that symbols and non-English characters can be included. Text objects can be coloured in any of the colour spaces and referred to by any coordinate system for which the window has been calibrated (see Windows). They can also be rotated arbitrarily and in real-time.

Sound

PsychoPy also provides direct and simple access to methods for presenting auditory stimuli. Sound objects can be created from files (wav, mpg), from pure tones (the user specifies the duration and either frequency or the name of the note and octave on a standard scale) or can be generated from arbitrary waveforms using the standard *numpy* library in Python. Sound objects can be played in full stereo in asynchronous threads, so as to overlap as necessary with each other and with visual presentations.

The ability to play arbitrary stereo waveforms as sounds makes PsychoPy perfectly capable of running full auditory psychophysical experiments, but the sounds can equally easily be used just to present feedback tones to subjects carrying out basic experimental tasks.

DotStim

A common stimulus in visual neuroscience is the random dot pattern (e.g. see Scase et al., 1996), also known as the Random Dot Kinematogram and this is provided in PsychoPy by the DotStim object. This allows either an array of dots, or an array of other PsychoPy stimuli (e.g. PatchStims) to be drawn as a field. The position of the dot elements can then be automatically updated by a variety of rules, for instance where a number of target dots move in a given direction while the remaining (distracter) dots move in random directions. This type of stimulus makes heavy use of OpenGL optimisations and allows a large number of dot elements (several hundred) to be drawn and updated in realtime without dropping frames.

MovieStim

PsychoPy can present movies in a variety of formats including mpeg, DivX, avi and Quicktime, allowing studies using natural scene stimuli or biological motion displays. As with most other stimulus types, these can also be transformed in a variety of ways (e.g. rotated, flipped, stretched) in real-time.

COLLECTING RESPONSES

Most experiments also need to receive and store information about responses from subjects. For PsychoPy, this can be achieved via a

number of simple means; keyboards, mice, joysticks and specialised hardware such as button boxes. The simplest possible input method is to examine recent events from the keyboard using the `event.getKeys()` and `event.waitKeys()` functions. These allow the user to see what keys have been pressed since the last call or to wait until one has been pressed (and may be restricted to a small number of allowed keys). The `event.Mouse` object allows PsychoPy users to determine where the mouse is at any given moment or whether a mouse button has been pressed with simple methods such as `getPos()`, `getWheelRel()` (to retrieve the relative movement of the mouse scroll wheel) and `getPressed()`. **Code Snippet 1** demonstrates how to use these mouse and keyboard facilities to control a drifting Gabor patch (a sinusoidal grating in a Gaussian-shaped envelope) in real-time within a PsychoPy window.

INTEGRATING WITH HARDWARE

Many input/output devices can be accessed directly from within PsychoPy by emulating keyboards or rodents. For example, the fORP MR-compatible button boxes (Current Designs, Philadelphia, USA) are capable of outputting signals that emulate key presses on a standard keyboard (e.g. keys 1–4 can represent buttons with key 5 representing a trigger pulse from an MRI scanner). Many touch-sensitive screens simply emulate a mouse press at the location where the screen was touched, and can therefore be used within PsychoPy as if a mouse event had occurred. These often provide the simplest methods of input to an experimental program. On other occasions these are unsuitable, either because the nature of the information being transmitted does not easily emulate such devices or because those devices are already in use. For example, what happens if you need button-box input as well as, and separate from, keyboard input?

PsychoPy also provides simple and complete access to input and output via serial and parallel ports (or via USB serial/parallel emulators, on systems where direct hardware ports are unavailable). An example of the use of serial and parallel port communications is shown in **Code Snippet 2**. Typically the parallel port is used to control and receive simple triggers in switching a current from high (+5 V) to low (0 V) or vice-versa and particularly useful in informing other hardware (such as an Electroencephalography device) of the precise onset of an event in PsychoPy. Serial ports can be used to pass more complex information, such as text characters or data in bytes at a fixed rate and are still heavily used by a large number of scientific devices because of their relative simplicity. For example, PsychoPy uses the serial port protocol to communicate with a PR650 spectrophotometer (Photo Research Inc, Chatsworth, USA) sending commands to begin measurements and receiving data back from the device such as the full power spectrum of the currently presented screen.

Some devices may also make use of calls from binary-compiled dynamically-loaded libraries (dlls on the Windows platform, dylibs on OS X). In particular most devices connecting via USB, Firewire or PCI cards will come with drivers that fall into this category. Python provides a module called `ctypes` (as of version 2.5), which allows seamless calls to any such drivers and dynamic libraries directly from Python itself.

Through one of these methods, any hardware that can communicate with your computer, can also communicate with Python and PsychoPy.


```

from psychopy import visual, core, event # import the PsychoPy libraries

#create a window to draw in
myWin = visual.Window((600.0,600.0), allowGUI=True)

#initialise some stimuli
fixSpot = visual.PatchStim(myWin,
    tex="none", mask="gauss", #no texture and a Gaussian shape
    pos=(0,0), size=(0.05,0.05), #size and location as fraction of window
    rgb=[-1.0,-1.0,-1.0]) #the colour of the fixation (black)
grating = visual.PatchStim(myWin,pos=(0.5,0),
    tex="sin",mask="gauss", #grating texture and a Gaussian shape
    rgb=[1.0,0.5,-1.0], #
    size=(1.0,1.0), sf=(3,0)) #set the size and the grating cycles
myMouse = event.Mouse(win=myWin) #a mouse object related to our window
message = visual.TextStim(myWin,pos=(-0.95,-0.9), #a TextStim to provide info
    alignHoriz='left', height=0.08,#specifying the size of the font
    text='left-drag=SF, right-drag=pos, scroll=ori') #and the actual text

for frameN in range(2000): #for 2000 frames
    #handle key presses each frame
    for key in event.getKeys(): #returns keys pressed this frame
        if key in ['escape','q']:
            core.quit()

    #get mouse events
    mouse_dX,mouse_dY = myMouse.getRel() #get position relative to previous
    mouse1, mouse2, mouse3 = myMouse.getPressed()
    #based on the mouse button and change in position, change the stimulus
    if (mouse1): #if button 1 is down (ie left-click)
        grating.setSF(mouse_dX/200.0, '+')
    elif (mouse3): #else if button 3 is down (ie right-click)
        grating.setPos([mouse_dX/400.0, -mouse_dY/400.0], '+')

    #Handle the mouse wheel(s)
    wheel_dX, wheel_dY = myMouse.getWheelRel()
    #change the grating orientation according to the wheel
    grating.setOri(wheel_dY*5, '+') #2 clicks will give 10deg rotation
    event.clearEvents() #get rid of other, unprocessed events

    #draw our stimuli (every frame)
    fixSpot.draw() #visual stimuli have a simple 'draw' function
    grating.setPhase(0.05, '+') #advance grating by 0.05 cycles per frame
    grating.draw()
    message.draw()
    myWin.flip() #update the window

core.quit() #when we're done (Python loops finish when code indentation ends)

```

CODE SNIPPET 1 | Presenting stimuli under real-time control. This demo script controls a drifting grating in real-time according to input from the mouse. It demonstrates the use of the Window, PatchStim, TextStim and Mouse objects and how to get keyboard input from the participant. These objects have associated methods that allow them to have their attributes changed.

TIMING

Timing is a critical issue for many experiments in neuroscience and psychology. Many studies require a temporal precision to within a few milliseconds, or even in the sub-millisecond range. PsychoPy provides various methods to achieve very precise timing of events and to synchronise with other devices. This is achieved by means

of synchronising drawing to the VBL of the monitor, by the use of very precise clocks on the host CPU and by access to rapid communication ports such as the serial and parallel ports.

PsychoPy (like most such software) uses a double-buffered method of rendering, whereby stimuli are initially drawn into a back buffer, a virtual screen in the memory of the graphics card.

```

from psychopy import core, parallel, serial

#initialise ports
serialPort = serial.Serial("COM1", baudrate=115200, bytesize=8, parity='N',
    stopbits=1, timeout=0.0001)
parallel.setPortAddress(0x378) #need to know your parallel port address

#set pin 2 to high and send a command to Cedrus RB730
parallel.setPin(pinNumber=2, state=1) #set pin 2 to high
serialPort.writelines("_d1") #send a command to the serial port

core.wait(0.5)

#set pin 2 to low and read response from Cedrus RB730
parallel.setPin(pinNumber=2, state=0) #set pin 2 to low
nCharsToGet = serialPort.inWaiting()
message = serialPort.read(nCharsToGet)#read the current characters
print message

```

CODE SNIPPET 2 | The use of serial and parallel ports to control hardware and synchronisation. The demo sends a command to the serial port (in this case the command would request information from a Cedrus box about its type and version) and reads the response after a 0.5-s pause. During this period pin 2 on the parallel port is set to high.

At the point when the VBL occurs (signifying the end of one frame and the beginning of the next) the contents of this back buffer are flipped with the actual screen buffer. When the command `Window.flip()` is sent, PsychoPy will halt all processing (or processing just in this thread if multiple threads are being used) until the graphics card signals that a frame flip has occurred. Since these frame flips occur at a very precise interval they can be used as a very precise timing mechanism and by executing a command immediately after the flip one can be certain that it is time-locked to the presentation of that stimulus frame.

The precision of this system can break down when frames are dropped – if too many commands are attempted (e.g. too many stimuli are drawn) between frames then the VBL may occur before the request to flip the buffers occurred, in which case the frame will remain unchanged for twice the normal period. In some cases this will be unimportant (e.g. if it occurs during an inter-trial interval it is likely to be irrelevant). At other times it could cause a slip in the timing of the study, causing a stimulus to be presented longer than intended. For dynamic stimuli it may change the perceptual appearance of the stimulus, causing a smoothly-moving stimulus to stutter in its motion, for instance.

PsychoPy alleviates this hazard by using the graphics card processor as much as possible for calculations involved in drawing, such as the transformations needed in rotating, scaling and blending multiple stimuli. For simple experiments, using just a few standard stimuli, almost any modern computer is likely to have the processing power to draw multiple stimuli without dropping frames. For studies needing large numbers of stimuli updating every frame, the need for faster computers and graphics cards exerts itself. In particular, the use of computers with “onboard” graphics processors (such as the GMA 950 graphics processor that comes on many Intel processors) is not recommended – even the cheapest nVidia and ATI graphics cards will easily outperform these chips. Also, as complexity increases, so does the need to write more efficient experiment scripts. Often this is simply a case of finding ways to reduce the number of commands

executed, for example by manipulating large lists of numbers as *numpy* arrays rather than iterating operations in for-loops. Sometimes it may mean having a better understanding of the speed of operations that will result from the command – giving a `PatchStim` a new texture is time-consuming if the texture is large, whereas changing its orientation or colour has a relatively small overhead, so preloading textures into stimuli is a good idea whenever possible.

Although PsychoPy and Python are potentially (subject to a well-written script) very precise in their reporting and generation of stimuli, there are a number of hardware limitations in most experimental setups that limit the absolute temporal accuracy of studies. The most obvious is the temporal resolution of the presentation device (typically a monitor or projector) but many experimenters are also unaware of the inherent latencies of other hardware components in their system. In general, these limit the accuracy rather than precision of the studies, since the latencies are relatively constant, but are nevertheless worthy of exploration.

Frame rates and monitor technology

The most fundamental limitation to the temporal precision of most studies is the frame rate of the monitor, and this varies dependent on the particular monitor technology. Cathode ray tube screens typically operate at refresh rates ranging 60–200 Hz, dependent on the monitor and the resolution of the display. For the majority of the frame period (say 12 ms for an 85-Hz refresh rate) pixels are being drawn sequentially in lines progressing from the top of the screen to the bottom. When the beam illuminating the pixels reaches the bottom of the screen there is a pause of around 1.5 ms while it returns to top, ready to draw the next frame (this is the VBL period). The obvious result is that visual stimuli cannot be changed at a rate greater than the frame rate – when a stimulus is scheduled for drawing, for example following some user response, it cannot be drawn until the next refresh of the screen. A less obvious result is that stimuli are drawn as much as 10 ms apart, even on the same frame, depending on their screen position.

LCD panel displays (either projectors or monitors) are typically limited to a screen refresh rate of 60 Hz and therefore share the problem of having a limited rate at which stimuli can be changed. They do not, however, draw the lines to the screen sequentially and so do not suffer from the problem that parts of the screen are drawn before others. On the other hand, the response time of these displays is considerably slower – an LCD switching from black to white changes rather gradually, over a period of around 20 ms. In cases where the screen is changed very rapidly this can have profound effects. For instance, if a stimulus is intended to flash black and white on alternating screens, it is unlikely on these monitors to reach full black and full white and a lower contrast stimulus will result.

The use of USB devices

Commonly the need for timing accuracy comes from the need to know how long a participant took to respond to the presentation of a stimulus, where their response is measured by pressing a button on a keyboard or response box. Unfortunately these devices are often USB-based and this introduces another temporal lag of, typically, 10–20 ms. Again, for a given device and computer system it is likely to be relatively constant, affecting the absolute accuracy of the response time measurement more than the precision.

DISCUSSION

PsychoPy is already a very useful tool for running experiments that require visual and auditory stimuli in a wide variety of environments. It is platform-independent, entirely free, simple to use and extremely versatile. It is also continuously improving in the variety of stimuli it can present, the accuracy and speed with which it can present them and in its ease of installation and use.

REFERENCES

- Brainard, D. H. (1997). The psychophysics toolbox. *Spat. Vis.* 10, 433–436.
- Derrington, A. M., Krauskopf, J., and Lennie, P. (1984). Chromatic mechanisms in lateral geniculate nucleus of macaque. *J. Physiol.* 357, 241–265.
- MacLeod, D. I., and Boynton, R. M. (1979). Chromaticity diagram showing cone excitation by stimuli of equal luminance. *J. Opt. Soc. Am.* 69, 1183–1186.
- Peirce, J. W. (2007). PsychoPy—Psychophysics software in Python. *J. Neurosci. Methods* 162, 8–13.
- Pelli, D. G. (1997). The VideoToolbox software for visual psychophysics: transforming numbers into movies. *Spat. Vis.* 10, 437–442.
- Scase, M. O., Braddick, O. J., and Raymond, J. E. (1996). What is noise for the motion system? *Vision Res.* 36, 2579–2586.
- Straw, A. D. (2008). Vision egg: an open-source library for realtime visual stimulus generation. *Front. Neuroinformatics* 2, 4.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.
- Received: 09 September 2008; paper pending published: 27 October 2008; accepted: 19 December 2008; published online: 15 January 2009.
- Citation: Peirce JW (2009) Generating stimuli for neuroscience using PsychoPy. *Front. Neuroinform.* (2009) 2:10. doi: 10.3389/neuro.11.010.2008
- Copyright © 2009 Peirce. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.

As an open-source project its continued development benefits from its increasing user base, and that of the wider Python community. Python is also a language suitable for a wide variety of other tasks, including complex data analysis and computational modelling. Data can be shared easily between PsychoPy and other Python-based packages (e.g. using stored *numpy* arrays), or can be exported to other programs using comma-separated or tab-delimited text files.

The variety of stimuli that PsychoPy can produce and its temporal precision in generating these in real-time make it an ideal environment for many neuroscience endeavours. It was originally designed for psychophysical studies in vision, but is also an ideal package for presenting stimuli in more traditional cognitive psychology experiments, including the ability to interface with touchscreens and, by virtue of its simple interface to parallel and serial ports, it is already being used by a number of labs for fMRI, MEG, EEG. PsychoPy is relatively young. Although it has been used as standard in the author's lab since 2004 it has been used in other labs only since 2006. The community around it is growing however; at the time of writing the package had been downloaded 5000 times and has an active mailing list with 50 members.

A great deal more information is available from the project's website (<http://www.psychopy.org>), including tutorials, demonstration code and reference material for the writing of scripts.

ACKNOWLEDGEMENTS

PsychoPy has been developed with support from a BBSRC project grant (BB/C50289X/1), a Wellcome Trust Grant and seed funding grants from The Royal Society and the University of Nottingham. Many thanks to all those that have provided constructive criticism, and destructive testing, especially Dr. B.S. Webb.



Modular toolkit for Data Processing (MDP): a Python data processing framework

Tiziano Zito^{1*}, Niko Wilbert^{1,2}, Laurenz Wiskott^{1,2} and Pietro Berkes³

¹ Bernstein Center for Computational Neuroscience, Berlin, Germany

² Institute for Theoretical Biology, Humboldt-Universität zu Berlin, Germany

³ Volen Center for Complex Systems, Brandeis University, Waltham, MA, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Nicholas T. Carnevale, Yale University
School of Medicine, USA
Thomas Natschläger, Software
Competence Center Hagenberg
GmbH, Austria

*Correspondence:

Tiziano Zito, Bernstein Center for
Computational Neuroscience,
Philippstraße 13, House 6, Humboldt-
Universität zu Berlin, 10115 Berlin,
Germany.

e-mail: tiziano.zito@bccn-berlin.de

Modular toolkit for Data Processing (MDP) is a data processing framework written in Python. From the user's perspective, MDP is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing sequences and more complex feed-forward network architectures. Computations are performed efficiently in terms of speed and memory requirements. From the scientific developer's perspective, MDP is a modular framework, which can easily be expanded. The implementation of new algorithms is easy and intuitive. The new implemented units are then automatically integrated with the rest of the library. MDP has been written in the context of theoretical research in neuroscience, but it has been designed to be helpful in any context where trainable data processing algorithms are used. Its simplicity on the user's side, the variety of readily available algorithms, and the reusability of the implemented units make it also a useful educational tool.

Keywords: Python, Modular toolkit for Data Processing, computational neuroscience, machine learning

INTRODUCTION

The use of the Python programming language in computational neuroscience has been growing steadily during the past few years. The maturation of two important open source projects, the scientific libraries NumPy¹ and SciPy², gives access to a large collection of scientific functions that rivals in size and speed well known commercial alternatives like The MathWorks™ Matlab^{®3}. Furthermore, the flexible and dynamic nature of Python offers the scientific programmer the opportunity to quickly develop efficient and structured software while maximizing prototyping and reusability capabilities. The Modular toolkit for Data Processing (MDP) package⁴ contributes to this growing community a library of widely used data processing algorithms, and the possibility to combine them according to a pipeline analogy to build more complex data processing software.

MDP has been designed to be used as-is and as a framework for scientific data processing development. From the user's perspective, MDP consists of a collection of supervised and unsupervised learning algorithms, and other data processing units (*nodes*) that can be combined into data processing sequences (*flows*) and more complex feedforward network architectures. Given a set of input data, MDP takes care of successively training or executing all nodes in the network. This allows the user to specify complex algorithms as a series of simpler data processing steps in a natural way. The base of available algorithms is steadily increasing and includes, to name but the most common, Principal Component Analysis (PCA

and NIPALS), several Independent Component Analysis algorithms (CuBICA, FastICA, TDSEP, and JADE), Locally Linear Embedding, Slow Feature Analysis, Gaussian Classifiers, Fisher Discriminant Analysis, Factor Analysis, and Restricted Boltzmann Machine (see **Table 1** for a more exhaustive list and references). Particular care has been taken to make computations efficient in terms of speed and memory. To reduce memory requirements, it is possible to perform learning using batches of data, and to define the internal parameters of the nodes to be single precision, which makes the usage of very large data sets possible. Moreover, an MDP subpackage in its final stages of development offers a parallel implementation of the basic nodes and flows.

From the developer's perspective, MDP is a framework that makes the implementation of new supervised and unsupervised learning algorithms easy and straightforward. The basic class, *Node*, takes care of tedious tasks like numerical type and dimensionality checking, leaving the developer free to concentrate on the implementation of the learning and execution phases. Because of the common interface, the node then automatically integrates with the rest of the library and can be used in a network together with other nodes. A node can have multiple training phases and even an undetermined number of phases. This allows the implementation of algorithms that need to collect some statistics on the whole input before proceeding with the actual training, and others that need to iterate over a training phase until a convergence criterion is satisfied.

MDP is distributed under the open source LGPL license. It has been written in the context of theoretical research in neuroscience, but was designed to be helpful in any context where trainable data processing algorithms are used. Its simplicity on the user's side together with the reusability of the implemented nodes make it also a useful educational tool.

¹<http://numpy.scipy.org>

²<http://www.scipy.org>

³<http://www.mathworks.com/products/matlab/>

⁴<http://mdp-toolkit.sourceforge.net>

Table 1 | Some of the nodes available in MDP.

Node class name	Algorithm and Reference
PCANode	Principal Component Analysis (Jolliffe, 1986)
NIPALSNode	Nonlinear Iterative Partial Least Squares PCA (NIPALS) (Fritzke, 1995)
CuBICANode	Cumulant-based Independent Component Analysis (CuBICA) (Blaschke and Wiskott, 2004)
FastICANode	Independent Component Analysis (FastICA) (Hyvärinen, 1999)
JADENode	Cumulant-based Independent Component Analysis (JADE) (Cardoso, 1999)
TDSEPNode	Temporal blind-source separation algorithm (TDSEP) (Ziehe and Müller, 1998)
LLENode	Locally Linear Embedding Analysis (Roweis and Saul, 2000)
HLLNode	Hessian Locally Linear Embedding Analysis (Donoho and Grimes, 2003)
FDANode	Fisher Discriminant Analysis (Bishop, 1995)
SFANode	Slow Feature Analysis (Wiskott and Sejnowski, 2002)
ISFANode	Independent Slow Feature Analysis (Blaschke et al., 2007)
RBMNode	Restricted Boltzmann Machine (Hinton et al., 2006)
GrowingNeuralGasNode	Growing Neural Gas (learn a graph structure of the data) (Fritzke, 1995)
FANode	Factor Analysis (Bishop, 2007)
GaussianClassifierNode	Supervised gaussian classifier
PolynomialExpansionNode	Expand the signal in a polynomial space
TimeFramesNode	Expand the signal using a sliding temporal window (temporal embedding)
HitParadeNode	Record local minima and maxima in the signal
NoiseNode	Additive and multiplicative noise injection

THE PACKAGE STRUCTURE

The MDP framework consists of a library of data processing nodes with a common Application Programming Interface (API) and a collection of objects which are used to connect nodes together to implement complex data processing workflows. In the following sections the framework structure is outlined followed by an example application. The full API together with an extensive tutorial covering both usage and instruction for writing extensions are available at the MDP homepage.

NODES

A *node* is the basic building block of an MDP application. It represents a data processing element, like for example a learning algorithm, a data filter, or a visualization step (see **Table 1** for a list of some of the available algorithms). Each node is characterized by an input dimension (i.e., the dimensionality of the input vectors), an output dimension, and a *dtype*, which determines the numerical type of the internal structures and of the output signal. By default, these attributes are inherited from the input data.

Nodes can have a *training phase*, where training data is analyzed in order to adapt the internal variables, and an *execution phase*, where new data can be processed using the learned parameters. For example, the Principal Component Analysis (PCA) algorithm (Jolliffe, 1986) requires the computation of the mean and covariance matrix of a set of training data from which the principal eigenvectors of the data distribution are estimated. MDP offers an implementation of this algorithm in the class `PCANode`. The node can be trained on the data using the interface common to all nodes: `PCANode.train(x)` analyzes a new batch of data *x*, and updates the estimation of mean and covariance matrix; `PCANode.stop_training()` finalizes the algorithm by computing and selecting the principal eigenvectors. Once the training is finished,

new data can be projected on the principal components calling the `PCANode.execute(y)` method. If the transformation specified by the underlying algorithm is invertible, the node can also be executed “backwards” using the `PCANode.inverse(z)` method. In the case of PCA, for example, this corresponds to projecting a vector in the principal components space back to the original data space.

Node was designed to be applied to arbitrarily long sets of data: if the underlying algorithms support it, the internal structures can be updated incrementally by sending multiple batches of data. It is thus possible to perform computations on amounts of data that would not fit into memory or to generate data on-the-fly. The general form of the training phase thus is:

```
# create an instance of the desired node
node_instance = mdp.nodes.XXXNode()

for data_batch in data_source:
    node_instance.train(data_batch)

node_instance.stop_training()
```

In the code, *data_source* can be any Python iterator⁵ (e.g. a list, an iterator object, or a generator function) that returns an array with a batch of training data. The last line finalizes the training phase. It is shown here for completeness, but can be replaced by a call to the `execute` or `inverse` methods. Nodes also define some utility methods, like for example `copy` and `save`, that return an exact copy of a node and save it in a file, respectively. Additional methods may be present, depending on the algorithm. The `PCANode.get_projmatrix` method, for example, returns the matrix projecting input data into the principal components’ space. For a toy signal-denoising application that makes use of the basic Node features just described in **Figure 1**.

⁵<http://docs.python.org/lib/typeiter.html>

```

# Simple denoising algorithm
# Given is a set of multidimensional signals, for example
# EEG waves, from which normal statistics are learned,
# and a set of noisy signals to be denoised.

# 1 - Create an instance of the PCA algorithm
#   The argument output_dim = 0.9 tells the node to retain
#   a number of principal components such that the
#   explained variance is at least 90%
#   A fixed number of output components can be specified
#   for example by output_dim=10
pcanode = mdp.nodes.PCANode(output_dim = 0.9)

# 2 - Perform PCA on the set of training signals
pcanode.train(signals)

# 3 - Stop learning and estimate the principal components
pcanode.stop_training()

# 4 - Project noisy signals in the principal component space
proj_signals = pcanode.execute(noisy_signals)

# 5 - Project the data back to the input space for visualization
#   and comparison with original data
denoised_signals = pcanode.inverse(proj_signals)

```

FIGURE 1 | A simple denoising application.

Some nodes, namely the one corresponding to supervised algorithms, e.g. Fisher Discriminant Analysis (Bishop, 1995), may need some labels or other supervised signals to be passed during training:

```

input = {'a': data_a, 'b':data_b, 'c':data_c}
fdanode = mdp.nodes.FDANode()
for label in ['a', 'b', 'c']:
    fdanode.train(input[label], label)

```

A node could also require multiple training phases. For example, the training of `fdanode` is not complete yet, since it has two training phases: The first one computing the mean of the data conditioned on the labels, and the second one computing the overall and within-class covariance matrices and solving the FDA problem. The first phase must be stopped and the second one trained:

```

fdanode.stop_training()
for label in ['a', 'b', 'c']:
    fdanode.train(input[label], label)

```

The easiest way to train multiple phase nodes is using flows, which automatically handle multiple phases (see Flows).

MDP makes it easy to write new nodes that interface with the existing data processing elements. The `Node` class is designed to make the implementation of new algorithms easy and intuitive. This base class takes care of setting input and output dimension and casting the data to match the numerical type (e.g. float or double) of the internal variables, and offers utility methods that can be used by the developer. To expand the MDP library of implemented nodes with user-made nodes, it is sufficient to subclass `Node`, overriding some of the methods according to the algorithm one wants to implement, typically the `_train`, `_stop_training`, and `_execute` methods. **Figure 2** shows an example of a simple node that removes the mean of the signal. A more detailed

introduction to writing new nodes in MDP can be found in the online tutorial⁶.

It is also possible to specify multiple training phases by defining additional training methods and overwriting the `_get_train_seq` method. For example

```

class MultiplePhaseNode(mdp.Node):
    def _get_train_seq(self):
        return [(self._train_A, self._stop_A),
                (self._train_B, self._stop_B)]
    [...]

```

defines a new node with two training phases, one updated by the method `_train_A` and finalized using `_stop_A`, and analogously the second is defined by the methods `_train_B` and `_stop_B`. The final user will still perform the training phase by calling the usual methods `train` and `stop_training` (although multiple times), and need not know about the specific implementation of the algorithm.

FLOWS

A *flow* is a sequence of nodes that are trained and executed together to form a more complex algorithm. Input data is sent to the first node and is successively processed by the subsequent nodes along the sequence. Using a flow as opposed to handling manually a set of nodes has a clear advantage: The general flow implementation automates the training (including supervised training and multiple training phases), execution, and inverse execution (if defined) of the whole sequence. For example, suppose we need to analyze a very high-dimensional input signal using Independent Component Analysis (ICA). To reduce the computational load, we would like to reduce the input dimensionality of the data using PCA. Moreover,

⁶<http://mdp-toolkit.sourceforge.net/tutorial.html>

```

class MeanFreeNode(mdp.Node):
    def __init__(self, input_dim=None, dtype=None):
        super(MeanFreeNode, self).__init__(input_dim=input_dim, dtype=dtype)
        self.avg = None
        self.tlen = 0

    def _train(self, x):
        # Initialize the mean vector with the right
        # size and dtype if necessary:
        if self.avg is None:
            self.avg = mdp.numx.zeros(self.input_dim, dtype=self.dtype)
        # Update the average
        self.avg += mdp.numx.sum(x, axis=0)
        # Update the number of data points examined
        self.tlen += x.shape[0]

    def _stop_training(self):
        # Compute the average signal
        self.avg /= self.tlen

    def _execute(self, x):
        return x - self.avg

    def _inverse(self, y):
        return y + self.avg

```

FIGURE 2 | Definition of a new node that removes the mean of the signal.

we would like to find the data that produces local maxima in the output of the ICA components on a new test set (this information could be used for instance to characterize the ICA filters). To implement this algorithm using MDP, we need to generate an instance of Flow using the appropriate nodes:

```

# Define a data processing sequence.
# - PCANode(output_dim=5) performs PCA and keeps
#       the first 5 principal
#       components only
# - CuBICANode() is a cumulant-based ICA algorithm
# - HitParadeNode(3) records the 3 largest local
#       maxima from the output of
#       the previous node
flow = mdp.Flow([mdp.nodes.PCANode(output_dim=5),
                 mdp.nodes.CuBICANode(),
                 mdp.nodes.HitParadeNode(3)])

```

The training and execution are performed as for the Node class:

```

# Train all the nodes using the data array 'x'
flow.train(x)
# Compute the output of the node sequence
# when presented with array 'x_test'
output = flow.execute(x_test)

```

A single call to the flow's train method will automatically take care of training nodes with multiple training phases, if such nodes are present.

Flow objects are defined as Python containers, and thus are endowed with most of the methods of Python lists: one can obtain slices, append new nodes, pop or insert nodes, and concatenate flows. For example, to get the maxima computed by the

HitParadeNode, one can refer to the last node using the list construct `flow[-1]`:

```
maxima, indices = flow[-1].get_maxima()
```

The Flow class defines a number of utility methods, including save and copy methods. It also implements a crash recovery mechanism that can be activated by setting a flag: in case an exception is thrown during training, the current state of the flow is saved for later inspection.

HIERARCHICAL NETWORKS

In case the desired data processing application cannot be defined as a sequence of nodes, the `hinet` subpackage makes it possible to construct arbitrary feed-forward architectures, and in particular hierarchical networks. It contains three basic building blocks (which are all nodes themselves): Layer, FlowNode, and Switchboard.

The first building block, Layer, works like a horizontal version of flow. It acts as a wrapper for a set of nodes that are trained and executed in parallel. For example, we can combine two nodes with 100-dimensional input to construct a layer with a 200-dimensional input:

```

node1 = mdp.nodes.PCANode(input_dim=100,
                           output_dim=10)
node2 = mdp.nodes.SFANode(input_dim=100,
                           output_dim=20)
layer = mdp.hinet.Layer([node1, node2])

```

The first half of the 200-dimensional input data is then automatically assigned to node1 and the second half to node2. We can train and execute a layer just like any other node. In order to be able to build arbitrary feed-forward node structures, `hinet`

provides a wrapper class for flows (i.e., vertical stacks of nodes) called `FlowNode`. For example, we can replace `node1` in the above example with a `FlowNode`:

```
node1_1 = mdp.nodes.PCANode(input_dim=100,
                             output_dim=50)
node1_2 = mdp.nodes.SFANode(input_dim=50,
                             output_dim=10)
node1_flow = mdp.Flow([node1_1, node1_2])
node1 = mdp.hinet.FlowNode(node1_flow)
node2 = mdp.nodes.SFANode(input_dim=100,
                             output_dim=20)
layer = mdp.hinet.Layer([node1, node2])
```

`node1` has two training phases in this example, one for each internal node. Therefore `layer` now has two training phases as well and behaves like any other node with two training phases. By combining and nesting `FlowNode` and `Layer`, it is thus possible to build complex node structures.

When implementing networks one might have to route different parts of the data to different nodes in a layer in complex ways. This is done by the `Switchboard` node, which can handle such routing. A `Switchboard` is initialized with a 1-dimensional array with one entry for each output connection, containing the corresponding index of the input connection that it receives its input from, e.g.:

```
switchboard = mdp.hinet.Switchboard(
    input_dim=6,
    connections=[0,1,2,3,4,3,4,5])
print switchboard
# should print: Switchboard(input_dim=6,
#                             output_dim=8,
#                             dtype=None)
x = mdp.numx.array([[2,4,6,8,10,12]])
print switchboard.execute(x)
# should print:
# array([[ 2,  4,  6,  8, 10,  8, 10, 12]])
```

The switchboard can then be followed by a layer that splits the routed input to the appropriate nodes, as illustrated in **Figure 3**.

Since hierarchical networks can become quite complicated to build and debug, `hinnet` includes the class `HiNetHTML` that translates an MDP flow into a graphical visualization in an HTML file.

A COMPLETE APPLICATION

In this section we show a complete example of MDP usage in a machine learning application, and use non-linear Slow Feature Analysis for processing of non-stationary time series. We consider a chaotic time series derived by a logistic map (a demographic model of the population biomass of species in the presence of limiting factors such as food supply or disease) that is non-stationary in the sense that the underlying parameter is not fixed but is varying smoothly in time. The goal is to extract the slowly varying parameter that is hidden in the observed time series. This example reproduces some of the results reported in Wiskott (2003). The complete code is shown in **Figure 4**.

We first generate the slowly varying driving force parameter r_t as a combination of three sine waves $r_t = \sin(10\pi t) + \sin(22\pi t) + \sin(26\pi t)$. We then generate the time series using the logistic

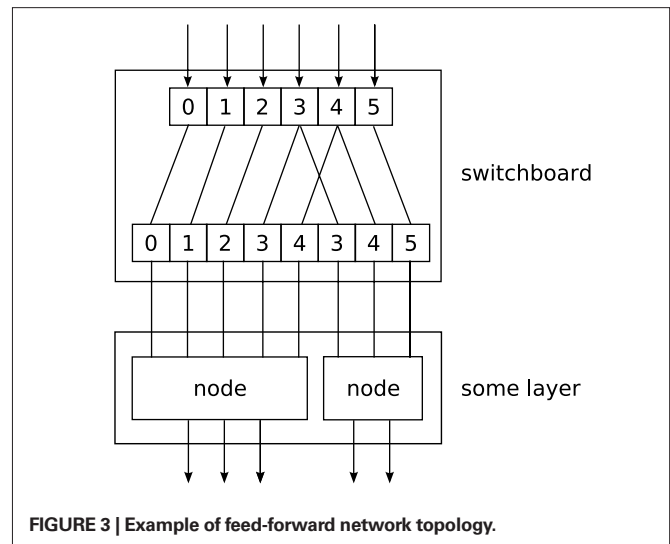


FIGURE 3 | Example of feed-forward network topology.

equation $x_{t+1} = (3.6 + 0.13r_t)x_t(1 - x_t)$. The resulting time series x is shown in **Figure 5**.

To reconstruct the underlying parameter, we define a `Flow` to perform SFA in the space of polynomials of degree 3. We first use a node that embeds the 1-dimensional time series in a 10-dimensional space using a sliding temporal window of size 10 (`TimeFramesNode`). Second, we expand the signal in the space of polynomials of degree 3 using a `PolynomialExpansionNode`. Finally, we perform SFA on the expanded signal and keep the slowest feature using the `SFANode`. In order to measure the slowness of the input time series before and after processing, we put at the beginning and at the end of the node sequence a node that computes the η -value (a measure of slowness, see Wiskott and Sejnowski, 2002) of its input (`EtaComputerNode`). The slow feature should match the driving force up to a scaling factor, a constant offset and the sign. To allow a direct comparison we rescale the driving force to have zero mean and unit variance. The real driving force is plotted together with the driving force estimated by SFA in **Figure 6**.

FUTURE DEVELOPMENT

MDP is currently maintained by a core team of three developers, but it is open to user contributions. Users have already contributed some of the nodes, and more contributions are currently being reviewed for inclusion in future releases of the package. The package development can be followed on the public subversion code repository⁷. Questions, bug reports, and feature requests are typically handled by the user mailing list⁸.

Development of the core functionality of MDP continues and the next release of MDP is going to include a new package for parallelization, designed for nodes in which a large part of the computation is *embarrassingly parallel*⁹ (e.g. calculating the covariance

⁷<http://mdp-toolkit.svn.sourceforge.net>

⁸http://sourceforge.net/mail/?group_id=116959

⁹In the jargon of parallel computing, an *embarrassingly parallel* problem is one for which no particular effort is needed to segment the problem into a very large number of parallel tasks, that can be executed more or less independently, without communication among tasks (Foster, 1995, Section 1.4.4.).


```

import mdp
N = mdp.numx

def logistic_map(x,r):
    return r*x*(1-x)

# time axis is 1 second sampled at 10KHz
t = N.linspace(0,1,10000,endpoint=0)
# driving force
dforce = N.sin(10*N.pi*t) + N.sin(22*N.pi*t) + N.sin(26*N.pi*t)

# resulting time series
series = N.zeros((10000,1),'d')
series[0] = 0.6 # initial condition
for i in range(1,10000):
    series[i] = logistic_map(series[i-1],3.6+0.13*dforce[i])

# define the flow
sequence = [mdp.nodes.EtaComputerNode(), mdp.nodes.TimeFramesNode(10),
            mdp.nodes.PolynomialExpansionNode(3), mdp.nodes.SFANode(output_dim=1),
            mdp.nodes.EtaComputerNode()]

flow = mdp.Flow(sequence, verbose=1)
# train the flow
flow.train(series)

# execute the flow to get the SFA estimate of the driving force
slow = flow.execute(series)

# rescale driving force to compare with SFA estimate
resc_dforce = (dforce - N.mean(dforce,0))/N.std(dforce,0)

# verify that the results are correct
# result should be > 0.99
print mdp.utils.cov2(resc_dforce[:-9],slow)
# result should be ~ 3000
print 'Eta value (time-series): ', flow[0].get_eta(t=10000)
# result should be ~ 10
print 'Eta value (slow feature): ', flow[-1].get_eta(t=9996)

```

FIGURE 4 | Python code to reproduce the results in Wiskott (2003).

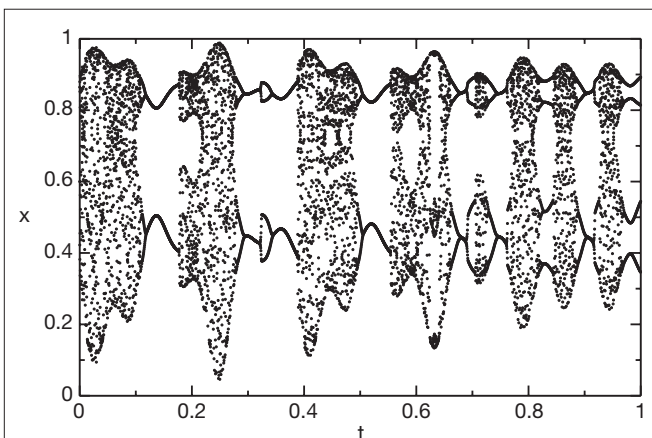


FIGURE 5 | Chaotic time series generated by the logistic equation.

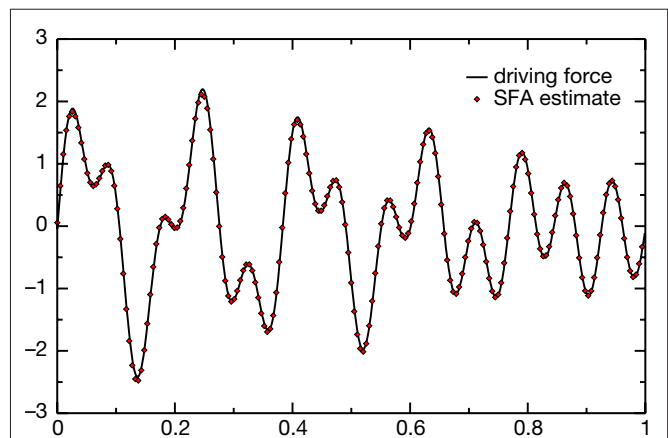


FIGURE 6 | The real driving force and the driving force as estimated by SFA.

matrix to perform PCA). The new parallel package will consist of two parts: The first part introduces parallel versions of the familiar MDP structures (nodes and flows, including `hinet`) that are able to split the computations for some of the algorithms (e.g. PCA and

SFA). The second part of the package consists of schedulers that take individual jobs and execute them in a parallel way. Currently a scheduler for parallelization across multiple processors (or cores) is provided. Since the scheduler code is largely independent of

MDP, one can write simple adapters for other schedulers like for example Parallel Python¹⁰. The new `parallel` subpackage can be tested already and it is available on the public code repository.

Another new, large MDP package is currently under development that will extend MDP with more complex data flows, including back-propagation and loops. This framework will be integrated with both the parallel and the `hinet` package to allow for large and complex data processing networks.

MDP could also act efficiently as a wrapper for the plethora of statistical data analysis algorithms already available in other libraries and languages. A prominent example is the R Project for Statistical Computing¹¹ with the Python wrappers RPy¹² and R/S Plus¹³.

CONCLUSIONS

With over 10,000 downloads since its first public release in 2004, MDP has become one of Python's major scientific packages. The package has minimal dependencies, requiring only the NumPy numerical extension, is completely platform-independent, and is available in the Linux Debian distribution and the Python(x,y)¹⁴ scientific Python distribution.

MDP has been used to implement a model of the visual system of a virtual rat moving around in a virtual environment (Franzius et al., 2007), to perform pattern recognition (Franzius et al., 2008) and handwritten digit recognition (Berkes, 2006), to analyze

intra-cerebral array-recorded neurophysiological data in the auditory forebrain of song birds¹⁵, and to perform PCA and spike-sorting of electrophysiological data (Wiltschko et al., 2008), to name a few of the applications in computational neuroscience. MDP has also been used embedded in the X-ray fluorescence mapping package PyMCA (Solé et al., 2007), to implement auto tagging capabilities into the personal organizer application Chandler¹⁶ by OSAF¹⁷, and as a framework for the implementation of data processing algorithms in the context of an advanced course in scientific computing (Zito and Wilson, 2008) aimed at graduate students.

As the number of its users and contributors is increasing, MDP appears to be a good candidate for becoming a community-driven common repository of user-supplied, freely available, Python implemented data processing algorithms.

ACKNOWLEDGMENTS

We wish to heartily thank Mathias Franzius for discussion and help during the early phases of the project, for being our main beta-tester afterwards, and for his code contributions. For contributing code and comments we thank Gabriel Beckers, Farzad Farkhooi, Susanne Lezius, Michael Schmuker, and Jake VanderPlas. For maintaining the Debian package we are grateful to Yaroslav Halchenko. We finally wish to acknowledge all those users who reported bugs and feature requests, which helped us making MDP a better library.

¹⁰<http://www.parallelpython.com>

¹¹<http://www.r-project.org/>

¹²<http://rpy.sourceforge.net/>

¹³<http://www.omegahat.org/RSPython/>

¹⁴<http://www.pythonxy.com>

REFERENCES

- Berkes, P. (2006). Temporal Slowness as an Unsupervised Learning Principle. Ph.D. Thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät I, <http://edoc.hu-berlin.de/docviews/abstract.php?id=26704>.
- Bishop, C. M. (1995). Neural Networks for Pattern Recognition. New York, NY, Oxford University Press.
- Bishop, C. M. (2007). Pattern Recognition and Machine Learning. New York, NY, Springer-Verlag.
- Blaschke, T., and Wiskott, L. (2004). CuBICA: independent component analysis by simultaneous third- and fourth-order cumulant diagonalization. *IEEE Trans. Signal Process.* 52, 1250–1256.
- Blaschke, T., Zito, T., and Wiskott, L. (2007). Independent slow feature analysis and nonlinear blind source separation. *Neural Comput.* 19, 994–1021.
- Cardoso, J. (1999). High-order contrasts for independent component analysis. *Neural Comput.* 11, 157–192.
- Donoho, D. L., and Grimes, C. (2003). Hessian eigenmaps: locally linear embedding techniques for high-dimensional data. *Proc. Natl. Acad. Sci. U.S.A.* 100, 5591–5596.
- Foster, I. (1995). Designing and Building Parallel Programs. Reading, MA, Addison-Wesley.
- Franzius, M., Sprekeler, H., and Wiskott, L. (2007). Slowness and sparseness lead to place, head-direction, and spatial-view cells. *PLoS Comput. Biol.* 3, e166.
- Franzius, M., Wiskott, N., and Wiskott, L. (2008). Invariant object recognition with slow feature analysis. In Proceedings of the 18th International Conference on Artificial Neural Networks (ICANN 2008), Prague, Czech Republic, September 3–6, 2008. Lecture Notes in Computer Science Series, Part I, Vol. 5163 (Berlin, Springer Verlag). <http://www.springerlink.com/content/v20024g580t1/>.
- Fritzke, B. (1995). A growing neural gas network learns topologies. In Advances in Neural Information Processing Systems 7. Proceedings of the 1994 Conference, November 28 to December 1, 1994, Denver, Colorado, G. Tesauro, D. S. Touretzky and T. K. Leen, eds. (Cambridge, MIT Press), pp. 625–632.
- Hinton, G., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Comput.* 18, 1527–1554.
- Hyvärinen, A. (1999). Fast and robust fixed-point algorithms for independent component analysis. *IEEE Trans. Neural Netw.* 10, 626–634.
- Jolliffe, I. (1986). Principal Component Analysis. New York, NY, Springer-Verlag.
- Roweis, S., and Saul, L. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science* 290, 2323–2326.
- Solé, V. A., Papillon, E., Cotte, M., Walter, P., and Susini, J. (2007). A multiplatform code for the analysis of energy-dispersive x-ray fluorescence spectra. *Spectrochim. Acta Part B* 62, 63–68.
- Wiltschko, A. B., Gage, G. J., and Berke, J. D. (2008). Wavelet filtering before spike detection preserves waveform shape and enhances single-unit discrimination. *J. Neurosci. Methods* 173, 34–40.
- Wiskott, L. (2003). Estimating Driving Forces of Nonstationary Time Series with Slow Feature Analysis. arXiv.org e-Print archive, <http://arxiv.org/abs/cond-mat/0312317/>.
- Wiskott, L., and Sejnowski, T. (2002). Slow feature analysis: unsupervised learning of invariances. *Neural Comput.* 14, 715–770.
- Ziehe, A., and Müller, K.-R. (1998). TDSEP – an efficient algorithm for blind separation using time structure. In Proceedings of the 8th International Conference on Artificial Neural Networks (ICANN 1998), Vol. 2, M. B. Boden, L. F. Niklasson and T. Ziemke, eds. (London, Springer), pp. 675–680.
- Zito, T., and Wilson, G. (2008). Software Carpentry for Scientists. <http://itb.biologie.hu-berlin.de/~zito/teaching/SCI/>.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 05 September 2008; paper pending published: 26 October 2008; accepted: 19 December 2008; published online: 08 January 2009.

Citation: Zito T, Wiskott L and Berkes P (2009) Modular toolkit for Data Processing (MDP): a Python data processing framework. *Front. Neuroinform.* (2009) 2:8. doi: 10.3389/neuro.11.008.2008
Copyright © 2009 Zito, Wiskott and Berkes. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



PyMOOSE: interoperable scripting in Python for MOOSE

Subhasis Ray and Upinder S. Bhalla*

National Centre for Biological Sciences, Bangalore, India

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Michael Hines, Yale University, USA
Hugo Cornelis, UTHSCSA, USA

***Correspondence:**

Upinder S. Bhalla, National Centre for
Biological Sciences, Tata Institute of
Fundamental Research, Bellary Road,
Bangalore 560065, India.
e-mail: bhalla@ncbs.res.in

Python is emerging as a common scripting language for simulators. This opens up many possibilities for interoperability in the form of analysis, interfaces, and communications between simulators. We report the integration of Python scripting with the Multi-scale Object Oriented Simulation Environment (MOOSE). MOOSE is a general-purpose simulation system for compartmental neuronal models and for models of signaling pathways based on chemical kinetics. We show how the Python-scripting version of MOOSE, PyMOOSE, combines the power of a compiled simulator with the versatility and ease of use of Python. We illustrate this by using Python numerical libraries to analyze MOOSE output online, and by developing a GUI in Python/Qt for a MOOSE simulation. Finally, we build and run a composite neuronal/signaling model that uses both the NEURON and MOOSE numerical engines, and Python as a bridge between the two. Thus PyMOOSE has a high degree of interoperability with analysis routines, with graphical toolkits, and with other simulators.

Keywords: simulators, compartmental models, systems biology, NEURON, GENESIS, multi-scale models, Python, MOOSE

INTRODUCTION

In computational biology there are two approaches to developing a simulation. First, write your custom program to do a specific simulation, and second, write a model and run it in a general-purpose simulator. While the first approach is very common, it requires the scientist to be a good programmer (or have one at her/his disposal) and moves the focus towards programming rather than science. Furthermore, it is very difficult for others to read such a program and understand how it relates to the targeted biological system. In this context, a model is a well-defined set of equations and parameters that is meant to represent and predict the behavior of a biological system. Ideally, a general-purpose simulator allows the model to be separated from the low-level data-structures and control. The scientist is no longer concerned with minutiae of software engineering and can concentrate on the biological system of interest. The model can be shared by other people and understood relatively easily using intermediate-level descriptions of the model with a more obvious mapping to the real biological system. General simulators also lend themselves to declarative, high-level model descriptions that have now become important part of scientific interchange in the computational neuroscience and systems biology communities (Beeman and Bower, 2004; Cannon et al., 2007; Goddard et al., 2001; Hucka et al., 2002; <http://www.morphml.org/>; <http://neuroml.org>, <http://sbml.org>). The goal of this paper is to show how the simulator Multi-scale Object Oriented Simulation Environment (MOOSE; <http://moose.ncbs.res.in/>, mirrored at <http://moose.sourceforge.net/>) uses Python to address these issues of interoperability with analysis software, graphical interfaces, and other simulators.

General-purpose simulators have been in use since the venerable circuit simulator SPICE was utilized to solve compartmental models (Bunow et al., 1985; Segev et al., 1985). While this level of generality ran into limitations of computing power, more specialized neuronal simulators such as GENESIS and NEURON (Bower and

Beeman, 1998; Carnevale and Hines, 2006; Hines, 1993) included optimized custom code that would allow the simulation to be run in affordable time and memory. This process of building domain-specific general simulators has continued with several simulators devoted to different aspects of computational and systems biology (e.g., VCell, Smoldyn, COPASI). This proliferation of simulators brings back the problems of model exchange and interoperability, albeit at a higher-level than raw Fortran or C code. While these simulators now have a common set of shared higher-level concepts (e.g., compartments, channels, synapses), they use entirely different vocabularies and languages for set up and control.

MOOSE is a new simulator project that supports simulations across a wide range of scales in computational biology, including computational neuroscience and systems biology. In order to improve interoperability, MOOSE uses two existing languages: the GENESIS scripting language, and Python. The Neurospaces (Cornelis and De Schutter, 2003; <http://neurospaces.sourceforge.net/>) project takes a distinct approach to supporting some GENESIS capabilities using backward-compatible scripting, and it too can utilize Python.

Most established simulators have their own scripting languages. For example, NEURON uses hoc along with modl files to set up simulations. GENESIS has its own custom scripting language. MOOSE avoids introducing a new language, and instead inherits the GENESIS parser. To increase compatibility, MOOSE has equivalents for most objects in GENESIS, and many old scripts can be run on MOOSE with little or no modification. Given these existing capabilities, why add Python scripting? Despite its flexibility, the GENESIS scripting language has several limitations:

1. Domain specificity: It is not used outside GENESIS. This forces the user to learn a special-purpose scripting language.
2. Problem with extensibility: While it is easy to write a script to define functions that can be included in other scripts, these

interpreted functions are much slower than compiled code. The GENESIS scripting language itself provides for some degree of extensibility, but this is difficult to implement. Adding a single command requires implementation in C, as well as definition of the command in a configuration file that must be pre-processed to include into the interpreter. The addition of a new class is still more involved.

3. Lack of existing libraries: The GENESIS scripting language is a special-purpose language and has no additional features other than those written into the language.
4. Syntax: The syntax is complex and inconsistent as a result of accretion of features by many developers and users. For example, arrays are implemented in three inconsistent ways in the GENESIS scripting language: as arrays of elements, entries within tables and extended fields.

To harness the capabilities provided by a modern widely used scripting language, we chose a Python interface. Among the plethora of programming languages, Python has some special advantages:

1. Interactive: We need a scripting language that comes with a command line interpreter. Python is suited for this. User interaction is as important as running standalone scripts. Simulations are built incrementally, and it is important that users can try out bits and pieces of code and get quick feedback from the system. Moreover, this practice helps in identifying errors early in the development process, which saves considerable time and computational resources.
2. It is easy to interface with other programming languages: Python itself is written in C. It has a standard developers' API for creating extension libraries. This simplifies creating Python interface for C/C++ code. Moreover, tools like Simplified Wrapper and Interface Generator (SWIG), Qt sip, boost-Python can automate the task of creating a Python interface from existing C/C++ code.
3. It is portable: Python runs on Linux, Solaris, Macintosh and Windows operating systems and many other platforms (<http://www.python.org/about/>).
4. Free: Python is free and open-source.
5. Widely used: Python is widely used in scientific community. There is a large repertoire of third-party libraries for Python. Many of these libraries are free, open source and mature.

In this study we show how PyMOOSE harnesses each of these capabilities.

MATERIALS AND METHODS

There are two common approaches to create a Python interface to a C/C++ library: (1) statically link it with the Python interpreter – which involves compiling the Python interpreter source-code, (2) create a dynamic link library and provide it as a Python module. We took the second approach as it provides more flexibility on the choice of the Python interpreter and reduces the burden on the maintainer.

MAPPING MOOSE CLASSES INTO PYTHON

MOOSE has a set of built-in classes for representing simulation entities. These classes provide a mapping from the concept space

to the computational space. Physical or chemical properties and other relevant parameters are accessible as member fields of the classes and the time-evolution of these parameters is calculated by a special process method of each class. These classes add another layer over ordinary C++ classes to provide messaging and scheduling as well as customized access to the member fields. MOOSE provides introspection (Maes, 1987; Smith, 1982), so that full field information for each class is accessible to the programmer. This class information is statically initialized for each class at startup time. We utilized this class information and SWIG (Beazley, 1996; <http://www.swig.org>) to build the Python interface.

SWIG is a mature software with good support for Python and C/C++ interfacing as well as many other languages. While it is rather simple to create an interface for ordinary C++ class using SWIG, our task was complicated because MOOSE classes have another layer over ordinary C++ classes. For this reason we created a framework for Python interface with additional C++ classes to wrap MOOSE classes and a few classes to manage the system.

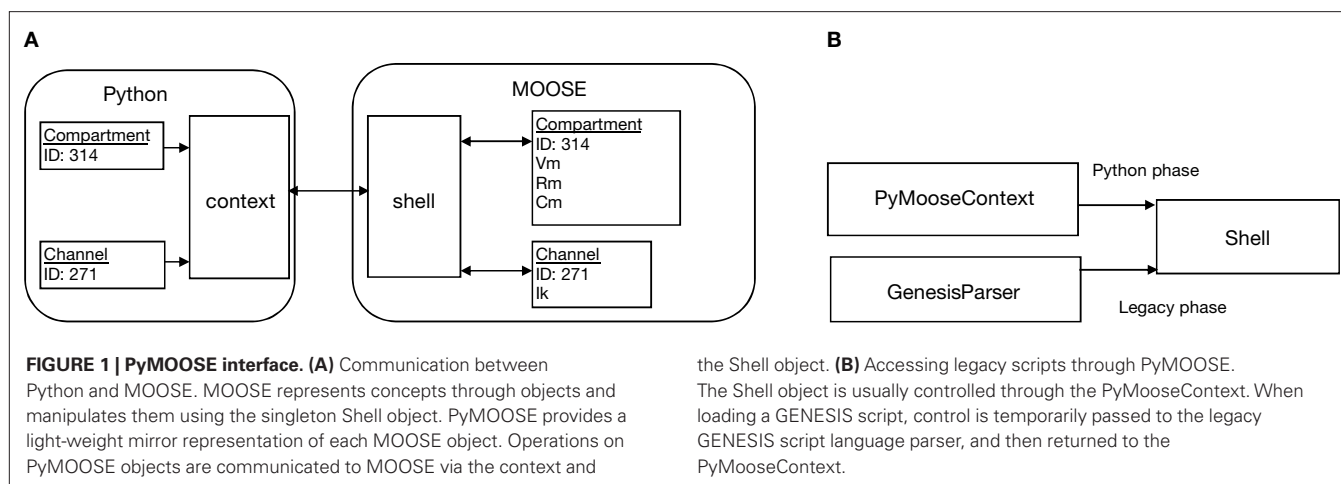
SIMULATOR CONTROL THROUGH PYTHON

All operations on MOOSE objects are carried out via a special class, Shell, of which there is a single instance on each processor node that is running MOOSE. In PyMOOSE we implemented a singleton context object to communicate with the Shell. The context object provides a set of functions that can be called to pass appropriate messages to the Shell. The user can call global MOOSE functions by calling the corresponding methods of the context object. Operations like creation of objects, setting integration time step, running the simulation are all done through the context object.

We created a one-to-one mapping of MOOSE classes to Python classes by means of light-weight C++ wrapper classes. All the wrapper classes were derived from one common base class. Each MOOSE object is identified by an Identifier (ID) field. The main data content of a wrapper class instance is the ID of the corresponding object in MOOSE. Additionally, the wrapper classes have a static pointer to the single instance of the context object. Wrapper classes provide accessor methods that can be used to access the fields in the corresponding MOOSE object.

These C++ wrapper classes were input to SWIG to create the Python module. After translation to Python, the user sees the member fields in the Python classes in place of the accessor methods in the C++ wrapper classes. Behind the scene the Python interpreter calls these accessor methods whenever the user script tries to access MOOSE object fields (**Figure 1A**).

Manually developing C++ wrapper classes for all MOOSE classes was a tedious but repetitive task. We therefore embedded stub code in the MOOSE initialization code to generate most of the wrapper code programmatically using Run-Time Type Information in C++. This auto-generated code was used with a few modifications to generate a Python module using SWIG. SWIG takes an interface file with SWIG-specific directives and generates a single C++ file for the library and a Python source-code file that contains support code. We completed the PyMOOSE code generation by compiling and linking the SWIG-generated C++ source-code as a dynamic library. This dynamic library can be imported in any Python program.



LEGACY MODELS AND PyMOOSE

The PyMOOSE context object keeps a single instance of the GenesisParser class in order to run legacy GENESIS scripts. Whenever the user asks for executing a GENESIS statement, the context object disconnects itself from the Shell and connects the GenesisParser object instead. The GENESIS statement string is passed to the GenesisParser object, which executes it as if the user typed it in at the MOOSE command prompt. After execution of the statement (or script) the GenesisParser object is disconnected from the Shell and the context object is reconnected (**Figure 1B**).

While it is valuable to run GENESIS scripts within PyMOOSE, this feature is intended only to support legacy code and is better avoided in new model development. The use of GENESIS scripting language inside Python defeats the whole purpose of moving to a general-purpose programming language. It reduces readability and the user needs to know both languages in order to understand the code.

RESULTS

We used the Python interface of MOOSE to achieve three key targets: (1) Interfacing with standard libraries in a mature scientific computing language, (2) giving access to a portable GUI library for developing user interface and (3) enabling MOOSE to work together with other simulators.

INTERFACING SIMULATIONS WITH PYTHON LIBRARIES

We used Python scientific and graphing libraries to analyze and display the output of a PyMOOSE simulation. The interface with Python gives the user freedom to choose from a wide variety of scientific and numerical libraries available from third parties. We demonstrate the use of two libraries along with PyMOOSE for developing simulations with plotting and data analysis within Python. The first of these, NumPy, is a library that provides data structures and algorithms for fast matrix manipulation (<http://numpy.scipy.org/>). Even though Python is interpreted, with attendant slow execution, NumPy library provides access to compiled code and hence the functions from the library are as fast as compiled code. The second library, matplotlib, provides a rich set of functions for plotting 2D data both in hardcopy formats and interactively (<http://matplotlib.sourceforge.net/>). It can use NumPy for

fast matrix operations in Python and several portable GUI toolkits (GTK/Qt/Tk/wxWidgets) as graphical back-end.

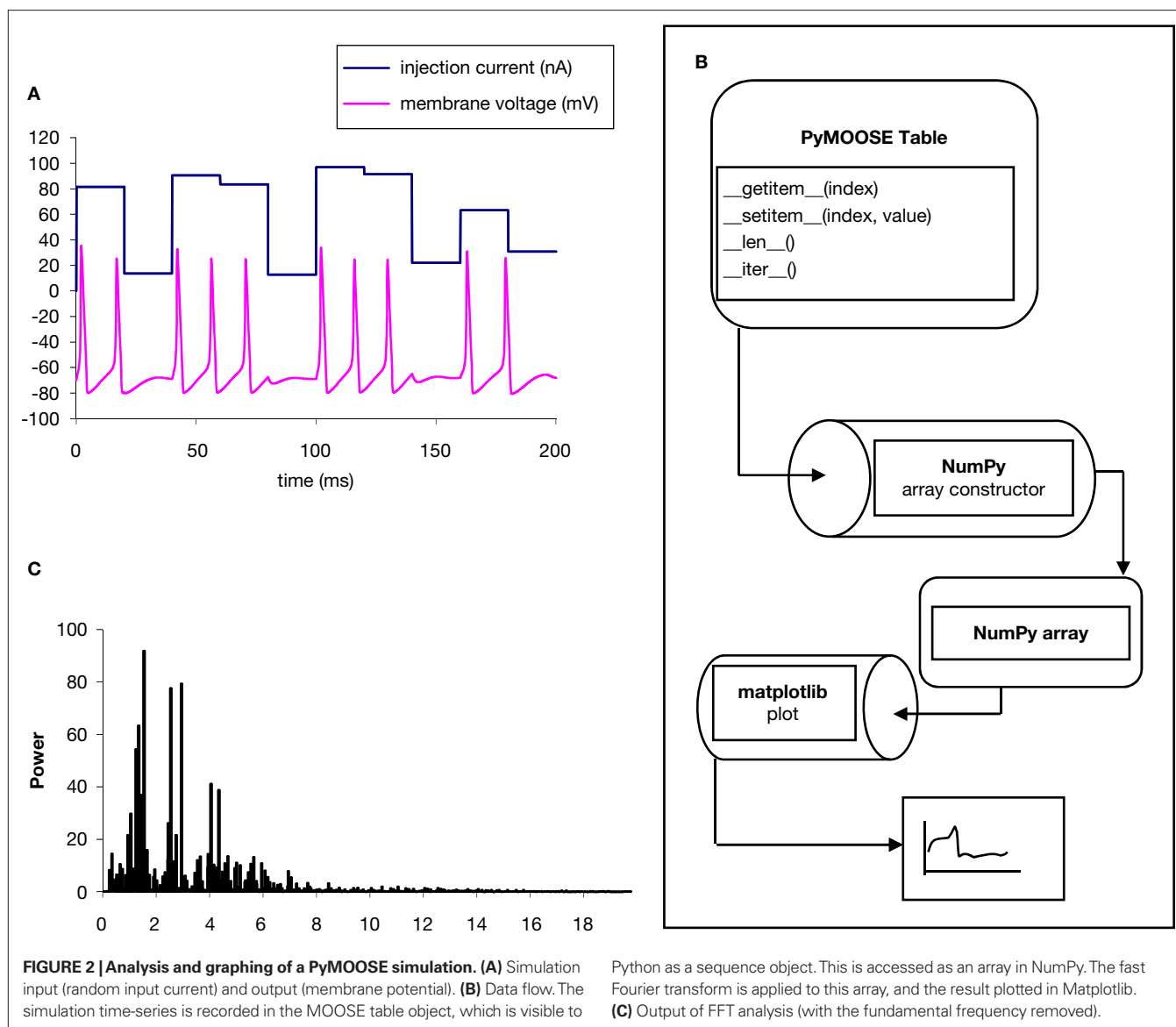
We implemented a simulation of the squid giant axon using Hodgkin–Huxley Na⁺ and K⁺ channels and parameters (script attached in Appendix). We applied an injection current with random amplitude uniformly distributed between 0 and 100 nA. We recorded the time-series for the membrane potential during the simulation in a MOOSE table object, which can accumulate a time-series of simulation output (**Figure 2A**). The interface to Python was done using the MOOSE table class. This class is exposed to Python with methods to emulate iterable type (Martelli et al., 2005). The array constructor in NumPy accepts an iterable object and creates a NumPy array with a copy of the contents of the object. Thus the user is relieved of explicitly iterating over the table entries and copying them to a NumPy array. This completes the interface from the MOOSE simulation output to NumPy (**Figure 2B**). We used the fast Fourier transform operation available in NumPy to compute the discrete Fourier transform of the time-series of the simulated membrane potential. We used matplotlib to plot the original time-series, as well as the output of the FFT (**Figure 2C**).

Overall, this example simulation illustrates how PyMOOSE facilitates interoperability of Python numerical and graphing libraries with MOOSE.

PORTABLE GUI THROUGH PYTHON

The use of Python separates the problem of GUI development from simulator development. Moreover, it gives one the freedom to choose from a number of free GUI toolkits. The major platform independent GUI toolkits with Python interfaces are Qt(TM) available as PyQt, wxWidgets (wxPython), Tk and GTK (<http://wiki.python.org/moin/GuiProgramming>; <http://www.python.org/doc/faq/gui/>). We used PyQt4 to develop a simple user interface for a clone of the GENESIS squid tutorial in MOOSE. We selected Qt4 as it is a mature and clean toolkit that is freely distributed and runs well on all the major operating systems.

The program was divided into three modules – (1) the squid axon compartment with Hodgkin–Huxley channels, (2) a model object which combined a few tables with the squid compartment to record various parameters through the time of the simulation, and



(3) the GUI to take user inputs and to plot data. We implemented the squid axon model as described in the previous section, using PyMOOSE to set up and parameterize the model. As before, the model was interfaced with table objects to monitor time-series output of the simulation. Finally, we implemented the GUI by loading in the PyQt4 libraries, and using Python calls to set up the interface (**Figure 3**). While there are Qt IDEs available (<http://trolltech.com/products/qt/>), we constructed the interface through explicit Python calls to create widgets, assign actions, and manage output data. Qt uses a signal-slot mechanism for passing event information. PyQt allows the use of arbitrary Python methods to be used as slots. Hence we could connect the GUI widgets to methods in the PyMOOSE model class and thus provided simulation control through the GUI in a clean manner. We used PyQwt, a Python interface of the Qt-based plotting library Qwt, for creating output graphs. Since PyQwt can take NumPy arrays as data, we converted the tables in MOOSE to NumPy arrays and used PyQwt plotting widgets to display them.

We based the layout of the simulation on the widely used GENESIS Squid tutorial program. To confirm portability of the system, we ran the model on Linux as well as the Windows operating system.

This exercise demonstrated the capability of PyMOOSE to draw upon existing graphical libraries for its graphical requirements. This is an important departure from GENESIS. The GENESIS graphical libraries (XODUS) were an integral part of the C code-base and XODUS objects were visible as, and manipulated in the same way as other GENESIS objects. In contrast, PyMOOSE did not need to implement any graphical objects within the MOOSE C++ code, but instead reused extant third-party graphical libraries available for Python. Furthermore the existing libraries are professionally designed and have a much more consistent look-and-feel than did the original GENESIS graphical library, XODUS (Bhalla, 1998).

SIMULATOR INTEROPERABILITY

With Python becoming a popular language for developing platform independent scripts, several neuronal simulators have implemented

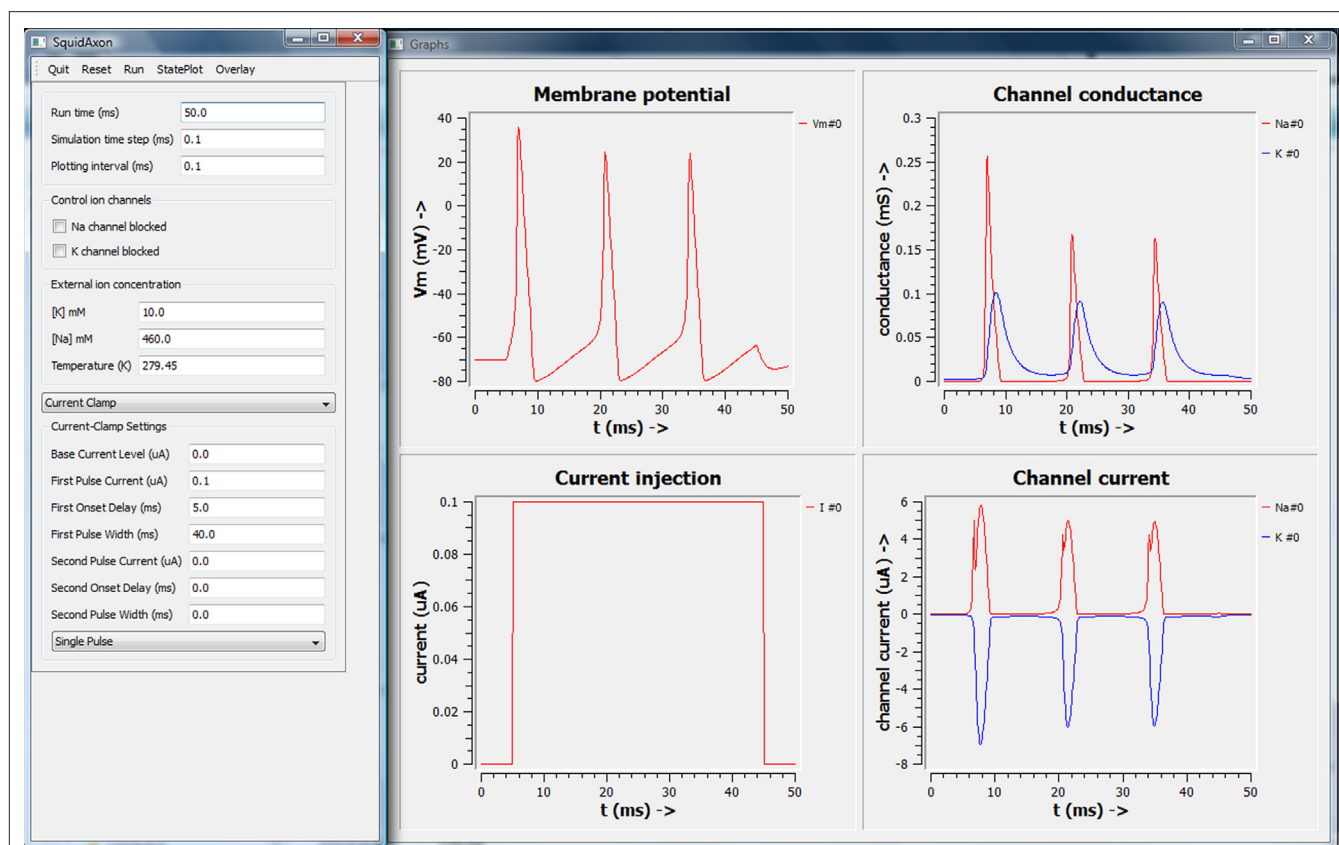


FIGURE 3 | Screen shot of PyMOOSE/Qt interface for the Hodgkin–Huxley model. The layout is closely modeled on the Squid demo from GENESIS.

Python interfaces. This raises the possibility of using Python as a glue language to run simulations that span different simulators. As a final demonstration of interoperability, we used PyMOOSE with PyNEURON to build a multi-scale, multi-simulator model that incorporates neuronal electrical activity as well as biochemical signaling (**Figure 4A**).

We used NEURON to model a multicompartmental electrical model of a Type A neuron from the CA3b region of the rat hippocampus (Migliore et al., 1995; <http://senselab.med.yale.edu/ModelDB/ShowModel.asp?model=3263>). This is a morphologically detailed model with experimentally constrained distribution of membrane ion channels. It reproduces experimental observations of firing behavior and intracellular Ca^{2+} dynamics. We modified the hoc script for the model, to run it for arbitrary time intervals. We directed the output data to Vector objects in NEURON. The Python wrapper class for this model provided a handle for the simulation parameters and functions defined in the hoc script. As described in the PyNEURON documentation (<http://www.neuron.yale.edu/neuron/docs/help/neuron/neuron/classes/python.html>), Python commands were directed to the NEURON engine by constructing hoc statement strings and executing them through the hoc interpreter instance provided by the neuron module. Moreover, hoc object references are directly available in Python as attributes of the hoc interpreter object. Thus accessing hoc objects was quite clean in Python (**Figure 4A**).

We used MOOSE to model calcium-triggered biochemical signaling events at the synapse. We used a model of a bistable MAPK-PKC-PLA2 feedback loop that was originally implemented in GENESIS/Kinetikit (Ajay and Bhalla, 2004; Bhalla and Iyengar, 1999; Bhalla et al., 2002) and uploaded to the DOQCS database (<http://doqcs.ncbs.res.in/template.php?&y=accessiondetails&an=79>). The model was defined in the GENESIS scripting language. We used the legacy scripting mode of PyMOOSE to load the GENESIS/kinetikit model. The simulation objects thus instantiated were standard MOOSE objects, and were accessible using Unix-like path strings. The PyMOOSE interface exposed these objects as regular Python objects. Thus access to the MOOSE objects, representing GENESIS data concepts, was also straightforward in Python (**Figure 4A**).

We used the Python interface to accomplish three critical operations to combine the two simulations: (1) Initialization, (2) run-time control and synchronization, and (3) variable communication and rescaling.

1. To initialize the models, we used PyNEURON command *load_file* to load the hoc script. Once the script is loaded, variables and functions defined in the script become available as members of the hoc interpreter instance inside Python. In this case we defined a setup function to initialize the NEURON simulation. This function is called in the constructor (`__init__`) of the Python wrapper class over the NEURON simulation. At this

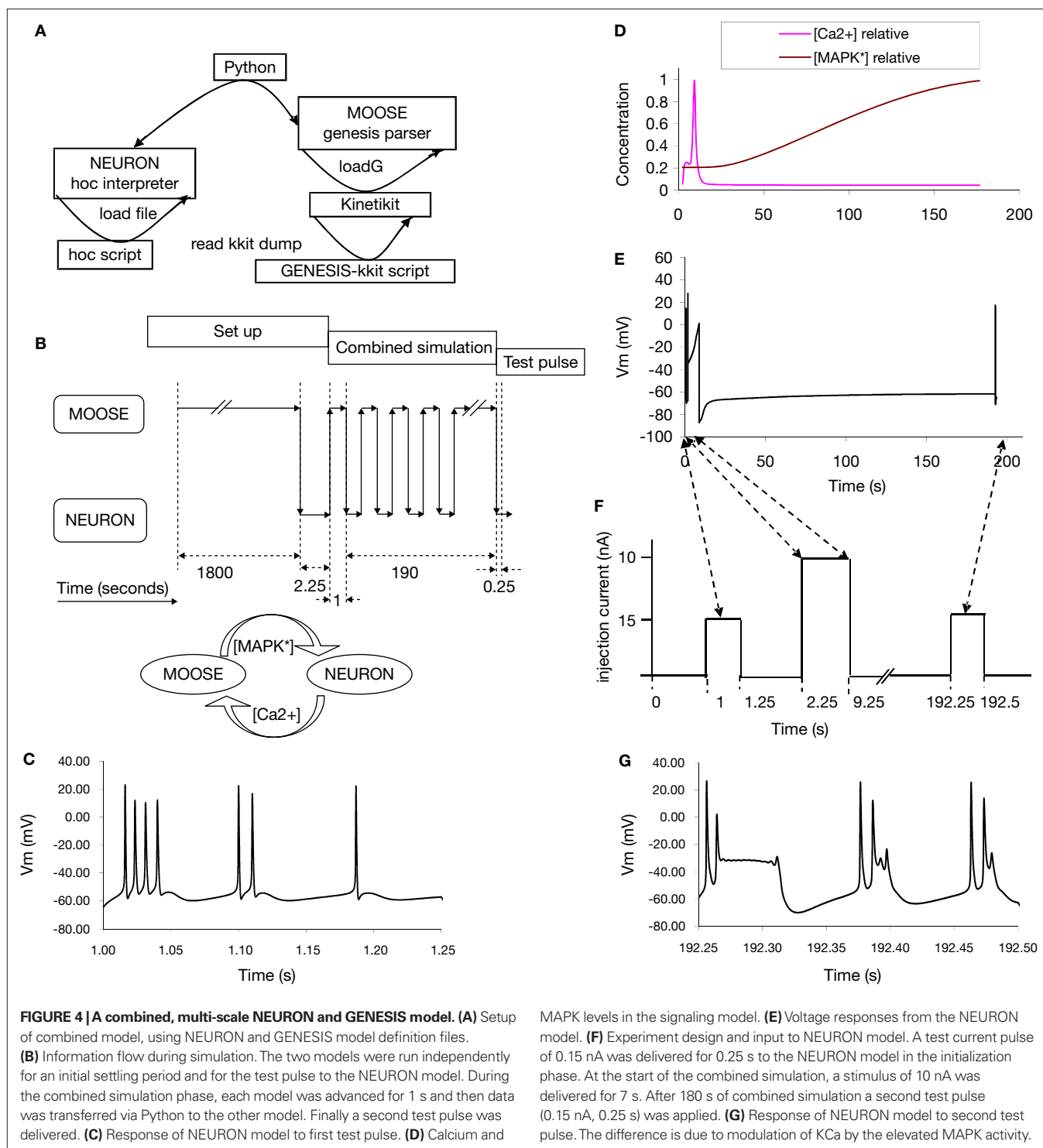


FIGURE 4 | A combined, multi-scale NEURON and GENESIS model. (A) Setup of combined model, using NEURON and GENESIS model definition files.

(B) Information flow during simulation. The two models were run independently for an initial settling period and for the test pulse to the NEURON model. During the combined simulation phase, each model was advanced for 1 s and then data was transferred via Python to the other model. Finally a second test pulse was delivered. **(C)** Response of NEURON model to first test pulse. **(D)** Calcium and

MAPK levels in the signaling model. **(E)** Voltage responses from the NEURON model. **(F)** Experiment design and input to NEURON model. A test current pulse of 0.15 nA was delivered for 0.25 s to the NEURON model in the initialization phase. At the start of the combined simulation, a stimulus of 10 nA was delivered for 7 s. After 180 s of combined simulation a second test pulse (0.15 nA, 0.25 s) was applied. **(G)** Response of NEURON model to second test pulse. The difference is due to modulation of KCa by the elevated MAPK activity.

stage we applied a test pulse of 1 nA for 250 ms to measure the firing properties of the neuron before potentiation. We then ran the NEURON model for 1 s to allow the model to settle. Similarly we loaded the GENESIS/Kinetikit model using the *loadG* command, and ran this simulation for 1800 s to settle.

2. In the Python wrapper class for each model, we defined a run method to advance the simulation in time. That for the NEURON model uses a *run* function we defined in the custom

hoc script. This run function calls NEURON's *advance* command to advance the simulation. In the wrapper class for the GENESIS/Kinetikit model the run method calls the *step* command to advance the simulation (**Figure 4B**).

3. We used the Python interface to read out somatic calcium levels from the NEURON model and insert them into the MOOSE model, and to feed back MAPK activity changes from the MOOSE model to modulate KCa conductances.

We wrote another higher-level function *run* to advance the coupled simulations using the two wrapper classes (not to be confused with the member method *run* of these classes). This function (1) creates instances of both wrappers, which involves initializing the models, (2) runs the NEURON simulation for 1 s, (3) reads out the calcium level, performs rescaling and updates the kinetic model with this value, (4) advances the kinetic simulation for 1 s to catch up with the electrical model, (5) reads out the activity level of MAPK from the GENESIS/Kinetikit model and modifies the $[Ca^{2+}]$ dependent K^+ channel conductances in the NEURON model in inverse proportion to this (**Figures 4E,F**).

Our simulated experiment is illustrated in **Figure 4F**. We loaded the models and allowed them to settle. We measured baseline neuronal responses at this stage using a 250-ms, 0.15 nA current pulse. Following this we used the *run* function for the further time-evolution of the system. We applied a strong LTP-inducing stimulus to the neuronal model for 7 s, and then allowed the simulation to continue for 183 s. Finally we repeated the 250 ms, 0.15 nA test for neuronal responses.

The time-evolution of membrane potential, Ca^{2+} levels, and MAPK activity are shown in **Figures 4D,E**. The initial and final burst waveforms of the neuron are shown in **Figures 4C,G**. We observe that the coupled model shows how electrical stimulation can lead to signaling events, with feedback effects on the electrical properties of the neuron. We should point out that this simulation is only a demonstration and the relationship between the chemical system and the biophysical properties of the neuron is over-simplified, although the two component models we used are realistic within their respective domains.

This example also illustrates the efficiency of using Python for data transfer when traffic volumes are small compared to the computational times. The neuronal calculations in NEURON took about 91% of the simulation run-time, the signaling calculations in MOOSE took ~8.5%, and the data transfer through Python accounted for only around 0.5%. As we discuss below, there may be other interface contexts where more efficient, low-level data transfer protocols may be needed, and the relatively facile Python interface may not be appropriate.

DISCUSSION

We have used PyMOOSE, the Python interface to MOOSE, to achieve interoperability at three levels. First, we used standard mathematical packages in Python to analyze MOOSE output. Second, we used the QT graphical toolkit from within Python to build a GUI for a MOOSE simulation. Third, we used Python as a glue language to run a cross-simulator model combining an electrophysiological model set up in NEURON with a biochemical signaling model set up in GENESIS/Kinetikit.

ISSUES WITH PYTHON INTEROPERABILITY

The strengths of the Python language make it perhaps too easy to repeat well-known mistakes in simulation development. We consider two such issues. First, Python is an interpreted language in most implementations. In the context of simulations, it is not meant for number crunching. Well-designed libraries like NumPy can hide some of these limitations from the user, and fast hardware can conceal other inefficiencies. However, given the same specialized

algorithms, a compiled language will perform better than an interpreted one. Therefore, for large simulations, we need to combine the best possible algorithms with optimized and compiled languages. MOOSE has as one of its goals the capability of managing the low-level, high-traffic flow of data between different numerical engines incorporated into MOOSE. We do not consider Python appropriate for such operations. Second, many aspects of model specification should be done using declarative rather than procedural approaches (Cannon et al., 2007; Crook et al., 2005, 2007). However, Python makes procedural model definition very easy, and may even provide a certain level of interoperability if several simulators provide equivalent calls for model setup. For example, there are some impressive recent efforts to develop a standard vocabulary for network definitions across simulators (<http://neuralensemble.org/trac/PyNN/>; this issue). While the presence of Python as a common link language may temporarily address the interoperability issues of this approach, we feel that it would be a cleaner design to use a separate, declarative definition for networks such as NeuroML (<http://neuroml.org>). Nevertheless, we completely agree that a standard vocabulary for model definitions is an important first step toward this goal.

MODEL SPECIFICATION VS. SIMULATOR CONTROL

Model specification and exchange issues have been ably addressed by the communities developing model specification languages (Le Novère et al., 2005; Qi and Crook, 2004; <http://neuroml.org>; <http://sbml.org>). The current paper focuses on the second problem, that of making it easier for researchers to control and set up these diverse simulation tools. We have shown how this can be done with the simulator MOOSE, using Python as a glue language. Run-time communication between simulators has previously been achieved using the NEOSIM framework, which uses Java (Goddard et al., 2001; Howell et al., 2002). More recently, the MUSIC framework specifies an API for simulators to use to communicate with each other (Ekeberg and Djurfeldt, 2008). Our study is novel in two respects. First, we use the built-in Python capabilities of two simulators to achieve run-time communication, without the need to modify either simulator or to build an additional framework for communication. Second, we carry out bidirectional communications across scales (biophysical to biochemical models) and involving continuous data types (channel conductance and calcium concentrations) rather than spike events.

The evolution of neuronal simulator technology has seen a gradual separation of different aspects of modeling, with a corresponding improvement in interoperability. The first step was to develop higher-level simulation tools (e.g., NEURON and GENESIS) to separate the numerical and housekeeping code from the model-specific code. This let people share models, provided they were written for the same simulator. The second was the development of declarative model specifications that were separate from the simulator. This initially took the form of semi-declarative cell morphology files (NEURON 'geom' files and GENESIS 'p' files), which required additional files for channel specification. This process of separation of model definition from simulator control has continued. The Neuroconstruct suite refines the declarative definition of models, with NeuroML and ChannelML as declarative definitions sufficient for most single-neuron models. Importantly,

at this level quite different simulators can use the same original model definition to run simulations. A third stage is the convergence of different simulators to use the same link language, in this case Python. This makes it possible to explicitly separate model definition from simulator control. In the current paper, we have illustrated this with a composite signaling-neuronal model drawing on NEURON and MOOSE. We have utilized two legacy models, one written for NEURON, and one written for GENESIS. Even though the legacy models themselves were not entirely set up in a declarative manner, we used the original model definitions only to load in the model specifications. We used Python as the procedural language to control these operations, and to mediate communication between the models at run-time.

SUSTAINABILITY OF PYTHON INTEROPERABILITY

Simulator interoperability has long been regarded as important (Crook et al., 2005, 2007; Goddard et al., 2001). Such projects have been difficult to execute, and still harder to maintain, because they

depend on multiple underlying simulator projects, each with different APIs, directions and life-cycles. Python is a potential way out of this problem. First, Python itself is a well-established language with a strong community and support. Second, the issues of interfacing to Python are now being undertaken by individual simulator development teams. Interoperability emerges from these independent efforts rather than requiring a separate project to achieve coordination. Third, PyMOOSE itself will be maintained for the long-term, since Python will be the default scripting language for MOOSE. We suggest that long-term improvements in interoperability will be driven both by widespread simulator support for declarative model specifications, and by a richer ecosystem of simulators fluent in Python.

APPENDIX

Program listing: ca3_db.hoc provides the functions to load and initialize the NEURON CA3 cell model as well as for advancing the simulation for a specified interval and for updating parameters.

```

/*****
* Derived from Hippocampal CA3 pyramidal neuron model from the paper
* M. Migliore, E. Cook, D.B. Jaffe, D.A. Turner and D. Johnston, Computer
* simulations of morphologically reconstructed CA3 hippocampal neurons, J.
* Neurophysiol. 73, 1157-1168 (1995).
* The original model is available in modeldb: accession no: 3263
* http://senselab.med.yale.edu/ModelDb/ShowModel.asp?model=3263
*
* Modified by: Subhasis Ray , 2008
*****/
objref ccode, vecCai, vecT, vecV, outFile, stim1, stim2, stim3, fih

vecV = new Vector()
vecCai = new Vector()
vecT = new Vector()
outFile = new File()
ccode = new CCode(0)
ccode.active(1)
ccode.atol(1e-3)
START = 2
AMP = 1.0
// ***** NEURON A *****

FARADAY=96520
PI=3.14159
secondorder=2
dt=0.025
celsius=30
flag1=0

xopen("ca3a.geo")

proc conductances() {
    forall {
        insert pas e_pas=-65 g_pas=1/60000 Ra=200
        insert cadifus
        insert cal gcalbar_cal=0.0025
        insert can gcanbar_can=0.0025
    }
}

```

```

        insert cat   gcatbar_cat=0.00025
        insert kahp  gkahpbar_kahp=0.0004
        insert cagk  gkbar_cagk=0.00055
    }

    soma {
        insert nahh    gnabar_nahh=gna
        insert borgkdr gkdrbar_borgkdr=gkdr
        insert borgka  gkabar_borgka=gka
        insert borgkm  gkmbar_borgkm=gkm
    }

    for i=0,1 dend2[i] {
        insert nahh    gnabar_nahh=gna
        insert borgkdr gkdrbar_borgkdr=gkdr
        insert borgka  gkabar_borgka=gka
        insert borgkm  gkmbar_borgkm=gkm
    }

    for i=0,2 dend3[i] {
        insert nahh    gnabar_nahh=gna
        insert borgkdr gkdrbar_borgkdr=gkdr
        insert borgka  gkabar_borgka=gka
        insert borgkm  gkmbar_borgkm=gkm
    }

    for i=37,38 dend3[i] {
        insert nahh    gnabar_nahh=gna
        insert borgkdr gkdrbar_borgkdr=gkdr
        insert borgka  gkabar_borgka=gka
        insert borgkm  gkmbar_borgkm=gkm
    }
}

proc init() {
    t=0
    coord_cadifus()
    forall {
        cao=2
        cai=50.e-6
        ek=-91
        v=-65
        if (ismembrane("nahh")) {ena=50}
    }
    vecV.record(&soma.v(0.5))
    vecCai.record(&soma.cai(0.5))
    vecT.record(&t)

    finitialize(v)
    fcurrent()

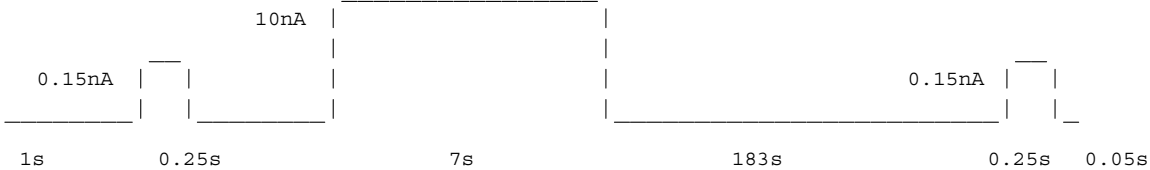
    forall {
        if (ismembrane("nahh")) {e_pas=v+(ina+ik+ica)/g_pas} else {e_pas=v+(ik+ica)/g_pas}
    }
    cvode.re_init()
}

```

```

proc setup(){
    strength = 1.0    /*namps*/
    tstim = 50
    tstop=500
    gna=0.015
    gkdr=0.03
    gka=0.001
    gkm=0.0001
    conductances()
    /* The schedule of experiment is as follows:

```



```

    The 1800 s runs with 1 s intervals interspersed with 1 s of
    kinetic simulation and update of gkbar for all ca dependent k
    channels.
    The genesis model needs over 1 uM [Ca2+] for 10 s.
    */

    soma {
        // first test pulse
        stim1 = new IClamp(0.5)
        stim1.amp = 0.15
        stim1.del = 1000.0
        stim1.dur = 250
        // tetanus pulse
        stim2 = new IClamp(0.5)
        stim2.amp = 1.0
        stim2.del = 2250
        stim2.dur = 7e3
        // final test pulse
        stim3 = new IClamp(0.5)
        stim3.amp = 0.15
        stim3.del = 192.25e3
        stim3.dur = 250
    }
    init()
}

proc update_gkbar(){/* multiply all Ca2+ dependent K+ conductance by $1 */
    forall {
        gkahpbar_kahp = gkahpbar_kahp * $1
    }

    soma {
        print "soma gkdrbar before:", gkdrbar_borgkdr

        gkdrbar_borgkdr = gkdrbar_borgkdr * $1
        gkmbar_borgkm = gkmbar_borgkm * $1
        print "soma gkdrbar after", gkdrbar_borgkdr
    }
    for i=0,1 dend2[i] {
        gkdrbar_borgkdr = gkdrbar_borgkdr * $1
    }
}

```



```

        gkmba_borgkm = gkmba_borgkm * $1
    }
    for i=0,2 dend3[i] {
        gkdrba_borgkdr = gkdrba_borgkdr * $1
        gkmba_borgkm = gkmba_borgkm * $1
    }

    for i=37,38 dend3[i] {
        gkdrba_borgkdr = gkdrba_borgkdr * $1
        gkmba_borgkm = gkmba_borgkm * $1
    }
    fcurrent()
}

access soma
distance()
/* run for interval specified as argument# 1 */
proc run(){
    t_start = t
    while (t < (t_start + $1)){
//      print "run() - @t=", t
        fadvance()
    }
//      print "run(): t_start =", t_start, " current time =", t, "run interval =", $1
}

proc do_run(){
    setup()
    print "setup done. running 7.25s"
    run(12250)
    print "t = ", t, "ms. done running. dumping data in test_neuron1.dat"
    outFile.wopen("test_neuron1.dat")
    for ii = 0, vecT.size() - 1 {
        outFile.printf("%g %g %g\n", vecT.x(ii), (vecCai.x(ii) - 50e-6)*2e6,
vecV.x(ii)) // the original GUI plots this function of cai instead of absolute
value - unit is nM*2
    }
    outFile.close()
    print "done dumping. running for 5s with 0.5nA"
    run(5000)
    print "t = ", t, "ms. soma.Cai = ", soma.cai(0.5), ". now updating gkbar"
    update_gkbar(10.0)
    print "done updating. writing to file"
    outFile.wopen("test_neuron2.dat")
    for ii = 0, vecT.size() - 1 {
        outFile.printf("%g %g %g\n", vecT.x(ii), (vecCai.x(ii) - 50e-6)*2e6,
vecV.x(ii)) // the original GUI plots this function of cai instead of absolute
value - unit is nM*2
    }
    outFile.close()
    print "done dumping. now running the rest"
    run(1800300)
    print "t = ", t, "ms. done running. writing to file"
    outFile.wopen("test_neuron3.dat")
    for ii = 0, vecT.size() - 1 {
        outFile.printf("%g %g %g\n", vecT.x(ii), (vecCai.x(ii) - 50e-6)*2e6,

```

```

vecV.x(ii)) // the original GUI plots this function of cai instead of absolute
value - unit is nM*2
    }
    outFile.close()
}

```

Program listing 2: moosenrn.py – this program wraps the GENESIS model and the NEURON model and provides simulation control and data exchange between the two simulators.

```

#!/usr/bin/env python

# Author: Subhasis Ray

import sys
sys.path.append("/home/subha/lib/python2.5/site-packages")
sys.path.append("/home/subha/lib/python2.5/site-packages/neuron")
import pylab
import numpy

import neuron
import moose

class NeuronSim:
    """Wrapper class for the neuron simulation"""
    def __init__(self, fileName="ca3_db.hoc"):
        """Load the file specified by fileName"""
        self.hoc = neuron.h
        self.hoc.load_file(fileName)
        self.hoc.setup()

    def run(self, interval):
        """Simulate for interval time in second"""
        self.hoc.run(interval * 1e3) # neuron keeps time in milli second

    def cai(self):
        """Returns cai of in nM"""
        return self.hoc.soma(0.5).cai

    def cai_record(self):
        """Returns a tuple containing the array of time points and the array
of cai values at the corresponding points"""
        timeVec = numpy.array(neuron.h.vecT)
        caiVec = numpy.array(neuron.h.vecCai)
        return (timeVec, caiVec)

    def v_record(self):
        """Returns a tuple containing the array of time points and the array
of membrane potential values at the corresponding points"""
        timeVec = numpy.array(neuron.h.vecT)
        vmVec = numpy.array(neuron.h.vecV)
        return (timeVec, vmVec)

    def update_kconductance(self, factor):
        """Modify the k hchannel conductances in inverse proportion of mapk_star_conc"""
        self.hoc.update_gkbar(factor)
        self.hoc.fcurrent()

```

```

def saveplots(self, suffix):
    cai = "nrn_cai_" + str(suffix) + ".plot"
    vm = "nrn_vm_" + str(suffix) + ".plot"
    t_series, vm_series, = self.v_record()
    t_series, cai_series, = self.cai_record()
    numpy.savetxt(cai, cai_series)
    numpy.savetxt(vm, vm_series)
    numpy.savetxt("nrn_t_" + str(suffix) + ".plot", t_series)

class MooseSim:
    """Wrapper class for moose simulation"""
    volume_scale = 6e20 * 1.257e-16
    def __init__(self, fileName="acc79.g"):
        self.settle_time = 1800.0
        self.ctx = moose.PyMooseBase.getContext()
        self.t_table = []
        self.t = 0.0
        self.ctx.loadG(fileName)
        self.ca_input = moose.Molecule("/kinetics/Ca_input")
        self.mapk_star = moose.Molecule("/kinetics/MAPK*")
        self.pkc_active = moose.Molecule("/kinetics/PKC-active")
        self.pkc_active_table = moose.Table("/graphs/conc2/PKC-active.Co")
        self.pkc_ca_table = moose.Table("/graphs/conc1/PKC-Ca.Co")
        self.mapk_star_table = moose.Table("/moregraphs/conc3/MAPK*.Co")
        self.mapk_star_table.stepMode = 3
        self.mapk_star_table.connect("inputRequest", self.mapk_star, "conc")
        self.mapk_star_table.useClock(2)
        self.ca_input_table = moose.Table("/moregraphs/conc4/Ca_input.Co")
        self.ca_input_table.stepMode = 3
        self.ca_input_table.connect("inputRequest", self.ca_input, "conc")
        self.ca_input_table.useClock(2)
        self.ctx.reset()
        self.ctx.reset()

    def set_ca_input(self, ca_input):
        """Sets the conc. of Ca_input molecule"""
        print "set_ca_input: BEFORE: nInit =", self.ca_input.nInit, ", n =",
self.ca_input.n, ", setting to: ", ca_input * MooseSim.volume_scale
        self.ca_input.nInit = ca_input * MooseSim.volume_scale
        self.ca_input.n = ca_input * MooseSim.volume_scale
        print "set_ca_input: AFTER: nInit =", self.ca_input.nInit, ", n =",
self.ca_input.n

    def ca_input(self):
        """Returns scaled value of Ca_input conc."""
        return self.ca_input.conc

    def run(self, interval):
        """Run the simulation for interval time."""
        self.ctx.step(float(interval))
        # Now expand the list of time points to be plotted
        points = len(self.pkc_ca_table) - len(self.t_table)
        delta = interval * 1.0 / points
        for ii in range(points):
            self.t_table.append(self.t)
            self.t += delta

```

```

def pkc_ca_record(self):
    """Returns the time series for pkc_ca conc."""
    return (self._t_table, self.pkc_ca_table)

def pkc_active_record(self):
    """Returns time series for pkc_active conc."""
    return (self._t_table, self.pkc_active_table)

def mapk_star_conc(self):
    """Returns MAPK* conc. in uM"""
    return self.mapk_star.n / MooseSim.volume_scale

def mapk_star_record(self):
    """Returns time series for [MAPK*]"""
    return (self._t_table, self.mapk_star_table)

def saveplots(self, suffix):
    pkc_a = "mus_pkc_act_" + str(suffix) + ".plot"
    pkc_ca = "mus_pkc_ca_" + str(suffix) + ".plot"
    mapk_star = "mus_mapk_star_" + str(suffix) + ".plot"
    ca_input = "mus_ca_input_" + str(suffix) + ".plot"
    numpy.savetxt("mus_t_" + str(suffix) + ".plot", self._t_table)
    self.mapk_star_table.dumpFile(mapk_star)
    self.pkc_ca_table.dumpFile(pkc_ca)
    self.pkc_active_table.dumpFile(pkc_a)
    self.ca_input_table.dumpFile(ca_input)

def test_run(self):
    self.run(500)
    print "After 500 steps of uninitiated run: [MAPK*] =", self.mapk_star_conc()
    self.ca_input.nInit = 10 * MooseSim.volume_scale
    self.ca_input.n = 10 * MooseSim.volume_scale
    self.run(5)
    print "After another 5 s with 10uM ca input: [MAPK*] =", self.mapk_star_conc()
    self.ca_input.nInit = 0.08 * MooseSim.volume_scale
    self.ca_input.n = 0.08 * MooseSim.volume_scale
    self.run(500)
    print "finished run. going to plot"
    print "After another 500 s with 0.08 uM ca input: [MAPK*] =",
self.mapk_star_conc()
    pylab.plot(pylab.array(self._t_table),
               pylab.array(self.pkc_active_table),
               pylab.array(self._t_table),
               pylab.array(self.pkc_ca_table))
    pylab.show()

if __name__ == "__main__":
    mus = MooseSim()
    mus.set_ca_input(0.08)
    mus.run(1800.0)
    mus.saveplots("1")
    start_mapk = mus.mapk_star_conc()
    nrn = NeuronSim()
    nrn.run(2.25)
    nrn.saveplots("1")
    file_ = open("cai_setings.txt", "w")

```



```

# Interleaved execution of MOOSE and NEURON model
# Synchronizing after every 1 s of simulation
while nrn.hoc.t < 192.25e3
    scaled_cai = scale_nrncai(nrn.cai())
    mus.set_ca_input(scaled_cai)
    print "scaled_cai =", scaled_cai
    file_.write(str(nrn.cai()) + " " + str(scaled_cai) + "\n")
    mus.run(1.0)
    gkbar_scale = start_mapk / mus.mapk_star_conc()
    start_mapk = mus.mapk_star_conc()
    print "[mapk*] = ", start_mapk
    nrn.update_kconductance(gkbar_scale)
    nrn.run(1.0)
    print "time is ", nrn.hoc.t * 1e-3, "s"
file_.close()
nrn.saveplots("2")
mus.saveplots("2")
# final test pulse run
nrn.run(0.3)
nrn.saveplots("3")
t_series, vm_series, = nrn.v_record()
t_series, cai_series, = nrn.cai_record()
pylab.subplot(121)
pylab.plot(t_series, numpy.array(vm_series), t_series, numpy.array(cai_series)
* 1e6)
t_series, pkc_act, = mus.pkc_active_record()
t_series, pkc_ca, = mus.pkc_ca_record()
t_series, mapk_star, = mus.mapk_star_record()
pylab.subplot(122)
pylab.plot(numpy.array(t_series), numpy.array(pkc_act), numpy.array(t_series), numpy.array(pkc_ca),
numpy.array(t_series), numpy.array(mapk_star))
pylab.show()

```

ACKNOWLEDGEMENTS

The development of MOOSE is supported by grants from the Department of Biotechnology, India, and the NIGMS/Systems

Biology Center of New York. We acknowledge support from FACETS to S. Ray to attend the FACETS/CodeJam meeting at CNRS, Gif-sur-Yvette, which further stimulated PyMOOSE development.

REFERENCES

- Ajay, S. M., and Bhalla, U. S. (2004). A role for ERKII in synaptic pattern selectivity on the time-scale of minutes. *Eur. J. Neurosci.* 20, 2671–2680.
- Beazley, D. M. (1996). SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th Annual USENIX Tcl/Tk Workshop*, Monterey, CA.
- Beeman, D., and Bower, J. M. (2004). Simulator-independent representation of ionic conductance models with ChannelDB. *Neurocomputing* 58–60, 1085–1090.
- Bhalla, U. S. (1998). Advanced XODUS techniques. In *The Book of GENESIS: Exploring Realistic Neural Models with the General Neural Simulation System*, 2nd edn, J. M. Bower and D. Beeman, eds (New York, Springer).
- Bhalla, U. S., and Iyengar, R. (1999). Emergent properties of networks of biological signaling pathways. *Science* 283, 381–387.
- Bhalla, U. S., Ram, P. T., and Iyengar, R. (2002). Map kinase phosphatase as a locus of flexibility in a mitogen-activated protein kinase signaling network. *Science* 297, 1018–1023.
- Bower, J. M., and Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the General Neural Simulation System*, 2nd edn. New York, Springer.
- Bunow, B., Segev, I., and Fleshman, J. W. (1985). Modeling the electrical behavior of anatomically complex neurons using a network analysis program: excitable membrane. *Biol. Cybern.* 53, 41–56.
- 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.
- Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge, Cambridge University Press.
- Cornelis, H., and De Schutter, E. (2003). NeuroSpaces: separating modeling and simulation. *Neurocomputing* 52–54, 227–231.
- Crook, S., Beeman, D., Gleeson, P., and Howell, F. (2005). XML for model specification in neuroscience. In *Special Issue on Realistic Neuro Modeling – Wam-Bamm ‘05 Tutorials*. J. M. Bower and D. Beeman (eds.). *Brains Minds Media*, Vol. 1, bmm228 (urn: nbn:de:0009-3-2282). <http://www.brains-minds-media.org/archive/228>
- Crook, S., Gleeson, P., Howell, F., Svitak, J., and Silver, R. A. (2007). MorphML: level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics* 5, 96–104.
- Ekeberg, Ö., and Djurfeldt, M. (2008). MUSIC – multisimulation coordinator: request for comments. *Nature Proceedings*. Available at: <http://dx.doi.org/10.1038/npre.2008.1830.1>.
- Goddard, N., Hood, G., Howell, F., Hines, M., and De Schutter, E. (2001). NEOSIM: portable large-scale plug and play modelling. *Neurocomputing* 38–40, 1657–1661.
- Goddard, N., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modeling in neuroscience. *Philos. Trans. R. Soc. Lond., B, Biol. Sci.* 356, 1209–1228.

- Hines, M. (1993). NEURON – a program for simulation of nerve equations. In *Neural Systems: Analysis and Modeling*, F. Eeckman, ed. (Norwell, MA, Kluwer), pp. 127–136.
- Howell, F., Bazhenov, M., Rogister, P., Seznowski, T., and Goddard, N. (2002). Scaling a slow-wave sleep cortical network model using NEOSIM. *Neurocomputing* 44–46, 453–458.
- Hucka, M., et al. (2002). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531.
- Le Novère, N., Finney, A., Hucka, M., Bhalla, U. S., Campagne, F., Collado-Vides, J., Crampin, E. J., Halstead, M., Klipp, E., Mendes, P., Nielsen, P., Sauro, H., Shapiro, B., Snoep, J. L., Spence, H. D., and Wanner, B. L. (2005). Minimum information requested in the annotation of biochemical models (MIRIAM). *Nat. Biotechnol.* 23, 1509–1515.
- Maes, P. (1987). Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. Orlando, FL, ACM, pp. 147–155.
- Martelli, A., Ravenscroft, A. M., and Ascher, D. (2005). *Python Cookbook*, O'Reilly, p. 14.
- Migliore, M., Cook, E. P., Jaffe, D. B., Turner, D. A., and Johnston, D. (1995). Computer simulations of morphologically reconstructed CA3 hippocampal neurons. *J. Neurophysiol.* 73, 1157–1168.
- Qi, W., and Crook, S. M. (2004). Tools for neuroinformatic data exchange: an XML application for neuronal morphology data. *Neurocomputing* 58C–60C, 1091–1095.
- Segev, I., Fleshman, J. W., Miller, J. P., and Bunow, B. (1985). Modeling the electrical behavior of anatomically complex neurons using a network analysis program: passive membrane. *Biol. Cybern.* 53, 27–40.
- Smith, B. C. (1982). *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA.
- Conflict of Interest Statement:** The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 15 September 2008; paper pending published: 13 October 2008; accepted: 01 November 2008; published online: 19 December 2008.

Citation: Ray S and Bhalla US (2008) PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* (2008) 2: 6: xx–xx. doi: 10.3389/neuro.11.006.2008 Copyright © 2008 Ray and Bhalla. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



A Python analytical pipeline to identify prohormone precursors and predict prohormone cleavage sites

Bruce R. Southey^{1,2*}, Jonathan V. Sweedler¹ and Sandra L. Rodriguez-Zas²

¹ Department of Chemistry, University of Illinois, Urbana, IL, USA

² Department of Animal Sciences, University of Illinois, Urbana, IL, USA

Edited by:

Rolf Kötter, Radboud University
Nijmegen, The Netherlands

Reviewed by:

Yoonseong Park, Kansas State
University, USA
Niovi Santama, University of Cyprus
and Cyprus Institute of Neurology and
Genetics, Cyprus

* Correspondence:

Bruce Southey, Department of
Chemistry, University of Illinois,
1207W. Gregory Dr., Urbana, IL 61801,
USA. e-mail: southey@illinois.edu

Neuropeptides and hormones are signaling molecules that support cell–cell communication in the central nervous system. Experimentally characterizing neuropeptides requires significant efforts because of the complex and variable processing of prohormone precursor proteins into neuropeptides and hormones. We demonstrate the power and flexibility of the Python language to develop components of an bioinformatic analytical pipeline to identify precursors from genomic data and to predict cleavage as these precursors are en route to the final bioactive peptides. We identified 75 precursors in the rhesus genome, predicted cleavage sites using support vector machines and compared the rhesus predictions to putative assignments based on homology to human sequences. The correct classification rate of cleavage using the support vector machines was over 97% for both human and rhesus data sets. The functionality of Python has been important to develop and maintain NeuroPred (<http://neuroproteomics.scs.uiuc.edu/neuropred.html>), a user-centered web application for the neuroscience community that provides cleavage site prediction from a wide range of models, precision and accuracy statistics, post-translational modifications, and the molecular mass of potential peptides. The combined results illustrate the suitability of the Python language to implement an all-inclusive bioinformatics approach to predict neuropeptides that encompasses a large number of interdependent steps, from scanning genomes for precursor genes to identification of potential bioactive neuropeptides.

Keywords: Python, bioinformatics, neuropeptides, machine learning, support vector machine, precursor cleavage, rhesus monkey

INTRODUCTION

Neuropeptides are a class of cell–cell peptides that can act as neurotransmitters and hormones and have various paracrine, endocrine, and autocrine effects (Boutrel, 2008; Heinrichs and Domes, 2008). Neuropeptides directly influence a diverse set of biological processes from growth and development to learning. For example, oxytocin is known as a mammalian hormone associated with reproduction but also is a neurotransmitter that has been associated with social behavior traits including trust, autism, inhibition of tolerance to additive drugs and impaired learning and memory functions. Furthermore, oxytocin and arginine vasopressin are intermediaries of social behaviors, including attachment, social cognition and stress, anxiety, and aggression (Heinrichs and Domes, 2008).

Experimental detection of neuropeptides in mammals has been limited to a few species (primarily human, mouse and rat) or the characterization of selected peptide families (such as insulin) across greater numbers of species. This lack of experimental characterization is predominantly because such experimental procedures are resource intense and the presence of neuropeptides varies with species, tissue, developmental stage and even organism state. Genomic sequencing provides the opportunity to discover neuropeptides in other species with limited or no experimental studies on neuropeptides. For example, while the rhesus macaque monkey (*Macaca mulatta*) is widely used as model organism and its genome has been sequenced (Rhesus Macaque Genome Sequencing and

Analysis Consortium, 2007), only four rhesus prohormone genes have been reported compared to over 90 human prohormone genes. Consequently, accurate bioinformatic identification of neuropeptide genes and characterization of precursors is essential in rhesus neuroscience research.

Several factors make annotating prohormones and their associated peptides difficult. First, neuropeptides result from a complex series of post-translational modifications (PTMs) of precursor proteins. Second, the conserved “bioactive” peptide sequence that interacts with its cognate receptor can be short, only a few amino acids long, with large sections of diverse sequences in the prohormone. Thus, homology to well-studied species is not enough to offer accurate neuropeptide predictions across species.

The typical structure of neuropeptide precursor after translation includes a signal peptide region and a region that contains one or more neuropeptides (Fricker, 2005; Hook et al., 2008). After translation, the signal peptide is removed by signal peptidases and the remaining peptide is cleaved by other proteases (notably proprotein or prohormone proteases) that cleave the sequence at basic (Arg or Lys) sites (Fricker, 2005; Hook, 2006; Hook et al., 2008). After cleavage, the N-terminal basic amino acids are typically removed by carboxylases and various additional PTMs such as amidation and glycosylation can occur (Fricker, 2005; Hook et al., 2008).

We address these points here with a bioinformatics toolkit to discover and characterize neuropeptides. Essential components

of this analytical pipeline are the computational identification of precursor genes in nucleic databases and model-based prediction of cleavage and other PTMs of the precursors. Python is an ideal language to develop this analytical pipeline for the discovery and characterization neuropeptides. The core language has easy to use functions that facilitate complex manipulation of information and integration of results from the multistep analytical pathway. The suitability of Python is further strengthened by third-party modules such as BioPython (<http://biopython.org>) for bioinformatics (Bassi, 2007) and Numerical and Scientific Python (<http://www.scipy.org>) for numerical computation (Oliphant, 2007). The combination of all these features in a single language makes Python an ideal choice for bioinformatic applications (Bassi, 2007; Kinser, 2008). In terms of a pipeline, the power and flexibility of Python can be used for the full pipeline or to integrate different components of the pipeline together. We illustrate the use of Python to implement an analytical pipeline that integrates vastly different components necessary to identify rhesus neuropeptides and associated precursors.

PRECURSOR IDENTIFICATION USING BIOINFORMATICS RESOURCES

An exhaustive survey of neuropeptide precursors in a genome is the first step in the complete characterization of the neuropeptidome of a species. The development of bioinformatic analytical pipeline to discover neuropeptide precursors requires the integration of multiple steps involving multiple tools. Bioinformatic tools including sequence homology search using BLAST (Altschul et al., 1997) and multiple sequence alignment using T-Coffee (Notredame et al., 2000) are available as standalone packages or via a web interface. BioPython provides an integrated environment that supports different aspects of bioinformatics including parsing results from bioinformatics tools. For example, Bassi (2007) illustrates the use of BLAST with BioPython.

In the first step of the prohormone analytical pipeline, rhesus precursors were identified based on precursor information from other mammalian species with extensive neuropeptide research. In particular, a list of human precursors and neuropeptide sequences was collected from the UniProt Knowledgebase database (The UniProt Consortium, 2008) and literature review (Amare et al., 2006; Tegge et al., 2008). The set of human precursors was queried against the database of predicted proteins derived from the rhesus genome (http://www.ncbi.nlm.nih.gov/projects/genome/guide/rhesus_macaque/) using a standalone version of BLAST (version 2.2.18) using the default parameter settings (e.g., expectation value of 10 and Blosom62 scoring matrix) except for disabling the filter option. Queries were conducted using the complete precursor sequence that included the regions that contain the signal peptide and neuropeptides to maximize the detection of the rhesus precursor. Human precursors were used because of the evolutionary relationships between the rhesus and human species and the completeness of the list in humans. Information from other species (e.g. mouse and rat) can also be used to evaluate the accuracy of the search process. The repetitive process of searching for each human precursor on the rhesus database was implemented by exploiting the ability of BLAST to handle multiple sequences and using Python to parse results. The query input file containing all human precursors was submitted to BLAST and the output was saved in an XML

formatted file. An XML format provides structured information in a machine readable format that permits repeated access.

The XML file of BLAST results can be also be parsed directly using standard Python libraries such as the elementtree library to extract the results for each of the human precursors. The script in **Listing 1** opens the specified XML file and recursively stores the contents in a Python class that contains the attributes and values specified by the XML document type definitions used by BLAST. After parsing the BLAST XML file, the script loops across the query sequences and displays the match and the score and e-value of the best match to the query sequence. Using a Python script allows greater control of the output including extracting precursors with the highest scoring BLAST hits, precursors with no hits, all hits that exceed a threshold determined by the user, or all hits. Furthermore, Python provides sufficient flexibility to identify the common scenarios with comparative genome analyses where multiple precursors match the same target or the same precursor matches different targets with similar scores.

The complete identification of precursors can require different levels of user input especially related to species divergence. The difficulties imposed by species divergence and available resources can be investigated by evaluating different BLAST specifications (e.g. selection of database, scoring matrices, E-value threshold), different genomic resources (e.g. unassembled sequences) and information from species when this is available. Due to the repetitive nature of these investigations, Python can be used to facilitate the rapid evaluation of the different specifications and combining the information for user assessment.

Although low E-values constitute statistical evidence that supports the detection of homologous sequences between species, false matches and partial matches are possible. The accuracy of the identification of predicted rhesus precursors was accessed by aligning the sequence to corresponding sequences from multiple other species using multiple sequence alignment tools such as T-Coffee. Most multiple sequence alignment tools only perform a single alignment so that it is necessary to perform one alignment for every precursor. Simple Python scripts can be used for the repetitive creation of sequence files including multiple sequences across species for each precursor and subsequent alignment for each precursor. The resulting alignments were then viewed to identify which rhesus precursor predictions are reliable or contain the prediction but are too long (the result of automated predictions and sequencing or assembly errors) or incomplete (due to incomplete coverage of the particular genomic region, sequencing or assembly errors). Based on the final alignments, 67 rhesus neuropeptides precursors were identified solely in the rhesus database of predicted proteins.

Identification of precursors using protein predictions and automated tools is fast and effective. However, this approach misses precursors that are partially predicted or not predicted due to sequencing or assembly issues. In order to identify if a human precursor is present in the genome of the rhesus monkey, the protein sequences of the precursors are queried against the nucleotide sequences from the genome assembly. The result of the BLAST query only provides the locations that sufficiently match the protein sequence and consequently ignore low scoring and intronic regions. The full precursor sequence can be extracted using Wise2 (<http://www.ebi.ac.uk/Tools/Wise2/index.html>; Birney et al., 2004). Wise2


```

import sys
from xml.etree.ElementTree import ElementTree

class blastparms:
    pass

def getbranch(trunk):
    twig=blastparms()
    for branch in trunk:
        leaves=branch.getchildren()
        if len(leaves) > 0:
            bud=[]
            for leaf in leaves:
                bud.append(getbranch(leaf))
            twig.__dict__.update({branch.tag: bud})
        else:
            twig.__dict__.update({branch.tag:
branch.text})
    return twig

tree=ElementTree(file=sys.argv[1])
root=tree.getroot()
branches=getbranch(root.getchildren())

for niter, branch_hit in enumerate(branches.BlastOutput_iterations):
    query=branch_hit.__dict__['Iteration_query-def']
    for index, ihit in
enumerate(branches.BlastOutput_iterations[niter].Iteration_hits):
        print '%s,%i,%s,%s,%s,%s' % (query, index,
ihit.Hit_id, ihit.Hit_def.replace(' ','_'),
ihit.Hit_hsp[0].Hsp_score, ihit.Hit_hsp[0].Hsp_evalue)

```

LISTING 1 | Parsing an BLAST XML file in Python.

predicts the gene structure by comparing a protein sequence to a genomic DNA sequence and using a gene prediction model that allows for introns and frameshift errors. The genomic sequence required by Wise2 was obtained by using Python to read the genomic DNA sequence of the assembled rhesus genome, identify and extract the relevant chromosomal region and perform the reverse transcription into the complementary strand if necessary. If the extracted region is insufficient to accurately identify the main gene structure components, the genomic region can be expanded and resubmitted to Wise2. An additional advantage of combining the BLAST and Wise2 tools is that the protein sequence, mRNA sequence and the location of the exons are simultaneously available and can be used to confirm the accuracy of the predictions.

The combined strategy of using BLAST and Wise2 directly identified eight additional precursors that were not been previously predicted and provide valuable information for the manual annotation of rhesus precursors. For example, the rhesus CCKN precursor was identified on chromosome 2 but lacked a match to last 28 amino acids of the human CCKN sequence. Examination the genomic sequence showed that a region of 91 unknown bases occurred immediately after the last residue of the mRNA sequence predicted using Wise2. This nucleotide segment most likely codes the missing precursor sequence that corresponded to the last exon of the human precursor gene and was missed in the assembly. The search for the missing region among the rhesus trace archives (a collection of raw sequence traces, http://www.ncbi.nlm.nih.gov/projects/genome/guide/rhesus_macaque/), uncovered a hit to a contig that contained the missing segment and resulted in the

prediction of a complete CCKN precursor. A different scenario was encountered with the NPS precursor because the Wise2 prediction missed the start of the NPS precursor. This failure was most likely due to the structure of the human gene where the first exon only codes for two amino acids. Consequently, the corresponding rhesus exon was identified by a query using the complete human NPS nucleotide and combined with the Wise2 prediction to obtain the complete rhesus NPS precursor.

There were also 17 precursors that could not be recovered solely based on the assembly alone without further examining the trace archives for unassembled or incorrectly assembled contigs. For example, the related crab-eating macaque (*M. fascicularis*) insulin (INS) precursor has been reported (Wetekam et al., 1982) and, thus, is expected to be found in the rhesus genome. Queries of the human and *M. fascicularis* INS sequence on the *M. mulatta* genome did not permit full recovery of the rhesus INS precursor due to gaps and a stop codon in the genomic assembly. The results from a search of the trace archives indicated that the inclusion of different contig (ti|523766964) would most likely result in the identification of the complete rhesus INS precursor.

The individual precursors undergo a number of additional processing steps before the final bioactive peptides are created. Thus, once the list of precursor protein sequences has been compiled, expected prohormone structural features such as a signal peptide and prohormone cleavage sites are identified for each individual precursor. The signal peptide was predicted using SignalP (Bendtsen et al., 2004) and the length of the signal peptide was recorded with the sequence. The rhesus precursors lack experimental cleavage

information so cleavage sites must be assigned based on homology to other animals or cleavage models. The reliability of the homology-based prediction of cleavage relies on the degree of conservation of the precursor between species available.

Human data were expected to provide the most accurate assignment of cleavage data due to the close evolutionary relationship between the human and rhesus species. Python scripts were developed to assign precursor cleavage information based on homology to human sequences. The human and rhesus sequences of each precursor were first aligned using T-Coffee. The locations of the human cleavage sites were then found in the corresponding aligned rhesus sequence. Finally the rhesus sequence and cleavage data was obtained after removing any gaps that had been entered during the sequence alignment. Assuming that the precursor cleavage assignment based on human information provides a perfect characterization of precursor processing in the rhesus, then the comparison of model-based cleavage predictions and confirmed or homology-based cleavage information will provide the number of true and false positives (cleavage sites) and true and false negatives (non-cleavage sites). These results can be used to construct further indicators of cleavage model performance including correct classification rate (ratio of true versus true and false results), sensitivity (ratio of true positives versus all positives), specificity (ratio of true negatives versus all negatives), positive and negative precision (Southey et al., 2006a).

CLEAVAGE PREDICTION USING MACHINE LEARNING TECHNIQUES

Prediction of the cleavage sites within the precursor is essential for identification of the final peptides produced by the prohormones, including the neuropeptides. Previously we have shown that machine learning techniques including logistic regression, artificial neural networks and memory-based reasoning are successful in predicting cleavage sites in neuropeptide precursors in diverse sets of species (Amare et al., 2006; Hummon et al., 2003; Southey et al., 2008; Tegge et al., 2008). An analytical pipeline to predict cleavage using machine learning involves preparing and processing the sequence and cleavage data, training and testing of prediction models using machine learning techniques to identify the most appropriate model, predict the possible peptides using the most appropriate model and any PTMs present in the predicted peptides.

Python can be used to process the sequence and cleavage data into a generic file that can be used by a single application as well by different applications following the steps outlined by Southey et al. (2008). Generally these steps involve: (1) reading the sequence and cleavage data, (2) removing the signal peptide, (3) splitting the remaining sequence into overlapping windows, (4) assigning cleavage status to the window and (5) recoding the amino acids as binary indicators with respect to the actual location within the window. The script in **Listing 2** demonstrates how a single neuropeptides sequence with length of signal peptide and cleavage site is processed. First the signal peptide is removed and the resulting sequence is padded to permit windows that may extend past the ends of the sequence. The sequence is then split into overlapping windows and windows with basic amino acids (Lys and Arg) are kept. The amino acids within each window are then recoded with

dummy values and cleavage status is assigned. The resulting location within the complete precursor sequence, the window of the sequence, cleavage status of the window and coding of the amino acids is then displayed.

The resulting generic file can be used as input to a stand-alone machine learning package or tool (e.g. R <http://www.r-project.org>), or by a tool directly implemented in Python (e.g. the SciKit learn <http://www.scipy.org/scipy/scikits/wiki/MachineLearning>), or automatically passed to a stand-alone tool using a Python interface and language bindings. This latter strategy will be illustrated using the Python bindings provided with the LibSVM package (Chang and Lin, 2001) that implement training and cross-validation of support vector machines in Python. The general use of LibSVM involves the input of data, selection of a support vector machine and associated parameter, training of the support vector machine given the data and parameters and evaluation of trained support vector machine. Following Salzberg (1997), the optimal parameters for the support vector machine were identified using cross-validation and a grid search across the parameters of the support vector machine. Preliminary results indicated that the default support vector machine with a radial basis function provided the same performance as other types and had the advantage of only requiring two parameters. The LibSVM also provides k-fold cross-validation where the training data was split into k components of which k – 1 components was used to train a model and the last component was used for testing. The cross-validation approach was repeated such that all data components were used as testing and the overall cleavage miss-classification rate across complete data is obtained.

A Python script was used process generic file previously obtained from the human and rhesus sequence and cleavage data into human and rhesus data sets in the format required by the LibSVM. Part of the script (**Listing 3**) loops across the two parameters of a support vector machine with a radial basis function (gamma and C) and within the loop calls the LibSVM cross-validation routine with the parameters of the support vector machine and supplied degree of cross-validation. This script also trains the support vector machine for the supplied parameters on the full training data set and computes the accuracy of this support vector machine on the test and training data sets. This script can be easily extended to evaluate multiple support vector machine specifications including linear and polynomial. In addition to the cross-validation, the script also trained a support vector machine on the full test data set for the supplied parameter values and tested the resulting support vector machine on the full test data and the training data. For data sets where the cross-validation and full data set support vector machine analyses for each combination of parameters becomes prohibitive, the script can be modified such that the support vector machine analysis of the full data set is only executed after the parameter values that provide the lowest miss-classification rate have been identified in a prior cross-validation step.

The parsing of the results from the Python script that trained and tested the support vector machine models offered insights into the similarities between the human and rhesus cleavage patterns. The rhesus and human cleavage prediction models selected had the highest 5-fold cross-validation accuracy and the fewest prediction errors in the training data. The evaluation of the parameters

```
# AA letters: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y
AAdummy={ 'A': '1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'C': '0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'D': '0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'E': '0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'F': '0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'G': '0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0',
            'H': '0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0',
            'I': '0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0',
            'K': '0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0',
            'L': '0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0',
            'M': '0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0',
            'N': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0',
            'P': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0',
            'Q': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0',
            'R': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0',
            'S': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0',
            'T': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1',
            'V': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1',
            'W': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1',
            'Y': '0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1' }

def codesequenc(psequence, psignal, pcleaved, left_win=9, right_win=9):
    sequence='x'*left_win+psequence[psignal:]+ 'x'*right_win
    for iaa, csite in enumerate(sequence):
        if csite in ['K','R']:
            location=str(iaa+psignal-left_win+1)
            coded=[]
            wcleaved='0'
            seq_frag=sequence[(iaa-left_win+1):(iaa+right_win+1)]
            for aa in seq_frag:
                coded.append(AAdummy.get(aa,
'0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0'))
            if location in pcleaved:
                wcleaved='1'
            print '%s,%s,%s,%s' % (location, seq_frag, wcleaved,
','.join(coded))

codesequenc('MVLVRRPWPALATVILALLVCLGALVDAYPIKPEAPGEDASPEELSRYYASLRHYLNL
VTRQRYGKRDPDALLSKTFPDGNGRPIRSRSEVPDLW',28, ['67']) # pyy_rhesus
```

LISTING 2 | Python script to recode an amino acid sequence into generic format for machine learning applications.

on the full data sets was also important because support vector machines with similar cross-validation correct classification rates had lower performance on the full and test data sets. For example, in the human support vector machine, the two highest scoring human support vector machines had correct classification rates of 91.0% and 90.6% after cross-validation. However, the highest scoring human support vector machine had correct classification rates of 99.9% and 99.6% in the human full data set and rhesus full data set, respectively. Whereas the second scoring human support vector machine had an approximately 3% lower correct classification rate in human full data set, and rhesus full data set (97.3% and 96.4%, respectively).

The performance of the support vector machine models was compared to the mammalian logistic regression model (Amare et al., 2006), the human logistic regression and human artificial neural models (Tegge et al., 2008) and the empirical Known Motif model Southey et al. (2006b). On the human data set, the human support vector machine had the highest correct classification rate (99.9%), as expected, followed by the rhesus support vector machine (97.9%), human artificial neural model (92.2%), human logistic regression (90.2%), mammalian logistic regression (82.5%)

and finally the Known Motif model (76.6%). The rhesus support vector machine provided perfect classification on the rhesus data set followed by the human support vector machine (99.6%), human artificial neural model (91.3%), human logistic regression (89.6%), mammalian logistic regression (82.4%) and finally the Known Motif model (76.7%). Models trained on human data had better prediction than general mammalian model or empirical known motif model. This result was expected independently of evolutionary relationships because the human cleavage data was used to assign cleavage in the rhesus.

The main reason for the different model performance was the lower number of false positive predictions by the support vector machines relative to the other methodologies. The rhesus support vector machine had slightly lower number of false negative predictions in the human data set than the human artificial neural network. The differences between the different prediction approaches are due to differences in the data sets used to train and test the models and the ability of the methodologies to accommodate linear and non-linear relationships between the input variables and cleavage patterns. Tegge et al. (2008) used 62 human precursors to train artificial neural network and logistic regression models,

```

import svm

def cross_validate(prob_x, prob_y, param, nr_fold):
    "Do cross validation for a given SVM problem."
    train_prob= svm.svm_problem(prob_y, prob_x)
    total_correct = 0
    pred_y = svm.cross_validation(train_prob, param, nr_fold)
    for (yin,ypred) in zip(prob_y, pred_y):
        if ypred == yin: total_correct += 1
    train_model = svm.svm_model(train_prob, param)
    print 'CorrectClassRate for %i-Fold cross-validation =%s' % (nr_fold,
(100.0 * total_correct / (len(prob_y))))
    return train_model

def model_accuracy(Pmodel, Ptext, Py, Px):
    TP= TN= FP=FN=0.0
    for (t,x) in zip(Py, Px):
        p=Pmodel.predict(x)
        if (t == -1 and p == -1.0): TN += 1
        elif (t == -1 and p == 1.0): FP += 1
        elif (t == 1 and p == 1.0): TP += 1
        elif (t == 1 and p == -1.0): FN += 1
    CorrectClass=((TP+TN)/(TP+FP+TN+FN))
    if (TP+FN) > 0: Sens=TP/(TP+FN)
    if (TN+FP) > 0: Spec=TN/(TN+FP)
    print 'Model Accuracy Statistics for %s: CorrectClassRate=%f,
Sensitivity=%f, Specificity=%f' % (Ptext, CorrectClass*100, Sens*100, Spec*100)
    return TP, FP, TN, FN

def svm_evaluate(Ytrain, Xtrain, Ytest, Xtest, type_svm, kernel_svm, gamma_svm,
C_svm, nfold):
    train_problem=svm.svm_problem(Ytrain, Xtrain)
    test_problem=svm.svm_problem(Ytest, Xtest)
    train_param = svm.svm_parameter(svm_type=type_svm, kernel_type =
kernel_svm, gamma=gamma_svm, C=C_svm)
    train_model= cross_validate(Xtrain, Ytrain, train_param, nfold)
    model_accuracy(train_model, 'Training', Ytrain, Xtrain)
    model_accuracy(train_model, 'Testing', Ytest, Xtest)

```

LISTING 3 | Python script for training and testing a support vector machine.

Amare et al. (2006) used 39 mammalian precursors to train logistic models and the human support vector machine model developed in this study were trained on 93 human precursors. The artificial neural network had perfect (100%) classification on the human data set reported in Tegge et al. (2008). The lower correct classification rate result by including more human precursors indicating that the human data set used by Tegge et al. (2008) likely does not contain complete information on cleavage that was used in training the support vector machines.

Across species, the impact of the precursor sequences used to train and test in the model performance can be assessed by comparing the performance of the same model across species. Comparison of the data used to train the support vector machines showed that all rhesus precursors had homologous in the human data set but 20 human precursors were not present on the rhesus data set. Of the 37 sites that received different cleavage classification by the two support vector machines, only 10 sites corresponded to precursors that were present in both species data sets; meanwhile the remaining sites were only present in the human precursor data set. Among the sites with differential cleavage prediction between species, four sites pertained to rhesus sequences that have different amino acids than the human sequence and these amino acids

have a strong association with cleavage patterns. For example, the INSL4 precursor in the rhesus includes a window with the amino acid sequence 'GCGPRFGKR↓MLSYCPMPE' where ↓ denoted the predicted cleavage site. However, this site was assigned a non-cleavage observed value because the homologous human window, 'GCGPRFGKHLLSYCPMPE', has not reported to be cleaved. Similarly, Southey et al. (2006b) reported a single amino acid difference between human and chimpanzee RFRP precursor that resulted in a false positive prediction in the chimpanzee sequence. These results demonstrate the value of bioinformatic prediction of precursor cleavage, especially in species with limited experimental confirmation. One important use of across species predictions is to eliminate false positive results from experimental consideration. As another use, this same information can also identify potentially species-specific cleavage sites to explain peptides that are unexpected based on homology alone.

APPLICATION/TOOL TO ASSIST IN THE IDENTIFICATION OF NEUROPEPTIDES

The prediction of cleavage sites in a protein sequence requires that the sequence must be processed into a usable format, then the prediction model is applied and finally the actual prediction

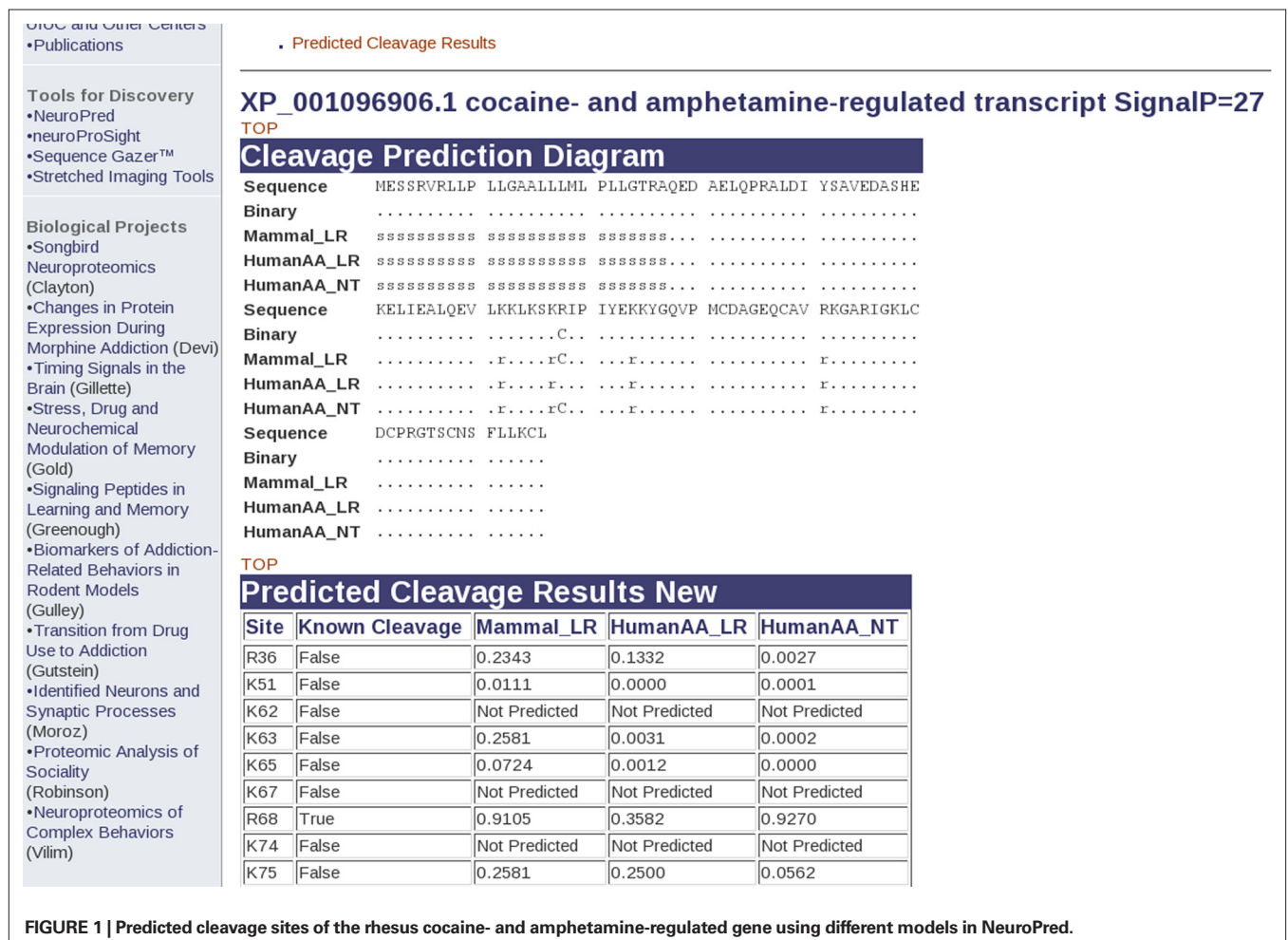
are returned. Each of these steps requires specialized knowledge ranging from processing the sequence to technical knowledge of applying the models derived from machine learning models. Developing a web application is one approach to remove specialized knowledge because a web form can be provided where the underlying script is responsible to convert the input into required format, apply the prediction models and display resulting predictions.

We developed NeuroPred (<http://neuroproteomics.scs.uiuc.edu/neuropred.html>), a web application in Python, to support the detection and characterization of the neuropeptidome (Southey et al., 2006a). The user requires only a sequence basic knowledge of neuropeptides and there is no requirement for specialized knowledge of areas such as Python or machine learning. Using a simple form, users can enter one or more protein sequences and then select one or more prediction models and different options that control the subsequent processing of the resulting peptides and output. A user can select either simple options where most options have been preselected for the user or more advanced options that provide all possible models and control of the input and output. NeuroPred validates all the inputs, predict cleavage sites for all sequences entered and models selected. Under the default options, NeuroPred

will display a cleavage prediction diagram indicating the predicted cleavage locations and optionally the probabilities of cleavage for the sequences entered and model selected (**Figure 1**).

To assist in the experimental studies using mass spectrometry (e.g., Hummon et al., 2005; Li and Sweedler, 2008), NeuroPred also computes the predicted mass of peptides including most of the known neuropeptide PTMs. The computation of the mass of the predicted peptides that can be used in high throughput mass spectrometry studies to assist in the identification of peptides. Depending on the options selected, NeuroPred will list the different peptides possible, the source for cleavage for the peptide (such as signal peptidase or prediction from one or more models), PTMs applied to the resulting peptides, predicted mass and full peptide sequence. NeuroPred also joins adjacent peptides to account for false positive cleavages and the presence of intermediate peptides that are eventually cleaved.

NeuroPred provides cleavage predictions using model developed from a vast range of species (including mollusk, insects and mammals) used in neuroscience research. Generally it is expected that the most appropriate model will be trained on the same or closely related species. However, it is expected that there are situations where there is no obvious appropriate model or that there is



a requirement for a greater understanding cleavage prediction at different sites. For these types of situations, NeuroPred can compute different model accuracy statistics when cleavage information is uploaded together with sequence. The resulting output enables the comparison of the selected models for individual precursors and for all precursors.

One valuable aspect of using Python was that much of the code developed for the analytical pipeline was reused in NeuroPred and can also be easily packaged into a stand-alone application. For example, the processing of sequence information and application of different cleavage prediction models requires the same code across the different applications. This feature allows the main coding to be focused on integrating components rather than developing a completely new application. Furthermore, additional or more efficient Python code developed for a new application can be reused by previous application. For example, the original prediction equations from different models were implemented using scalar computations. However, faster code was generated by implementing the prediction equations as a series of vector-matrix multiplications in Numerical Python. Improvements in computational speed were beneficial for all applications and particularly for NeuroPred because of the volume of requests handled by this public web service.

The text processing capabilities in Python were important to enable the integration of the NeuroPred application with the visual appearance of the main web site. The main site provides static information that does not change in response to the user. In contrast, the output from NeuroPred is dynamic because the output depends on user interaction. If the html coding recoding is directly used within the script, the script must be changed when the main web site changes. However, the string processing ability of Python permits Python scripts to easily search and replace portions of text. In particular, the template of the main web site or an existing web page in the required format can be directly parsed by Python and the necessary portions replaced such that the web application will provide the same visual appearance as the main web site. Alternatively, Python web frameworks such as Django (<http://www.djangoproject.com/>) can be used to develop and maintain extensive web sites.

CONCLUSION

The Python language is well-suited to implement a bioinformatics approach that encompasses a large number of interdependent steps, from scanning genomes for precursor genes to identification of neuropeptides. We did not encounter any shortcomings with Python that were specific to our application or that hampered our efforts to obtain results. The series of steps encompassed in the analytical pipeline implemented in Python reflect the flexibility of this language to support diverse applications. The versatility of Python across all steps, identification of neuropeptide precursors from genomic sequences, generation and training of cleavage prediction models, and development of a web application to predict cleavage sites, PTMs, and resulting peptides was illustrated.

The components of an neuropeptide analytical pipeline developed using Python supports the examination and annotation of genomes, prediction of cleavage sites, and characterization of resulting peptides, irrespectively of the extent of experimental neuropeptide evidence. The successful application of the discovery aspect of this pipeline led to the identification of 78 rhesus neuropeptide precursors, including 11 precursors that had not been predicted during the automated annotation of the genome. The training and evaluation of models to predict cleavage sites in rhesus precursors resulted in models that had correct classification rate of over 80% based on homologous cleavage assignments from human precursors indicating successful application of the cleavage prediction component of the pipeline. NeuroPred is a direct application of the neuropeptide analytical pipeline to provide an all-inclusive Python web application that allows users to predict precursor cleavage and subsequent PTMs of the resulting peptides. This application supports targeted experimental search for likely predicted peptides and greatly facilitates the laborious search for neuropeptides in mass spectra from high throughput proteomic studies.

The level of user input required to comprehensively identify the precursor complement depends on the available resources and on the divergence of the species under study with respect to other species with known precursor information. In this study we demonstrated how Python routines can aid with many tedious components of genome-wide precursor identification and cleavage prediction such as the processes that must be repeated for each precursor. Our routines help to address the challenges associated with species divergence and in-progress sequencing and assembly processes (e.g. coverage, accuracy) by facilitating the evaluation of different specifications (e.g. databases, scoring matrices, E-value thresholds) and of models from species with different level of divergence.

Results from characterization of the rhesus neuropeptidome using an analytical pipeline and implementation of the pipeline as a public web application that serves the neuroscience community demonstrate the suitability of the Python language for multiplexed and high throughput bioinformatics applications. The object-orientated nature of the Python language enabled considerable reuse of code at the different stages of development. A completely integrated approach can also be achieved by combining the bioinformatics tools in BioPython and the numerical tools in Numerical and Scientific Python.

ACKNOWLEDGEMENTS

The support of the National Institute on Drug Abuse Award P30 DA 018310 to the UIUC Neuroproteomics Center on Cell to Cell Signaling is gratefully acknowledged.

SUPPLEMENTARY MATERIAL

Supplementary material can be found online at <http://www.frontiersin.org/neuroinformatics/paper/10.3389/neuro.11/007.2008/>.

REFERENCES

- Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.* 25, 3389–3402.
- Amare, A., Hummon, A. B., Southey, B. R., Zimmerman, T. A., Rodriguez-Zas, S. L., and Sweedler, J. V. (2006). Bridging neuropeptidomics and genomics with bioinformatics: prediction of mammalian neuropeptide prohormone processing. *J. Proteome Res.* 5, 1162–1167.
- Bassi, S. (2007). A primer on Python for life science researchers. *PLoS Comput. Biol.* 3, e199.
- Bendtsen, J. D., Nielsen, H., von Heijne, G., and Brunak, S. (2004). Improved prediction of signal peptides: SignalP 3.0. *J. Mol. Biol.* 340, 783–795.
- Birney, E., Clamp, M., and Durbin, R. (2004). GeneWise and genomewise. *Genome Res.* 14, 988–995.
- Boutrel, B. (2008). A neuropeptide-centric view of psychostimulant addiction. *Br. J. Pharmacol.* 154, 343–357.
- Chang, C., and Lin, C. (2001). LIBSVM: a library for support vector machines. Available at: <http://www.csie.ntu.edu/~cjlin/libsvm>.
- Fricker, L. D. (2005). Neuropeptide-processing enzymes: applications for drug discovery. *AAPS J.* 7, E449–E455.
- Heinrichs, M., and Domes, G. (2008). Neuropeptides and social behavior: effects of oxytocin and vasopressin in humans. *Prog. Brain Res.* 170, 337–350.
- Hook, V. Y. (2006). Unique neuronal functions of cathepsin L and cathepsin B in secretory vesicles: biosynthesis of peptides in neurotransmission and neurodegenerative disease. *Biol. Chem.* 387, 1429–1439.
- Hook, V., Funkelstein, L., Lu, D., Bark, S., Wegrzyn, J., and Hwang, S. (2008). Proteases for processing proneuropeptides into peptide neurotransmitters and hormones. *Ann. Rev. Pharmac. Toxicol.* 48, 393–423.
- Hummon, A. B., Hummon, N. P., Corbin, R. W., Li, L. J., Vilim, F. S., Weiss, K. R., and Sweedler, J. V. (2003). From precursor to final peptides: a statistical sequence-based approach to predicting prohormone processing. *J. Proteome Res.* 2, 650–656.
- Hummon, A. B., Richmond, T. A., Verleyen, P., Baggerman, G., Huybrechts, J., Ewing, M. A., Vierstraete, E., Rodriguez-Zas, S. L., Schoofs, L., Robinson, G. E., and Sweedler, J. V. (2005). From the genome to the proteome: uncovering peptides in the Apis brain. *Science* 314, 647–649.
- Kinser, J. (2008). Python for Bioinformatics. Sudbury, Massachusetts: Jones and Bartlett Publishers.
- Li, L., and Sweedler, J. V. (2008). Peptides in the brain: mass spectrometry-based measurement approaches and challenges. *Annu. Rev. Anal. Chem.* 1, 451–483.
- Notredame, C., Higgins, D., and Heringa, J. (2000). T-coffee: a novel method for multiple sequence alignments. *J. Mol. Biol.* 302, 205–217.
- Oliphant, T. E. (2007). Python for scientific computing. *Comput. Sci. Eng.* 9, 10–20.
- Rhesus Macaque Genome Sequencing and Analysis Consortium (2007). Evolutionary and biomedical insights from the rhesus macaque genome. *Science* 316, 222–234.
- Salzberg, S. L. (1997). On comparing classifiers: pitfalls to avoid and a recommended approach. *Data Min. Knowl. Disc.* 1, 317–328.
- Southey, B. R., Amare, A., Zimmerman, T. A., Rodriguez-Zas, S. L., and Sweedler, J. V. (2006a). NeuroPred: a tool to predict cleavage sites in neuropeptide precursors and provide the masses of the resulting peptides. *Nucleic Acids Res.* 34, W267–W272.
- Southey, B. R., Rodriguez-Zas, S. L., and Sweedler, J. V. (2006b). Prediction of neuropeptide prohormone cleavages with application to RFamides. *Peptides* 27, 1087–1098.
- Southey, B. R., Sweedler, J. V., and Rodriguez-Zas, S. L. (2008). Prediction of neuropeptide cleavage sites in insects. *Bioinformatics* 24, 815–824.
- Tegge, A. N., Southey, B. R., Sweedler, J. V., and Rodriguez-Zas, S. L. (2008). Comparative analysis of neuropeptide cleavage sites in human, mouse, rat, and cattle. *Mamm. Genome* 19, 106–120.
- The UniProt Consortium (2008). The universal protein resource (UniProt). *Nucleic Acids Res.* 36, D190–D195.
- Wetkam, W., Groneberg, J., Leineweber, M., Wengenmayer, F., and Winnaker, E. -L. (1982). The nucleotide sequence of cDNA coding for preproinsulin from the primate *Macaca fascicularis*. *Gene* 19, 179–183.

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

Received: 04 September 2008; paper pending published: 26 September 2008; accepted: 11 November 2008; published: 16 December 2008

Citation: Southey BR, Sweedler JV and Rodriguez-Zas SL (2008) A Python analytical pipeline to identify prohormone precursors and predict prohormone cleavage sites. *Front. Neuroinform.* (2008) 2:7. doi: 10.3389/neuro.11.007.2008

Copyright: © 2008 Southey, Sweedler and Rodriguez-Zas. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.



Brian: a simulator for spiking neural networks in Python

Dan Goodman* and Romain Brette

Département d'Informatique, École Normale Supérieure, Paris, France

Edited by:

Rolf Kötter, Radboud University
Nijmegen, Netherlands Antilles

Reviewed by:

Robert C. Cannon, Textensor Limited, UK
Markus Diesmann, RIKEN Brain
Science Institute, Japan

***Correspondence:**

Dan Goodman, Equipe Audition,
Département d'Études Cognitives,
École Normale Supérieure, 29 rue
d'Ulm, 75230 Paris Cedex 05, France.
e-mail: dan.goodman@ens.fr

"Brian" is a new simulator for spiking neural networks, written in Python (<http://brian.di.ens.fr>). It is an intuitive and highly flexible tool for rapidly developing new models, especially networks of single-compartment neurons. In addition to using standard types of neuron models, users can define models by writing arbitrary differential equations in ordinary mathematical notation. Python scientific libraries can also be used for defining models and analysing data. Vectorisation techniques allow efficient simulations despite the overheads of an interpreted language. Brian will be especially valuable for working on non-standard neuron models not easily covered by existing software, and as an alternative to using Matlab or C for simulations. With its easy and intuitive syntax, Brian is also very well suited for teaching computational neuroscience.

Keywords: Python, spiking neurons, simulation, integrate and fire, teaching, neural networks, computational neuroscience, software

INTRODUCTION

A reasonable question to ask is whether there is any need for another neural network simulator. There are now several mature simulators, which can simulate sophisticated neuron models and take advantage of distributed architectures with efficient algorithms (Brette et al., 2007). Yet, many researchers in the field still prefer to use their own Matlab or C code for their everyday modelling work. It might be that currently available simulators do not fulfill the expectations of those users. Generally, what we expect from simulation software is that it should be able to run our specific model (flexibility) in a reasonable amount of time (efficiency). However efficiency is not only about the speed of simulations. The time it takes the user to implement the model is at least as important in many situations. For example, if it takes only 1 s to simulate a model with a given tool but 30 min to write the simulation script, one might prefer to use a tool which simulates the model in 10 s but for which the script can be written in 3 min. For those modelling situations, we only want the simulation software to be "reasonably fast".

Brian is a new project (<http://brian.di.ens.fr>) to create a clock driven spiking neural network simulator with the goals of being easy to learn and use, highly flexible, and "reasonably fast". It is ideally suited to rapid prototyping and refinement of networks of single compartment model neurons. Brian is written entirely in the Python programming language and will run on any platform that supports Python (i.e. almost all platforms). Users with a C compiler on their system can take advantage of a slight speed increase by opting to use certain core routines written in optimised C code, but these are strictly optional. Everything works the same without them. The way Brian works is that it is a Python package providing functions, classes and objects. It can be used either interactively using a Python shell, or as part of a Python program (module). **Figure 1** shows a very simple Brian script. This script defines a randomly connected network of 4000 leaky integrate-and-fire neurons with exponential synaptic currents. This is Brian's implementation of the current-based (CUBA) model network used as one of the benchmarks in Brette et al. (2007). The simulation takes 3–4 s on a typical PC, for 1 s of biological time (with $dt = 0.1$ ms). **Figure 2**

shows a more complicated example, illustrating many of the features of Brian.

BACKGROUND

One of the difficulties with current software for neural network simulation is the necessity to learn and use custom scripting languages for each tool: for example Neuron's Hoc and NMODL (Carnevale and Hines, 2006), NEST's SLI (Gewaltig and Diesmann, 2007), and Genesis' SLI (Bower and Beeman, 1998), the last two being different languages with the same name. This increases the learning curve and is less flexible than using an established language with strong support and development tools such as integrated development environments (IDEs), debuggers and profilers. Data analysis is either limited to those functions provided by the tool, or has to be carried out in another application such as Matlab, which can slow down the process of prototyping and refining models. Writing extensions to these tools can be rather difficult or somewhat inflexible, depending on whether extensions are written in the same language as the simulator itself.

To address this problem, there are projects in various stages of completion to provide Python interfaces for each of the tools mentioned above (see other chapters in this special issue). Because it is both easy and powerful, Python is rapidly becoming a standard tool in the field and in scientific computing more generally. In addition, the PyNN project is working to provide a unified Python interface to each simulator. These projects have considerable benefits. Users will only need to learn a single programming language rather than one or more for each tool, and that language is easy to learn, highly developed, very powerful, and has a large user base which provides excellent support and tools. A great deal of time can be saved working in just one environment, rather than having to switch back and forth between different applications and GUIs for developing models, running simulations and analysing data.

Brian complements these projects and has some additional benefits unique to it. Firstly, equations – differential equations in particular – can be defined at the highest level using standard mathematical notation (see **Figures 1 and 2**). Brian does not restrict you to using standard models of neurons and synapses (although many are provided in the


```

from brian import *
eqs = '''
dV/dt = (ge+gi-(V+49*mV))/(20*ms) : volt
dge/dt = -ge/(5*ms) : volt
dgi/dt = -gi/(10*ms) : volt
'''
P = NeuronGroup(4000, model=eqs,
                threshold=-50*mV, reset=-60*mV)
Pe = P.subgroup(3200)
Pi = P.subgroup(800)
Ce = Connection(Pe, P, 'ge')
Ci = Connection(Pi, P, 'gi')
Ce.connect_random(Pe, P, p=0.02,
                  weight=1.62*mV)
Ci.connect_random(Pi, P, p=0.02,
                  weight=-9*mV)
M = SpikeMonitor(P)
P.V = -60*mV+10*mV*rand(len(P))
run(.5*second)
raster_plot(M)
show()

```

$$\tau_m \frac{dV}{dt} = -(V - E_L) + g_e + g_i$$

$$\tau_e \frac{dg_e}{dt} = -g_e$$

$$\tau_i \frac{dg_i}{dt} = -g_i$$

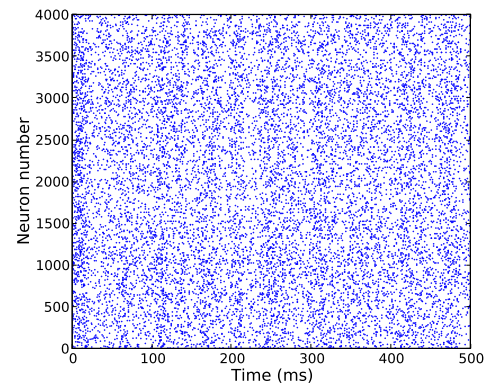


FIGURE 1 | The CUBA network in Brian, with code on the left, neuron model equations at the top right and output raster plot at the bottom right. This script defines a randomly connected network of 4000 leaky integrate-and-fire neurons with exponential synaptic currents, partitioned into a group of 3200 excitatory neurons and 800 inhibitory neurons. The `subgroup()` method keeps track of which neurons have been allocated to subgroups and allocates the next available neurons. The process starts from neuron 0, so `Pe` has neurons 0 through 3199 and `Pi` has neurons 3200 through 3999. The script outputs a

raster plot showing the spiking activity of the network for a few hundred ms. This is Brian's implementation of the current-based (CUBA) network model used as one of the benchmarks in Brette et al. (2007), based on the network studied in Vogels and Abbott (2005). The simulation takes 3–4 s on a typical PC (1.8 GHz Pentium), for 1 s of biological time (with $dt = 0.1$ ms). The variables g_e and g_i are not conductances, we follow the variable names used in Brette et al. (2007). The code `: volt` in the equations means that the unit of the variable being defined (V , g_e and g_i) has units of volts.

library), and neuron models based on new differential equations can be used without writing or compiling any code. Secondly, as Brian is written entirely in Python itself, it has all the advantages of the projects above and some additional ones. Integration with Python is tighter because the implementation is not in a separate language to the interface. This means that Brian can be used more flexibly, for example to write code which reads and modifies the variables of the simulation as it runs. Additionally, extensions to Brian are easy to write because everything is written in the same language.

TEACHING

Brian was originally designed for research, but it would also make an ideal tool for teaching purposes. First of all, the Python language is extremely quick and easy to learn and the syntax allows code to be written very compactly, saving time and making it easier to present examples. Secondly, since Brian is written in pure Python, it works on almost every platform, so there are less compatibility issues for students with different hardware or operating systems. Finally, using Brian itself is very easy, and the core concepts and syntax of Brian code correspond very straightforwardly to neuroscientific concepts (see Figure 1). Equations are specified using a familiar mathematical syntax, for example `eqs='dV/dt=-V/tau: volt'`, where the only unfamiliar part of the syntax is the `: volt` term, which specifies that V has units of volts. Figure 1 shows that defining thresholds and resets is typically just a single keyword term such as

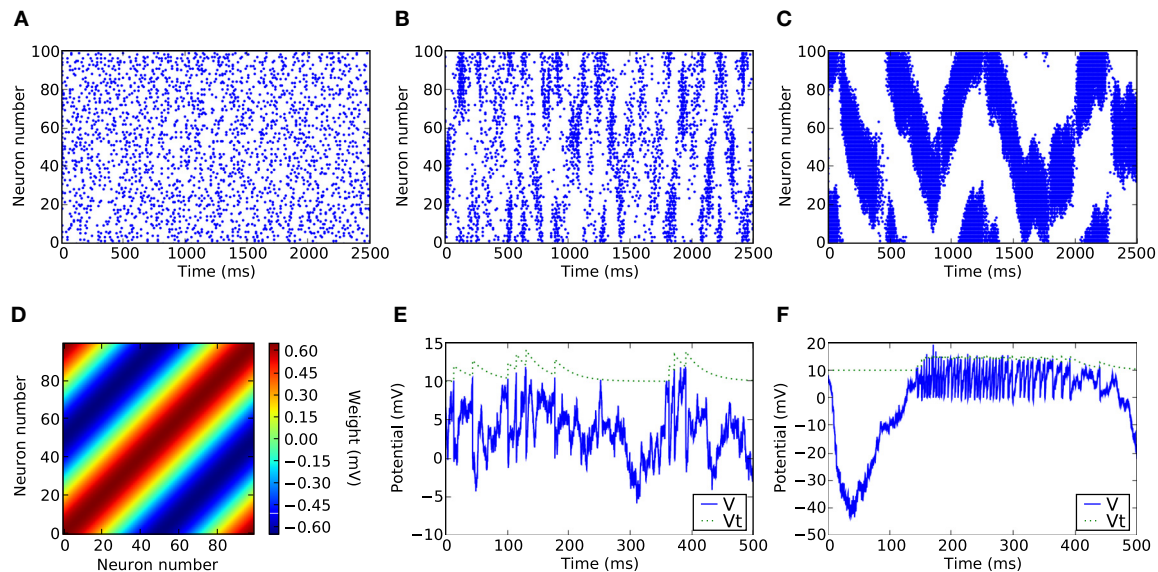
`threshold=-70*mV` or `reset=-55*mV`, and creating groups of neurons is as simple as writing `G=NeuronGroup(N,model)`.

FEATURES

Brian is a clock driven simulator, that is, all events take place on a fixed time grid $t = 0, dt, 2dt, 3dt, \dots$. Neuron models are normally defined by differential equations which can be arbitrary linear, nonlinear or stochastic, specified either by directly writing the equations in a string, by using standard equations such as leaky integrate-and-fire, or by building more complicated sets of equations using standard components such as K^+ and Na^+ currents. Both integrate-and-fire and Hodgkin–Huxley type models can be used. Multiple compartment models are possible, but at the moment they are neither particularly convenient nor efficient for more than a few compartments. For linear differential equations, Brian uses exact updates. For nonlinear differential equations, Euler (explicit) and exponential Euler (semi-implicit) methods are available (and more are planned).

Network connectivity can be built either directly by specifying connectivity per pair of neurons (i, j) , or more efficiently with all-to-all or random connectivity, where the synaptic weights can be either single values or specified by a weight function $f(i, j)$. Synaptic connections can include delays.

Network activity can be controlled in various ways. For spiking behaviour there are various standard models such as Poisson



```

from brian import *
w = .5*mV
def adaptive_threshold_reset(P, spikes):
    P.V[spikes] = 0*mV
    P.Vt[spikes] = clip(P.Vt[spikes]+2*mV, 10*mV, 15*mV)
eqs = ''' dV/dt = (5*mV-V)/(10*ms) + 4*mV*xi/(10*ms)**.5 : volt
          dVt/dt = (10*mV-Vt)/(30*ms) : volt '''
group=NeuronGroup(100, model=eqs,
                  threshold=lambda V,Vt:V>=Vt,
                  reset=adaptive_threshold_reset)
C = Connection(group, group, 'V', delay=2*ms)
S = SpikeMonitor(group)
C.connect_full(group, group, weight=lambda i,j:w*cos(2.*pi*(i-j)*1./100))
group.V = rand(100)*5*mV+5*mV
group.Vt = 10*mV
run(2.5*second)
raster_plot(S)
show()

```

FIGURE 2 | An example showing many of the features of Brian in action. The neuron model in this code follows a stochastic differential equation $dV/dt = -(V - E_i)/\tau + \sigma \xi(t)/\sqrt{\tau}$, $dV_t/dt = -(V_t - V_{t0})/\tau_t$. Here all the undefined symbols are constants except for $\xi(t)$ which corresponds to the term \mathbf{xi} in the code, and represents a white noise term ($\langle \xi(t) \xi(t') \rangle = \delta(t - t')$). The rest of the neuron model is defined by a custom reset function `adaptive_threshold_reset` which increases the value of V_t by a constant each time a neuron spikes (but never takes it above a fixed ceiling), and a custom threshold function `lambda V, Vt: V>=Vt` which defines the condition for a spike. The arguments to the custom reset function are a `NeuronGroup` object `P` (a population of neurons),

and an array `spikes` containing the indices of the neurons in `P` that have spiked. Together these two custom functions define an adaptive threshold model. The option to specify custom functions makes Brian's reset and threshold mechanism very flexible. The code also shows synaptic delays, and setting the synaptic weights with a custom function of (i, j) , $w \cdot \cos(2 \cdot \pi \cdot (i - j) \cdot 1 / 100)$. The output of the code shown is the raster plot in (B), with the value $w = .5 \text{ mV}$. (A) shows $w = .1 \text{ mV}$ and (C) shows $w = .65 \text{ mV}$. (D) shows the synaptic weight matrix for the $w = .65 \text{ mV}$ case. (E) and (F) show the values of V (solid blue) and V_t (dashed green) for the neuron with index 50 for the raster plots immediately above them ((B) and (C)) with $w = .5 \text{ mV}$ and $w = .65 \text{ mV}$ respectively.

spiking neurons, and more direct control mechanisms can be used to specify spike times for a neuron with a list or Python function. While the simulation is running, all the variables of the simulator are directly accessible and this can be used for controlling almost any aspect of the simulation. The emphasis is on flexibility, and most aspects of the way Brian works can be overridden.

Basic support for short term plasticity and spike timing dependent plasticity is included. This will be standardised and made easier to use in later releases.

Brian also has a system for specifying quantities with physical dimensions, which makes things easier because variables can be entered without having to look up the scale defined for that variable

by the simulator package, and is useful because it helps to catch hard to debug problems stemming from parameters or equations having inconsistent units (see Physical Units).

Finally, Brian is fairly efficient. Although Python is an interpreted language, it can still achieve speeds comparable to that of code written directly in C, and typically better than code written in Matlab. See the section “Simulation Speed” for a discussion of performance issues.

HOW IT WORKS

Brian is designed to be easy to use, flexible and reasonably fast. To achieve the first goal, Brian uses features of the Python programming language, in particular its extremely dynamic typing which allows code to be much simpler and more expressive. Flexibility in Brian stems from using a single high-level language for user code and the library itself, and from making differential equations a fundamental high-level data structure (see Background). For the third goal, Brian uses the strategy of vectorised code.

Brian makes considerable use of Python’s dynamic typing to make writing models easier, and to make the syntax concise and readable. So for example, in specifying a neuron model a thresholding procedure is required for producing spikes. This can be done by specifying a single number, a function, or a threshold object. In the first case, with the threshold specified by a single number V_t say, Brian infers the thresholding condition $V \geq V_t$. In the second case, Brian examines the function provided. Consider a neuron model with variables V and Vt , and the threshold specified as the function `lambda V, Vt: V>=Vt` (which is the Python expression for a function of two variables V and Vt which returns the value $V \geq Vt$). In this case Brian examines the names of the arguments to the function and passes the appropriate values so that the code behaves as expected. This would be one way of providing a variable threshold condition (because Vt is a variable of the neuron model, and could evolve according to a differential equation or function of other variables for example). Another way is to provide a threshold object, either one of the standard types in the library, or a user-defined one by writing a class that derives from the `Threshold` class. The variable threshold condition above corresponds to the standard object `VariableThreshold('Vt')` for example.

Vectorising code is the strategy, familiar to users of Matlab, to minimise the amount of time spent in interpreted code compared to highly optimised array functions. This typically means trying to minimise the number of `for` loops in code, and using data structures and algorithms that make this easier. Brian uses the NumPy package (see below) which has an array data type that makes, for example, the expression `V[spikes]=Vr` equivalent to but much faster than `for i in spikes: V[i]=Vr`. In Matlab this would be `V(spikes)=Vr`, and in many cases the NumPy syntax is very similar to the Matlab syntax making the transition between the two very easy. The issue of Brian’s speed and efficiency is covered in more detail in the section “Simulation Speed”.

Brian uses the following standard Python packages: Numerical Python, which is designed for providing efficient array data structures and operations (NumPy, <http://www.scipy.org/NumPy>), Scientific Python, which extends NumPy to include more general algorithms for scientific work (SciPy, <http://www.scipy.org>),

and PyLab/Matplotlib for plotting (<http://matplotlib.sourceforge.net/>).

WORKED EXAMPLE

Figure 3 shows a slightly simplified version of the code in Figure 1 with diagrams showing schematically the meaning or function of each group of lines of code. Panels A through F illustrate lines of code, and Panel F, which corresponds to actually running the simulation, is composed of four sub-panels a through d which illustrate the four steps involved in each timestep dt of the simulation. We proceed to explain how this example works with reference to the figure.

A Firstly, the differential equations for the model are defined. This is illustrated in Panel A which shows the code which defines the equations and the equations in a more standard mathematical form. These equations will be used to define an integrate-and-fire neuron with exponential inhibitory and excitatory synapses with different time constants. The differential equation for V defines a leaky integrator with currents g_e and g_i . The variable g_e is used for excitatory currents. When an excitatory spike arrives, the value of g_e is increased instantaneously by a fixed amount. The inhibitory variable g_i works similarly. Technically then, the full mathematical differential equations for the model would be:

$$\begin{aligned}\tau \frac{dV^j}{dt} &= -(V^j - V_r) + g_e^j + g_i^j \\ \tau_e \frac{dg_e^j}{dt} &= -g_e^j + \tau_e \sum_l \sum_{k=1}^N W_e^{kj} \delta(t - t_l^k) \\ \tau_i \frac{dg_i^j}{dt} &= -g_i^j + \tau_i \sum_l \sum_{k=1}^N W_i^{kj} \delta(t - t_l^k)\end{aligned}$$

where the superscripts indicate neuron indices, W_*^{kj} are the excitatory and inhibitory weight matrices, $N = 4000$ is the number of neurons, and t_l^k is the time of the l th spike fired by neuron k . The spike propagation behaviour is defined in Panels C and D, see the description below.

B Having defined the differential equations, a group `P` of 4000 neurons is created with these equations, a threshold mechanism set to fire spikes if $V \geq V_t = -50$ mV, and a reset $V \leftarrow V_r = -60$ mV. The diagram in Panel B shows Brian’s internal data structure for this group. It is a two-dimensional array or matrix S . At a given time the i th column of S holds the state variables for the i th neuron. Each row of the matrix is a vector of length 4000 of the values of a particular variable for all the neurons in the group.

C The next step is to create the network structure. We create two subgroups `Pe` and `Pi` of 3200 and 800 neurons respectively. The `subgroup()` method of the `NeuronGroup` object keeps track of which neurons have been allocated to subgroups and when called allocates the next available neurons. The process starts from neuron 0, so `Pe` has neurons 0 through 3199 and `Pi` has neurons 3200 through 3999. These two subgroups will be the excitatory and inhibitory neurons. In the diagram in Panel C, we have separated the columns of the state matrix S corresponding to each neuron. The excitatory and inhibitory

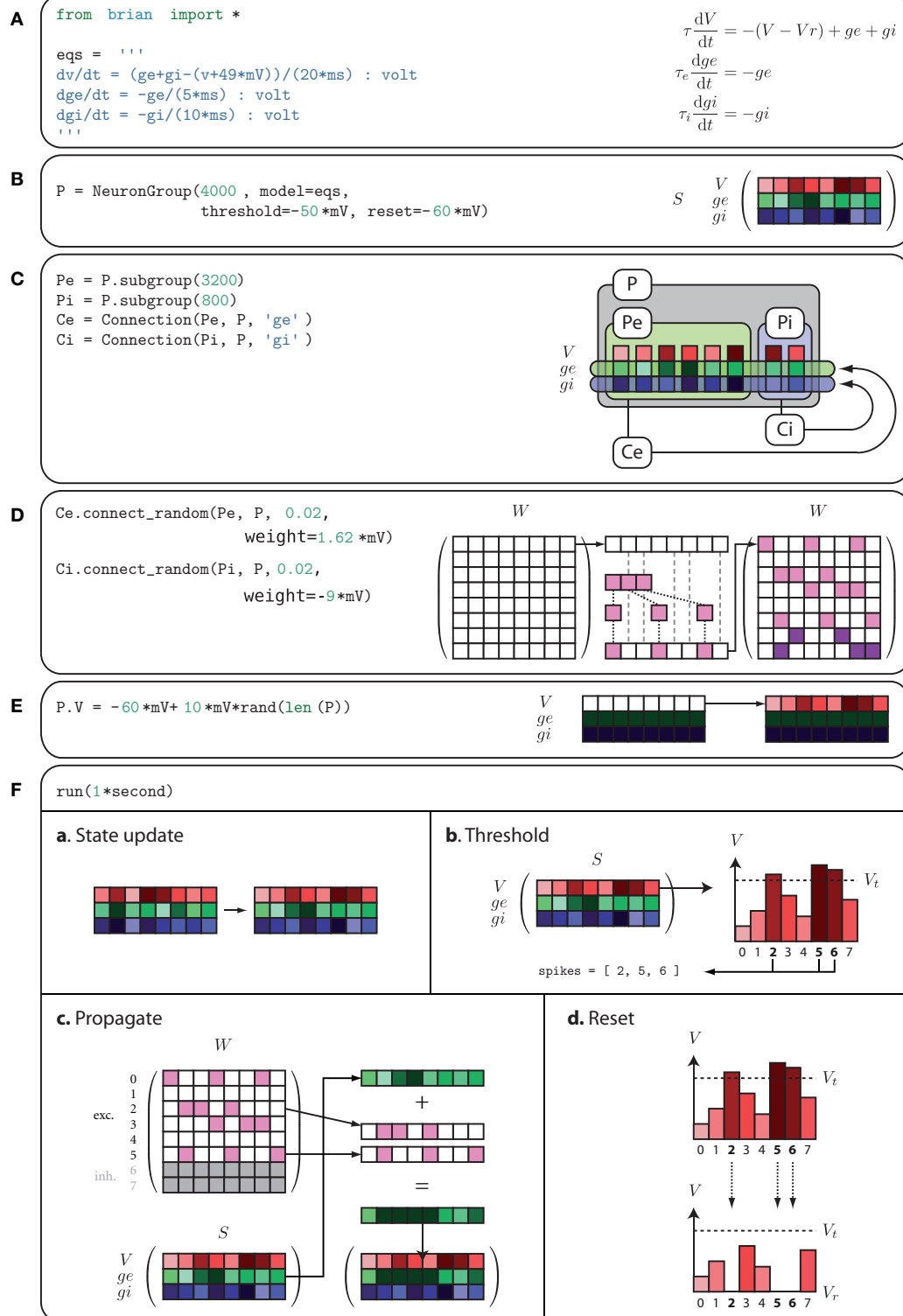


FIGURE 3 | The code from Figure 1 expanded to show how Brian works internally. In (A), the equations for the model are defined. In (B), a group of 4000 neurons is created with these equations. In (C), the logical structure of the network is defined, partitioning the 4000 neurons into excitatory and inhibitory subgroups with corresponding connections to the whole group. In (D), the weight matrices for the excitatory and inhibitory connections

are defined. In (E), the membrane potential is initialised uniformly randomly between reset and threshold values. In (F), the simulation is running, consisting of repeated applications of four operations each time step; (a) shows the update of the state matrix; (b) shows the thresholding operation; (c) shows the propagation of spikes; and (d) shows the reset operation.

subgroups are boxed and labelled P_e and P_i respectively. Next, excitatory and inhibitory connections C_e and C_i are created. The declaration of C_e specifies that the group P_e (the excitatory subgroup) should be connected to the variable g_e (the excitatory current) of the group P (the whole group), and similarly for C_i . This means that when a neuron in P_e fires a spike, the variable g_e will be increased for those neurons in P which the neuron in P_e synapses onto.

- D** Having defined the logical network structure, we create the weight matrix itself. Each pair of neurons (i, j) are connected independently at random with probability 0.02. The excitatory synapses have weight 1.62 mV and the inhibitory ones have weight -9 mV (negative to make it inhibitory, and larger than the excitatory synapses as there are less inhibitory neurons). For efficiency, the random connectivity function constructs the sparse matrix row by row. For each row it generates a binomial random number k from $B(N, p)$ which is the number of synapses in that row, and then randomly allocates those k synapses amongst the N possible target neurons, assigning them with equal fixed weight values. This process is illustrated in the diagram in Panel **D**.
- E** Now we prepare to actually run the simulation. The first step is to initialise the variables. At the start, all variables have the value zero. In Panel **E**, on the left hand side of the diagram, this is indicated by the V row being white (as 0 is much bigger than the threshold value which is negative), and the g_e and g_i rows being almost black. We leave the values of g_e and g_i as 0, and set V to be uniformly distributed between the reset and threshold values. The notation $P.V$ refers to the first row, the V row, of the state matrix S .
- F** Finally, we run the simulation. Panel **F** shows the four operations executed each time step dt of the simulation: state update, threshold, spike propagation, and reset. In the state update phase (sub-panel **a**), the state matrix S is updated from $t \rightarrow t + dt$, which as the differential equations are linear is just multiplication of S by a fixed matrix and addition of a fixed vector to each column of S . In the thresholding stage (sub-panel **b**), each value of V is simply compared to V_r and a list **spikes** of the indices of each of the neurons satisfying the condition is returned. In the propagation phase (sub-panel **c**), which is carried out separately for the excitatory and inhibitory connections, for each index $i \in \text{spikes}$ the i th row of $W_e, W_i[i, :]$, is added to the row vector corresponding to the variable g_e . Finally, in the reset phase (sub-panel **d**), for each index i in **spikes**, V is reset to V_r .

This worked example shows the general anatomy of a Brian script: import the Brian package and define neuron models (Panel **A**); create groups of neurons (Panel **B**); create synaptic connections (Panels **C** and **D**); create monitors and other operations for recording data and controlling variables as a simulation runs (not shown in figure); initialise variables (Panel **E**); run the simulation (Panel **F**); and finally analyse and plot the data using any Python package (not shown in figure). Creating monitors and plotting output is not shown in **Figure 3** but can be seen in **Figure 1**. The lines `M=SpikeMonitor(P)` and `raster_plot(M)` record and plot the spikes produced by the neurons in P . The `raster_plot` function

is part of Brian, but there are many Python packages which can be used for analysing and plotting data, including the ones used by Brian itself, NumPy, SciPy and PyLab/Matplotlib.

PHYSICAL UNITS

Brian also features a system for specifying physical quantities with units. This is an independent package originally written for Brian but now available as a standalone package called Piquant (<http://piquant.sourceforge.net/>). It builds on the NumPy and SciPy packages, adding support for physical quantities. This has various benefits. It makes it possible to write code which syntactically and semantically expresses both the physical dimensions and scale of numbers. So for example, something like `conductance=36*mS` rather than `conductance=36`. In the latter case, the code alone does not express the value without knowing the standard scale for the software, and this often leads to errors which can be very hard to debug. In addition, because units retain their physical dimensions as well as their scale, accidentally writing something using the wrong units will cause an error (for example in Brian, differential equation with inhomogeneous units will raise an error).

A quantity with physical units is a standard float value with an additional array of the indices of the seven fundamental SI units distance, mass, time, etc. The float value expresses the quantity at the standard SI scale, so that for example the float value of `1*mV` is 0.001. Operating on quantities with physical units is clearly more computationally demanding than operating on quantities without. To ameliorate this problem, Brian does two things. First of all, internal calculations done by Brian during a simulation only use the underlying float values, so that only initialisation code and custom functions use the units system. Secondly, Brian includes an option for switching the units system off globally. This only requires the addition of a single line of code to the top of a Brian program, and simply converts all the objects with units to their underlying float values. So for example with units turned off the symbol `mS` becomes the float value 0.001. The recommended usage is to leave the units system on when developing a model or when adding new code, and turning it off for longer and larger runs once the code is stable.

TECHNICAL DETAILS

The user specifies a model by providing the mathematical equations which define it. This can either be done directly by writing out the differential equations in full, or by building a set of equations using objects from the library (for things like ion channels or synapses). The former is useful in situations where there are not too many equations and where they are constantly being changed in the process of developing the model. The latter is useful in situations where the model is built from standard components and produces an unwieldy number of equations.

Given a final set of equations, Brian produces a `StateUpdater` object. In general, this is an object that updates the state variables of a group of neurons in any way. For differential equations, it performs the integration step updating the state variables from times t to $t + dt$. Brian automatically inspects the equations to choose the most appropriate type of `StateUpdater`. For linear differential equations for example, updates are exact. More precisely, if the

equations are $\dot{X} = M(X - B)$ then the exact solution for the update step is $X(t + dt) = e^{Mdt}(X(t) - B) + B$, where e^{Mdt} is a constant matrix and B is a constant vector evaluated (numerically) at initialisation time (see Morrison et al., 2007 for a closed form method). Nonlinear equations are integrated by default with Euler updates, and the exponential Euler method (a semi-implicit method, MacGregor, 1987) is also implemented for Hodgkin–Huxley models. The second-order Runge–Kutta method is also implemented. Stochastic differential equations are integrated with Euler updates (i.e., adding normally distributed random numbers every time step). Nonlinear equations given as text are compiled to Python functions at initialisation time, then used directly during the update phase with vector arguments [for example, $\dot{x} \leftarrow x + f(x)dt$ for a single state variable x and equation $dx/dt = f(x)$].

A `NeuronGroup` object is created by specifying the number of neurons in the group and a model. A model requires a set of differential equations or a `StateUpdater` object, and can have optional thresholding and reset mechanisms. A `Connection` object is a mechanism for propagating spikes from one `NeuronGroup` to another. It is specified by an input group, an output group (which can be the same) and a target state variable. When a neuron in the input group fires a spike, the target state variable is increased for all the neurons in the output group to which that neuron is connected. This mechanism is very general and allows for all the standard types of synapses. Once a `Connection` object has been created, the actual connectivity of neurons can be specified in various ways. The main four ways are full connectivity, random connectivity, functionally specified connectivity (e.g. for spatial distributions) or by providing a connectivity matrix directly. The `Connection` methods `connect_random` and `connect_full`, for random and full connectivity respectively, take as their first two arguments the source and target neuron groups. This seems redundant because the `Connection` object knows the source and target groups, but the weight matrix can be constructed in blocks and the first two arguments to these methods can be subgroups of the groups specified in defining the `Connection`. In the present version, homogeneous synaptic delays can also be specified. Each neuron group stores a circular list of the last spikes over the required delay, each element of that list being an array of the indexes of neurons that spiked during one timestep. Spikes are then delivered in the same way as explained in the section “Worked Example” (Panel F).

SIMULATION SPEED

Python is an interpreted language, and although it is very fast there is an overhead for every Python operation. Brian can achieve very good performances by using the technique of vectorisation, similar to the same technique familiar to Matlab users. The idea is to replace loops by operations on large vectors, so that the interpretation overhead becomes negligible. Brian uses vectorisation for both the simulation and the construction of the model (e.g., initialisation of synaptic weights).

For example, for a single neuron i with state vector \mathbf{x}_i , the update step from $\mathbf{x}_i(t)$ to $\mathbf{x}_i(t + dt)$ might be $\mathbf{x}_i(t + dt) = M\mathbf{x}_i(t) + \mathbf{b}$ for a matrix M and vector \mathbf{b} . This operation is the same for every i so rather than looping through all the neurons carrying out the same operation, we write a state matrix S whose columns are the state vectors of each neuron. Now the loop carrying out the operation for

each neuron i can be written in one operation, $S(t + dt) = MS(t) + B$ (where B is a matrix with every column equal to \mathbf{b}). The number of mathematical operations is the same, but the interpretation overhead is reduced from N interpretation operations for N neurons to 1 interpretation operation. Brian uses the NumPy package for these vectorised operations. NumPy is written in optimised C code, and for linear algebraic operations uses the Basic Linear Algebra Subprograms (BLAS) application programming interface (API). This means that NumPy can be combined with an implementation of the BLAS API that is optimised for the specific details of the processor it is running on. For large networks, the time spent on mathematical operations is much larger than the time spent on interpretation operations and so Brian is very efficient. For smaller networks, the interpretation overhead is much larger in proportion but in many situations it is not critical because the simulation time is shorter too. The least favourable scenario for Brian is the simulation of a small network for a long biological time.

PERFORMANCE OF VECTORISED SIMULATIONS

In this section, we outline an analysis of Brian’s performance. A formula for the simulation time of a network with a clock-driven algorithm is given in Brette et al. (2007):

Update + Propagation

$$c_U \times \frac{N}{dt} + c_p \times F \times N \times p$$

where c_U is the cost of one update and c_p is the cost of one spike propagation, N is the number of neurons, p is the number of synapses per neuron, F is the average firing rate and dt is the time step (the cost is for 1 s of biological time). If the simulation is fully vectorised, then interpretation can be included in this formula as a constant overhead c_I per time step:

Update + Propagation + Interpretation

$$c_U \times \frac{N}{dt} + c_p \times F \times N \times p + \frac{c_I}{dt}$$

and the interpretation overhead becomes negligible when the network is large. In more detail, the update constant c_U grows with the complexity of the model (in particular the number of variables) and the interpretation constant c_I grows with the number of objects created, such as groups of neurons. Therefore, the strategy for running efficient simulations with Brian is to collect all neurons sharing the same differential equations in the same group. It is still possible to have heterogeneous groups in this way, for example the following code defines a group of 100 integrate-and-fire neurons with membrane time constants between 5 and 30 ms:

```
eqs = '''
dv/dt = -v/tau : volt
tau : second
'''
G = NeuronGroup(100, model=eqs, threshold=15*mV,
reset=0*mV) G.tau = linspace(5*ms, 30*ms, 100)
```

Here `tau` becomes a state variable instead of a parameter. The same method can be used to obtain the results of a simulation for different parameter values. Note that with this change the differential

equation becomes nonlinear with respect to the two variables; equations are then integrated with an approximation scheme (Euler by default). A mechanism for declaring state variables to be constant so that the above equation would be considered linear and integrated with exact matrix updates (one matrix for each parameter value) is in preparation for a future release of Brian.

In many cases, the initialisation can also be vectorised. For example, the following instruction connects all pairs of neurons of a group with a distance-dependent weight (the topology is a ring):

```
C.connect_full(group,group,weight=lambda i,j:
cos((2.*pi/N)*(i-j)))
```

The program builds the weight matrix row by row by calling the weight function with arguments (i, j) where i is the row number and j is the vector $(0, 1, \dots, N-1)$. Thus, the matrix is constructed with N vector-based operations, in a way that is transparent to the user. This is made possible by the fact that Python is a dynamically typed language (functions do not need to specify the type of their arguments in their definition).

COMPARISON WITH C AND MATLAB

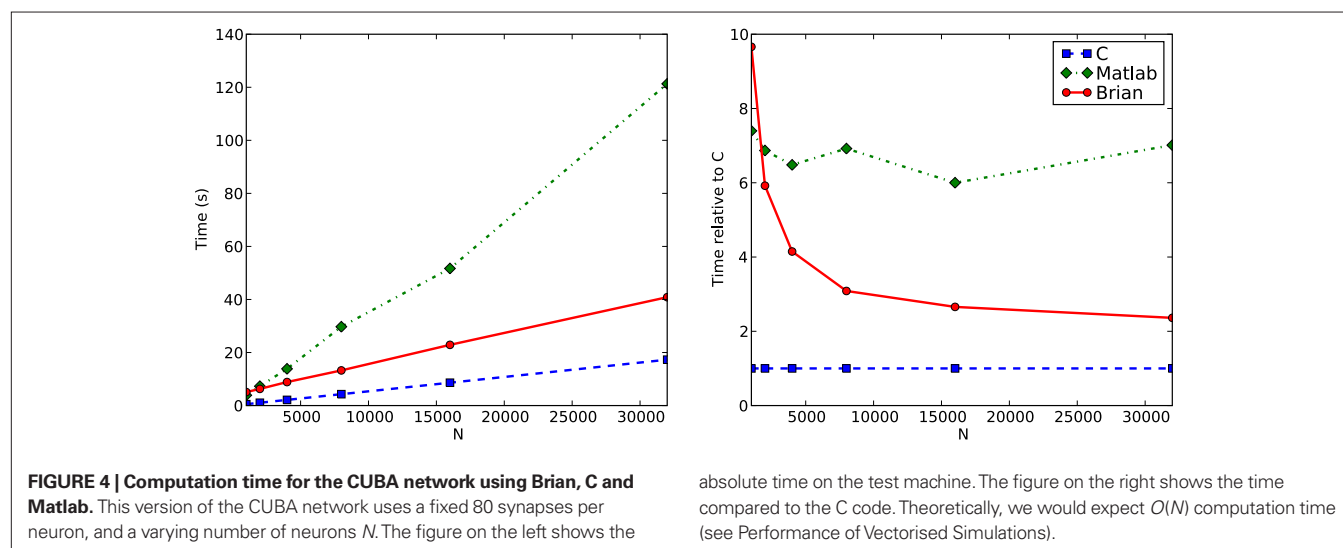
In this section, we compare the empirical performance of Brian with that of C and Matlab. We compare absolute performance and, since it was always the fastest, times relative to C. The C code was always compiled with the heaviest optimisations possible, the `-O3` switch with the `gcc` compiler. Brian was always run with the optional compilation switch on, and unit checking turned off. This means that certain key routines (the thresholding operation and the spike propagation phase) were written in C to avoid the Python overheads. These key operations are very generic, and so having them written in C rather than pure Python does not affect the flexibility of Brian as a whole. Note that this compilation switch is optional, and on a system without a C compiler installed Brian will use alternative versions of these core routines which are slightly slower but still very usable. Typically, running Brian with pure Python only takes about 25–50% longer than with the C routines. In the following benchmarks, times were computed by running each set of parameters 10 times and taking only the 7 best times, which

helps to remove outliers where performance is degraded due to the operation of an unrelated process running on the system. The comparisons shown were obtained using a 2.33 GHz Intel Xeon processor with 2 GB RAM running on Windows XP. The version of NumPy used was 1.1.1 with the default BLAS linear algebra package. Using a custom build of NumPy with a BLAS package tuned for the particular CPU architecture would give better performance. The source code for the comparisons is available on request.

The first benchmark we consider is a modified version of the CUBA network presented above in **Figures 1 and 3**. This is a network of linear differential equations, and Brian does exact updates for the state matrix for $t \mapsto t + dt$ which amounts to a matrix multiplication. We used the same mechanism exactly for the C and Matlab code. In all cases, the connection matrix uses a sparse matrix data structure implemented in effectively the same way.

We first modify the network so that instead of random connectivity with each pair of neurons connected with probability 0.02, the probability is p/N , where N is the number of neurons, making an average of p synapses per neuron independent of N . This guarantees that the firing rate of an individual neuron is independent of N . According to the calculations in the section “Performance of Vectorised Simulations” then, the computation time as a function of N should be proportional to N . **Figure 4** shows the times for this network. You can see that the performance of Brian is better than Matlab, but not as good as C. You can also see that as N increases, the relative performance of Brian compared to C improves. This is because the Python overheads are a fixed cost independent of N . At $N = 32,000$, Brian takes approximately 2.4 times as long as C, and we would expect that this ratio would improve further for larger N . For this N , Matlab takes approximately seven times as long as C.

The next benchmark is the same CUBA network, but this time with all synapses removed. Performance in general is largely dominated by two factors: the state update phase, and the spike propagation phase. This benchmark gives an idea of how performance for the state update phase alone scales. **Figure 5** shows the comparison. For large N , Brian takes around twice as long as C, and Matlab about four times as long. The jump in the times for Brian going from $N = 16,000$ to $N = 32,000$ may be due to CPU cache behaviour.



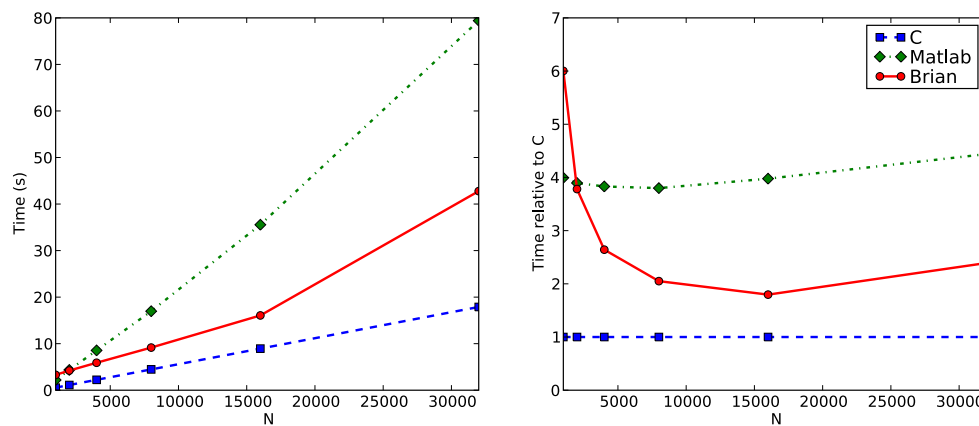


FIGURE 5 | Computation time for the CUBA network if all synapses are removed. This largely demonstrates the performance for the state update step, which in this case is a matrix multiplication.

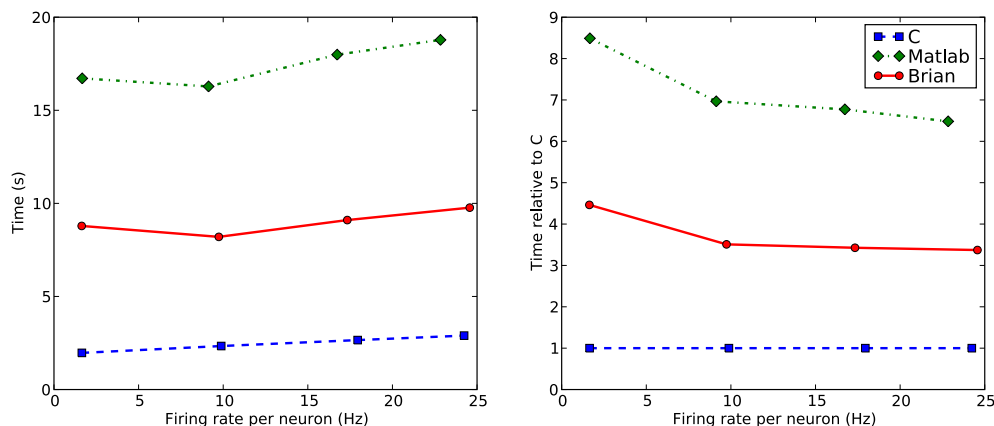


FIGURE 6 | Computation time for the CUBA network with on average $p = 500$ synapses per neuron and $N = 4000$ at different firing rates. The parameter w_e , the excitatory weight, was varied between 1.62 and 4.8 mV which had the effect of varying the firing rate between about 5 Hz and about

25 Hz. This shows how performance scales with the number of spikes. Here the firing rates as well as the times are averaged over the seven fastest trials, as firing rates vary from trial to trial. Note that times due to spiking depend on both the firing rate and the number of synapses per neuron.

The next benchmark uses a fixed N , but varies the parameter w_e , the excitatory synaptic weight. Increasing this increases the firing rate. **Figure 6** shows the comparison. For the range of w_e shown, leading to a range of firing rates from about 5 Hz to about 25 Hz, the times appear to grow at a similar rate for each of C, Matlab and Brian.

In conclusion, Brian is mostly around two to four times slower than C code for the typical network considered, and Matlab is around seven times slower. For smaller networks, Brian is slower than this, and for larger networks, we expect Brian to be faster than this. This seems like a reasonable trade off, given that smaller networks tend to take less time to run in absolute terms than larger networks.

DISCUSSION AND FUTURE WORK

Brian has been developed for quickly coding models of spiking neural networks in everyday situations. It is easy to learn, intuitive and flexible, which also makes it ideal for teaching. Although it is written in an interpreted language, it remains computationally

efficient in many situations thanks to vectorised algorithms. It is however not currently designed for very large scale simulations which require clusters of computers, or for detailed biophysical models with complex morphologies.

COMMUNITY

Brian is open source, and we are following the open source strategies of code reuse and interoperability. To make the development effort lighter and support easier, we chose to use existing packages and components as much as possible, and only write what is necessary on top of that. In writing Brian, we have used the NumPy, SciPy and PyLab/Matplotlib packages. There is a PyNN module for Brian currently in development, through which Brian will support open standards such as NeuroML (Goddard et al., 2001) and other XML description standards (Cannon et al., 2007).

We would also encourage others to make their code written with Brian accessible to others. Complete models can be posted to ModelDB (Hines et al., 2004), and in addition there is the new “Computational Neuroscience Cookbook” project hosted on the

NeuralEnsemble website (<http://neuralensemble.org/cook-book>). The idea of the cookbook is for submission of fragments of code which can be cut and pasted into others' code. Finally, we encourage others to contribute to the Brian project itself (<http://brian.di.ens.fr/contribute.html>).

FUTURE WORK

In the near future, our priorities for improving Brian are increasing the efficiency of Brian simulations and adding more modelling features. Specifically, we have started using the parallel processors present in modern graphics cards (GPU, Graphics Processing Unit) to improve the speed of Brian simulations with no additional work from the user (Luebke et al., 2004). These can be used as parallel coprocessors for vectorised calculations (Cummins et al., 2008). On the modelling side,

we are focusing our efforts on synaptic plasticity. It is already possible to simulate spike timing dependent plasticity (STDP, as in e.g. Song et al., 2000) and short term plasticity (STP; Tsodyks and Markram, 1997) with the current mechanisms implemented in Brian (since these are defined as differential equations with resets in those references, see Morrison et al., 2008 for a review of plasticity rules), and we are working on making it as flexible and simple to use as possible.

ACKNOWLEDGEMENTS

This work was partially supported by the European Union (Visiontrain, a Marie Curie Research Training Network) and by the French ANR (ANR-RIAM Wired Smart). The authors would like to thank all those who tested early versions of Brian and made suggestions for improving it.

REFERENCES

- Bower, J. M., and Beeman, D. (1998). *The Book of GENESIS: Exploring Realistic Neural Models with the General NEural Simulation System*, 2nd edn., Springer-Verlag, New York.
- Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., Boustani, S. E., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J. Comput. Neurosci.* 23, 349–398.
- Cannon, R., Gewaltig, M.-O., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., and Schutter, E. D. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.
- Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press, Cambridge, UK.
- Cummins, G., Adams, R., and Newell, T. (2008). Scientific computation through a GPU. In *Proceedings of the Southeastcon 2008*, an IEEE conference, Huntsville, AL, pp. 244–246. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4494293
- Gewaltig, O., and Diesmann, M. (2007). NEST (neural simulation tool). *Scholarpedia* 2, 1430.
- Goddard, N. H., Hucka, M., Howell, F., Cornelis, H., Shankar, K., and Beeman, D. (2001). Towards NeuroML: model description methods for collaborative modelling in neuroscience. *Philos. Trans. R. Soc. Lond., B, Biol. Sci.* 356, 1209–1228.
- Hines, M. L., Morse, T., Migliore, M., Carnevale, N. T., and Shepherd, G. M. (2004). ModelDB: a database to support computational neuroscience. *J. Comput. Neurosci.* 17, 7–11.
- Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, L., Woolley, C., and Lefohn, A. (2004). GPGPU: General Purpose Computation on Graphics Hardware. Los Angeles, CA, ACM, p. 33.
- MacGregor, R. J. (1987). *Neural and Brain Modeling*. Academic Press, San Diego.
- Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biol. Cybern.* 98, 459–478. PMID: 18491160
- Morrison, A., Straube, S., Plesser, H. E., and Diesmann, M. (2007). Exact sub-threshold integration with continuous spike times in discrete-time neural network simulations. *Neural Comput.* 19, 47–79.
- Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nat. Neurosci.* 3, 919–926.
- Tsodyks, M. V., and Markram, H. (1997). The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proc. Natl. Acad. Sci. U.S.A.* 94, 719–723.
- Vogels, T. P., and Abbott, L. F. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci.* 25, 10786–10795.

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

Received: 11 September 2008; paper pending published: 30 September 2008; accepted: 26 October 2008; published online: 18 November 2008

Citation: Goodman D and Brette R (2008) Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* (2008) 2:5. doi: 10.3389/neuro.11.005.2008
Copyright © 2008 Goodman and Brette. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.

Vision Egg: an open-source library for realtime visual stimulus generation

Andrew D. Straw*

Bioengineering, California Institute of Technology, Pasadena, CA, USA

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

Reviewed by: Laurent Perrinet, INCM-CNRS, France
Jonathan W. Peirce, University of Nottingham, UK

Modern computer hardware makes it possible to produce visual stimuli in ways not previously possible. Arbitrary scenes, from traditional sinusoidal gratings to naturalistic 3D scenes can now be specified on a frame-by-frame basis in realtime. A programming library called the Vision Egg that aims to make it easy to take advantage of these innovations. The Vision Egg is a free, open-source library making use of OpenGL and written in the high-level language Python with extensions in C. Careful attention has been paid to the issues of luminance and temporal calibration, and several interfacing techniques to input devices such as mice, movement tracking systems, and digital triggers are discussed. Together, these make the Vision Egg suitable for many psychophysical, electrophysiological, and behavioral experiments. This software is available for free download at visionegg.org.

Keywords: visual stimulus generation, open source, Python

INTRODUCTION

A neuroscientist may need precisely defined spatial, temporal, spectral, and polarization properties of light to perform a particular visual experiment. Standard computer monitors and projectors are capable of producing a wide range of stimuli sufficient for many experiments, and special purpose displays may be built or purchased with a standard interface. A tool which produces precisely controlled signals from a video port (such as VGA) is therefore of great utility. This paper outlines the Vision Egg, a programming library developed to serve as such a tool in combination with a standard computer and other software libraries.

HISTORICAL CONTEXT

A brief outline of the display systems with the most impact on the design of the Vision Egg follows.

In the 1980s and 1990s, vision scientists frequently displayed their stimuli on a Tec-tronix 608 display, a small (~12 cm diagonal) cathode ray tube with independent X,Y and luminance inputs originally intended for use in a high-bandwidth analog oscilloscope. However, instead of using it as an oscilloscope display, vision scientists often controlled the 608 with an Innisfree Picasso device, a specialized function generator that creates

a raster scan of X,Y positions and modulates luminance to produce a variety of simple stimuli such as sinusoidal gratings and rectangles. Many scientists found the Picasso wonderfully easy to use, as its intuitive interface with a myriad of switches and potentiometers allowed rapid experimentation until a suitable stimulus was found. Furthermore, by providing BNC connections for voltage inputs, time-varying stimuli could be driven via analog outputs from the same data acquisition system being used to record responses, simplifying experimental design. The main limitations of the Picasso are essential to its design as a specialized function generator – namely that it is tied to a specific (and now rare) display device, and that the range of stimuli it could produce were limited.

Computers provide the ability to produce arbitrary visual stimuli, but with a new set of limitations. Early systems developed in the 1990s required no specialized hardware but could only draw pre-rendered stimuli and movies (e.g., early releases of the PsychToolbox: Brainard, 1997; Pelli, 1997) or were limited to simple stimuli and required extensive programming and debugging in low-level C (e.g., John Maunsell's custom LabLib). These systems achieved frame-by-frame temporal precision by operating within a cooperative multitasking operating system such as Mac OS (prior to Mac OS X) and running at interrupt time. Under such conditions, the underlying OS would not preempt a program's use of the CPU or other resources. With the rise of pre-emptive multitasking operating systems such as Windows 95, GNU/Linux, and Mac OS X, such an approach to precise timing was no longer guaranteed. Another issue, which persists today, is that the general-purpose nature of display hardware meant that producing stimuli with a large dynamic range of contrast can be difficult.

Custom hardware solutions, such as the Cambridge Research Systems' VSG 2/3F, addressed the issues of precise timing and

*Correspondence: Andrew D. Straw, Bioengineering, California Institute of Technology, 1200 E. California Boulevard, Mail Code 138-78, Pasadena, CA 91125, USA. e-mail: astraw@caltech.edu

Received: 15 September 2008; paper pending published: 26 September 2008; accepted: 08 October 2008; published online: 04 November 2008.

Citation: *Front. Neuroinform.* (2008) 2: 4. doi: 10.3389/neuro.11.004.2008

Copyright © 2008 Straw. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.

dynamic range through the use of special purpose processing units and digital to analog converters isolated from the main computer system on a PCI card. Programs would execute onboard these cards independently from the host operating system, bypassing the issues outlined above. Such cards were expensive, however, often costing five or more times the price of the host computer itself, with additional RAM costing still more. Additionally, programming the VSG 2/3F involved either using a script language with limited performance or a low-level, assembly-like language specific to the processing unit onboard the card.

By the year 2000, OpenGL, a library to abstract standard graphics hardware, was being used for realtime generation of 3D graphics on broadcast television without skipping frames. I was encouraged to try a similar approach for my own experiments on the visual system of flies, where the ability to use 3D video acceleration hardware was appealing because it meant that wide-field stimuli could be accurate across displays subtending very large angles. Such graphics hardware was appealing more generally for vision research because this hardware was very fast at mathematical operations involved in drawing scenes while the open nature of the OpenGL specification meant that solutions would be portable to future hardware. The high speed allowed new possibilities for the display of visual stimuli that change over time. Dynamic scenes of high complexity, including in 3D, could be rendered in realtime, only an instant before display. This could be done at high update rates without skipping frames, and these video cards could display anything from simple shapes to naturalistic 3D scenes. The immediate benefit for my research was to enable drawing at 200 Hz of perspective-corrected Gabor wavelets (Straw et al., 2006) and temporally anti-aliased (so called *motion blurred*) moving natural images (Straw et al., 2008). Both of these types of stimuli had been very difficult to implement with the other systems.

OPEN SOURCE SOFTWARE AND PYTHON

Fundamental to the scientific process is the repeatability of measurements. For this reason, open source software should be preferred in scientific applications – this prevents software mistakes from becoming hidden in proprietary code, allows others to learn from and independently reproduce work, and allows a community approach to solve problems together. As illustrated by the articles in this issue, Python is becoming a standard high-level, open source language in neuroscience. Perhaps the most exciting aspect of the confluence of tools available in Python is the possibility of software that incorporates components from various sources into software with new capabilities. The suitability of Python for drawing visual stimuli is well described in Peirce (2007), and additional notes are in Section “Timing of Visual Stimuli: Speed and Latency.” The Vision Egg also makes use of software for which no Python interface previously existed. These function calls are written as C extension modules to Python included with the Vision Egg.

VISION EGG

The aim of this paper is to describe the Vision Egg, an open source (LGPL license) computer programming library which makes use of modern hardware accelerated graphics using OpenGL to generate visual stimuli. One important goal for the project is to allow non-experts to use modern computer hardware to its maximum capability for common vision science tasks. A screenshot of an included demonstration script showing several of the visual stimulus possibilities is shown in Figure 1, and source code to a moving sinusoidal grating is shown in Figure 2.

At the initial development and release of the Vision Egg in 2001–2002, existing software for vision scientists was not able to take advantage of the capabilities present in the emerging hardware standards. Now, almost every personal computer being sold is equipped with graphics hardware suitable for many

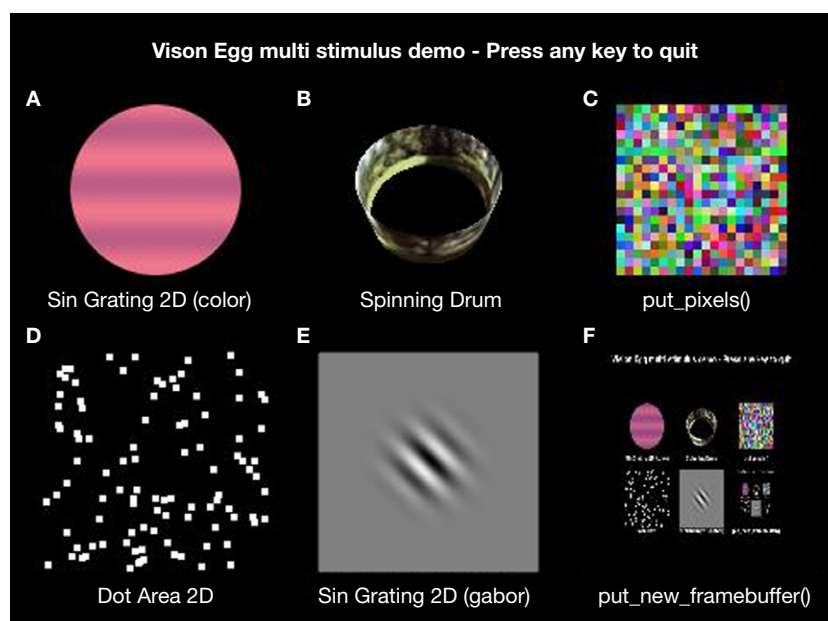


Figure 1 | **Screenshot of Vision Egg `multi_stim.py` demonstration script showing several included visual stimulus types.** The dynamic stimuli are updated in realtime without skipping frames at rates up to the fastest vertical refresh rate of the display tested (200 Hz). Stimuli are: **(A)** A circularly windowed color grating changing in space and color over time. **(B)** A rotating, perspective distorted drum with a natural panorama used as a texture image. **(C)** Arbitrary arrays of RGB data updated on each frame generated from a uniform random distribution. **(D)** Random dot stimuli with 100 independently moving dots. **(E)** A drifting Gaussian windowed sinusoidal grating. **(F)** A copy of the framebuffer recursively redrawn at smaller scale.



```

A # Import modules
import VisionEgg
VisionEgg.start_default_logging(); VisionEgg.watch_exceptions()

from VisionEgg.Core import get_default_screen, Viewport
from VisionEgg.FlowControl import Presentation
from VisionEgg.Gratings import SinGrating2D

# Initialize OpenGL window/screen
screen = get_default_screen()

# Create sinusoidal grating object
stimulus = SinGrating2D(spatial_freq = 10.0 / screen.size[0],
                        temporal_freq_hz = 1.0,
                        )

# Create viewport - intermediary between stimuli and screen
viewport = Viewport( screen=screen, stimuli=[stimulus] )

# Create presentation object and go
p = Presentation(go_duration=(5.0,'seconds'),viewports=[viewport])
p.go()

B # Import modules
import VisionEgg
VisionEgg.start_default_logging(); VisionEgg.watch_exceptions()

from VisionEgg.Core import get_default_screen, Viewport, \
    swap_buffers
from VisionEgg.Gratings import SinGrating2D

# Initialize OpenGL window/screen
screen = get_default_screen()

# Create sinusoidal grating object
stimulus = SinGrating2D(spatial_freq = 10.0 / screen.size[0],
                        temporal_freq_hz = 1.0,
                        )

# Create viewport - intermediary between stimuli and screen
viewport = Viewport( screen=screen, stimuli=[stimulus] )

# Use our own main loop
tstart = VisionEgg.time_func()
while (VisionEgg.time_func() - tstart) <= 5.0:
    screen.clear()
    viewport.draw()
    swap_buffers()

```

Figure 2 | **Source code of simple Vision Egg program to draw a moving sinusoidal grating illustrating a simple but complete program.** Two means of controlling the flow of execution are available, as described in Section “Mid-level Software Overview: Controlling Program Flow.” **(A)** Program flow is controlled by the Vision Egg’s Presentation class. **(B)** Program flow is explicitly specified within the script.

experiments. Although more expensive hardware, often designed with computer games in mind, continues to push the limits of performance, the modest graphics systems now found in laptops and some motherboards perform fine for many experimental purposes. Even the creation of artificially closed-loop “virtual-reality” experiments with the Vision Egg is possible with relatively inexpensive hardware (e.g., Fry et al., 2004, 2008) but the library is also useful for a variety of simpler tasks.

The biggest challenge with such an approach is addressing potential problems when attempting to produce precisely controlled stimuli for visual science on hardware which was not explicitly designed for the task. The remainder of this paper describes the implementation of the Vision Egg, some experiments to characterize its performance, a discussion of it in relation to other visual stimulus technologies, and some potential future directions.

LOW-LEVEL HARDWARE AND SOFTWARE OVERVIEW

HARDWARE

This section presents a brief review of modern computer architecture from a hardware perspective for drawing visual stimuli. Applications run on the CPU of the host computer, though which they manipulate the memory, video system, and other devices of the computer. Video cards have onboard graphics processors (GPUs) that are faster than CPUs at pushing pixels. By shifting the majority of the drawing work onto the video card, the role of the CPU can be limited to directing the powerful GPU. To render a complicated 3D scene, for example, the CPU computes a wireframe model that is transmitted, along with rasterization instructions such as texture images and coordinates, to the video card. This communication is specified by OpenGL, which hides the hardware level details such as transmission of data across the computer bus. The GPU renders this image to a framebuffer, which is then read out either by a high-speed digital to analog converter (RAMDAC) or a digital transmitter (e.g., DVI, HDMI, and Display Port). Luminance and color information is limited in typical framebuffers because they store 8 bits per color per pixel, or 256^3 values of red, green, and blue each for a total of 256^3 (16.6

million) possible colors. The RAMDAC converts these digital values to an analog voltage after passing them through a color lookup table, which can be used to correct non-linearities the display process such as *gamma* (see Section “Precise Control of Color and Luminance: Results of Luminance Calibration”). Recently, manufacturers have been increasing the precision of the lookup tables in the RAMDAC, and although many 8 bit per color RAMDACs are still available, 10 bit cards are becoming more common. Furthermore, some higher-end cards have 10 bit framebuffers.

DRAWING IN OpenGL

The Vision Egg scripts enter a loop which draws a new frame on each cycle. Often each frame can be drawn completely from scratch, allowing realtime control of stimuli or simply to eliminate a common brute force approach of pre-rendering several frames and then displaying them sequentially. Furthermore, the frame skips do not lead to cumulative error if each frame is drawn in realtime based on an accurate clock time. In an OpenGL system, a double buffering technique is used, meaning that new frames are rendered to the back framebuffer while the RAMDAC draws the contents of the front buffer to the display. Due to this double buffering, partially completed frames are not drawn to the screen. When finished rendering to the back framebuffer, the application informs the graphics system to use the back buffer as the source of data for the RAMDAC. Thus, the front and back buffers are swapped (with an OpenGL `flip()` or Vision Egg `swap_buffers()` function call) and drawing continues on the new back buffer. In the so-called *vsync* (vertical sync) mode, the buffer swap is synchronized to occur only between frame draws by the display, and thus no “tearing” artifacts are present. With small displacements between individual frames, however, tearing is minimal without using vertical sync. Regardless of vsync mode, the main loop OpenGL delays execution of the program until the buffer swap command is sent to the video hardware.

A member of the Vision Egg community has performed extensive testing on the latencies associated with drawing in OpenGL (Sol Simpson, SR Research, personal communication), which are

in agreement with my personal observations and more limited testing. His tests show that even with vsync on, the actual call to `swap_buffers()` acts in an asynchronous manner when no buffer swaps are pending, but begins blocking when another swap is scheduled. In other words, the first call to `swap_buffers()` will return immediately and the graphics card is instructed to swap buffers during the next vertical retrace. However, if another call to `swap_buffers()` is issued before the retrace occurs, this call is blocked (does not return) until the first scheduled buffer swap happens. Thus, a program which paces itself via returning from blocked calls to `swap_buffers()` will always be drawing frames which will be drawn not on the next buffer swap, but on the second buffer swap.

Thus, if a program calls `swap_buffers()` less than once per retrace interval, then the `swap_buffers()` call is not blocked and returns right away and not necessarily at the start of a retrace. In this case, one does not see a constant 1 retrace interval delay. Instead, one will see a variable delay (the time between when `swap_buffers()` returns and when the display is actually updated), with a duration up to the retrace interval depending on when `swap_buffers()` was called.

This suggests that one cannot not rely on when `swap_buffers()` returns to determine when the flip actually occurs and instead should use a combination of `swap_buffers()` followed by some code that actually waits until, or determines, the start of the next retrace. The Vision Egg currently provides such a function for Windows (see Section “Low-level Hardware and Software Overview: Detecting Retrace Events and Refresh Rates”). The same results are found with the Vision Egg, pure C OpenGL and with SDL when using the DirectX backend on ATI and nVIDIA graphics cards (Sol Simpson, SR Research, personal communication).

Due to the intricacies of the above latency issue when vsync is on and the lack of a way to detect retrace events on all supported platforms, the Vision Egg currently (up to and including 1.1.1) simply assumes that frames are drawn when `swap_buffers()` returns. This gives an accurate estimate of whether refresh intervals were skipped and consequently a frame was not updated, but results in latency increased by one refresh interval.

Recent video cards (e.g., nVIDIA GeForce 8500 GT with the Forceware version 163.71 driver on Windows XP) support “triple buffering.” In this mode, there are two back buffers that are alternately drawn upon, and the most recently completed buffer is used at the start of display of a new frame to the screen. Although I have not tested this technique, it theoretically allows near-minimal latencies without tearing artifacts or difficult programming involving refresh detection.

OPERATING SYSTEMS

The Vision Egg runs on any platform which supports Python and OpenGL. It is known to run on Microsoft Windows (95, 2000, and XP), GNU/Linux with kernels 2.4 and 2.6 (Ubuntu, Redhat, Debian), Mac OS X and SGI IRIX. All of these are preemptive multitasking operating systems, with important ramifications described in section “Timing of Visual Stimuli: Speed and Latency.”

DETECTING RETRACE EVENTS AND REFRESH RATES

The Vision Egg offers some platform-dependent features. One of these is the ability to detect or wait for a vertical retrace event. This is implemented according to the method of Riemersma (2000) and implemented in the `Win32_vretrace.pyx` file. Furthermore, the refresh rate can be detected on Windows and

Mac OS X as implemented in the `win32_getrefresh.c` and `darwin_getrefresh.m` files. Unfortunately, the Vision Egg does not currently allow the user to set the refresh rate.

MAXIMUM PRIORITY MODE

Operating systems typically have means to boost the priority of some processes above that of other processes. The details are specific to each platform, but the Vision Egg includes support for raising priority on Windows via the `SetPriorityClass()` and `SetThreadPriority()` functions, on POSIX systems (such as Linux) via the `sched_setscheduler()` and `mlockall()` functions, and on Mac OS X via the `thread_policy_set()`, `setpriority()` and `pthread_setschedparam()` functions. On Mac OS X, these function calls tell the kernel's realtime scheduler to grant programs a periodic time slice from the CPU, which theoretically might give hard realtime performance (guaranteed latency), but practically is limited by the issues described in Section “Timing of Visual Stimuli: Speed and Latency.”

MID-LEVEL SOFTWARE OVERVIEW

DISPLAY OF STIMULI

The Vision Egg has methods to draw a wide variety of stimulus types. These stimuli operate within defined guidelines so that they only modify certain values of the OpenGL state machine, but leave all other values unchanged. In this way, multiple stimuli can be combined simultaneously, as in Figure 1. Both 2D and 3D stimuli are available. 2D stimuli commonly use an orthographic projection such that coordinates are specified in pixel units. Perspective projections can be used for 3D stimuli such that a calibrated projection will provide an accurate representation of object shapes when viewed on a flat display (e.g., Kern et al., 2001; Straw et al., 2006). Included with the Vision Egg are routines for drawing luminance sinusoidal gratings (2D or 3D, with or without contrast windows, which can be circular or anisotropic Gaussian in shape), color sinusoidal gratings, random dot stimuli, arbitrary image files, arbitrary numeric array data, QuickTime movies, MPEG movies, a spinning 3D drum with a textured image, rectangles and fixation points.

Many features of OpenGL are supported, including realtime resampling of the texture image data using linear interpolation and use of mipmapped textures generated with bicubic interpolation (or other means). These features allow display of slowly moving images without quantization of other systems where pixel-by-pixel steps must be made in integer multiples of the inter-frame interval. Other features, such as realtime lighting and shadows, are not currently implemented.

USER INTERACTION AND ALTERNATIVE SOURCES OF INPUT

User interaction, such as handling of keystrokes, mouse clicks, and joysticks can occur within the main loop of a Vision Egg program by using the pygame library. Additionally, because the Vision Egg is written in Python and can be easily extended with C, there are many potential sources of external input. For example, the UDP network protocol is frequently used in online computer games for low latency network communication and can be used for realtime control of visual stimuli from an external program. In this manner, a Vision Egg script may be written which is controlled from a data acquisition environment written in Python, Lab View, or MATLAB. The TCP network protocol, although slower than UDP, offers built-in error checking and correction, and has been used to provide realtime input for the Vision Egg (Fry et al., 2004, 2008).



CONTROLLING PROGRAM FLOW

The Vision Egg offers two ways of program flow control. The most conceptually simple of these is to let the programmer specify what happens on every frame, as illustrated in **Figure 2B**.

Because the Vision Egg was originally developed for studies in which controlling motion adaptation was critical, I paid careful attention to issues such as allowing a stimulus to continue moving while not in an experimental trial. The result is the programmer relinquishes control by entering the `go()` method of the `Presentation` class, as defined in the `VisionEgg.FlowControl` module, as in **Figure 2A**. This is the concept of a *go loop*, which usually corresponds to the experimental trial, and the concept of refreshing stimuli *between go loops*. Any function calls or stimulus updates not automatically performed by the Vision Egg must be implemented by means of `Controllers`, which are implementations of callback functions. Such a *main-loop-and-callback* style of programming is common in GUI programming. For example, the WX Widgets toolkit and the Mac OS X Cocoa libraries operate this way.

HIGH-LEVEL SOFTWARE OVERVIEW

SPECIFYING GRAPHICS STATE

A configuration GUI (**Figure 3**) can optionally be called at the beginning of any Vision Egg script. Although all options are available from the programmatic interface, it is often convenient to see and edit these parameters through this interface. Particularly important are the options for loading the color

lookup tables to perform gamma correction as illustrated in Section “Precise Control of Color and Luminance: Results of Luminance Calibration.”

AN APPLICATION FOR ELECTROPHYSIOLOGY

The Vision Egg includes two applications for integration within an electrophysiology environment (see **Figure 4**). The first is `ephys_server.py`, which draws stimuli on its video hardware. To minimize the possibility of frame skipping, this program may run as the sole application on a dedicated stimulus computer. This server program listens on a network port for a connection from the `ephys_gui.pyw` program, which offers a GUI for the experimenter to control.

THE QUEST ALGORITHM

A pure Python implementation of Watson and Pelli’s (1983) QUEST algorithm is available from the Vision Egg website. This well-known Bayesian adaptive method allows estimating psychometric thresholds, and was translated directly from the MATLAB code of Denis G. Pelli, who graciously allowed redistribution of the Python version under an open-source BSD license.

QuickTime AND MPEG MOVIES

The Vision Egg includes support to decode movies and send them to OpenGL by using Apple’s QuickTime API on Windows and Mac OS X and `pygame/SDL`’s `Movie` objects on all supported operating systems.

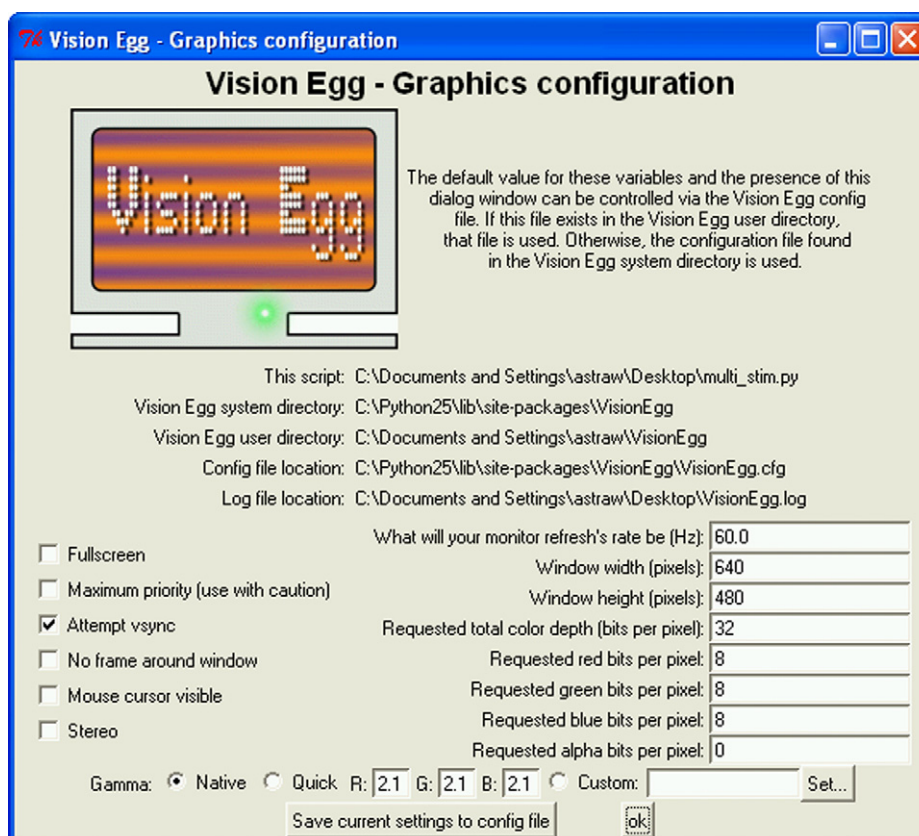


Figure 3 | Screenshot of the standard Vision Egg configuration GUI. Numerous options for configuration are available, including framebuffer size and bit depth, color lookup tables for gamma correction and platform-dependent realtime priority, as described in Section “High-level Software Overview: Specifying Graphics State.”

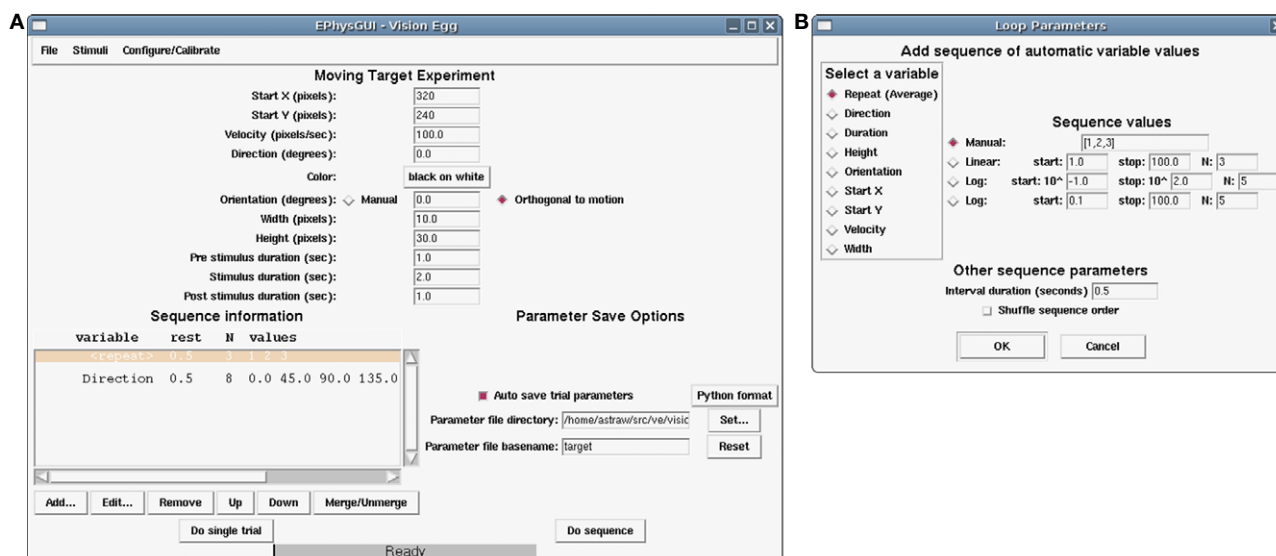


Figure 4 | Screenshot of electrophysiology-oriented GUI application included with the Vision Egg called `ephys.gui.pyw` and described in Section “High-level Software Overview: An Application for Electrophysiology.” (A) Main window shows parameters for repeated presentations of a stimulus with the possibility of automatically sequencing over variables. All settings can be saved and loaded from disk. (B) The loop parameters window allows control of experiments.

TIMING OF VISUAL STIMULI

METHODS TO MEASURE LATENCY

This section contains the results of experiments in which the total latency of the system, from input to output, was measured. Because it is difficult to measure the precise time of events happening inside and outside a computer on the same clock (or synchronized clocks), a task was chosen in which only a single time reference was necessary. The task was to measure the duration for a USB mouse movement to be translated into the movement of a rectangle drawn on the screen, both of which were filmed with a high speed video camera and later analyzed. The latencies measured in this task should be comparable to the latencies of other input–output tasks.

An LED was rigidly fixed to each computer mouse (Logitech MX-300 USB and Dell DEL1 Optical USB). The mouse was connected to a USB port on the motherboard of the computer (Acer Aspire T690 with Intel ICH7 chipset including USB2 EHCI and USB UHCI controllers). A PCI-Express xl6 video card (nVIDIA GeForce 8500 GT) was connected to a CRT monitor (Iiyama Vision Master 450) using a VGA cable. The display was set to a resolution of 800×600 at 140 Hz update rate using the nVIDIA control panel (Forceware version 163.71) on Windows XP Service Pack 2 and confirmed by using the monitor’s on screen display.

The Vision Egg version 1.1.1 was used to draw a 3×3 pixel white square using the `Target2D` class on a `Screen` with a black background color in a way that it acted as a mouse cursor. The position of the mouse controlled the position of this small square using a version of the `mouseTarget.py` demo program that was simplified to remove the code that set the orientation of the target.

A high speed digital video camera (Photron Fastcam APX 120) was placed to record the LED and target location on the screen in the same image frame. Images were acquired at 2000 frames per second while the mouse was rapidly moved back and forth by hand in a roughly sinusoidal manner (e.g., Figure 5).

Digital images were analyzed to identify the “center of mass” of the bright areas using the `center_of_mass()` function of the `scipy.ndimage` module. For the on-screen target, this only occurred approximately every 14th frame due to the discrete nature of raster scan CRT displays.

SPEED AND LATENCY

Because Python is an interpreted language, programs written in it will run more slowly than a well-written C program. However, Python is fast enough for two primary reasons. First, the most computation-intensive task, manipulation of large data arrays is performed with high-performance C and FORTRAN code via the `numpy` module of Python. Thus, Python code directs computationally intensive tasks without performing them in the slower interpreted environment. Second, computer displays cannot be refreshed beyond their maximum vertical frequency, which typically ranges up to 200 Hz. This therefore represents an upper bound on the amount of computation required for realtime rendering tasks.

In fact, the biggest timing-related concern is unrelated to the programming language used. A pre-emptive multitasking operating system may take control of the CPU from the stimulus generating program for periods longer than an inter-frame interval, thus leading to skipped frames. Even if the OS takes control of the CPU from an application for much less than an inter-frame interval, frames may still be skipped if the stimulus generation program uses a strategy of waiting until the last instant to render a frame and CPU control is taken at this critical instant. Operating systems may have some means addressing this issue such as a realtime scheduler that guarantees uninterrupted CPU time at specified intervals. The Vision Egg makes use of such facilities where available (see Section “Low-level Hardware and Software Overview: Maximum Priority Mode”). Although they can certainly help eliminate timing issues, such priority-boosting solutions cannot provide absolute guarantees about timing because OpenGL implementations themselves may be subject to unpredictable



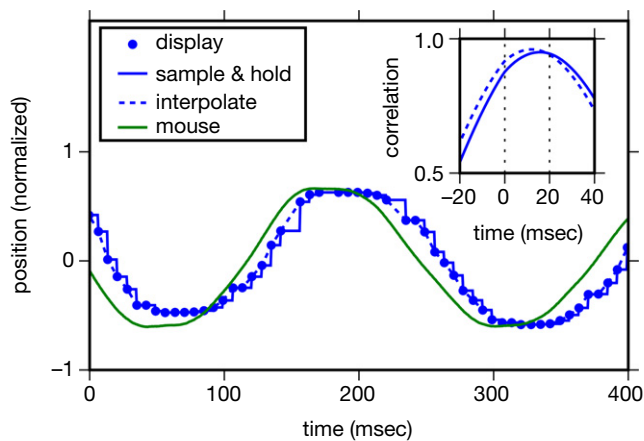


Figure 5 | Total latency of system, including input from an optical USB mouse and display on 140 Hz CRT display, can be reduced to about 15 ms, as described in Section “Timing of Visual Stimuli: Measurements of Latency.” The main panel shows representative data gathered from a high speed camera of an LED fixed to a mouse (green line) and a bright spot on the screen controlled by the mouse (blue dots). Display positions could reasonably be interpolated using a sample-and-hold function (blue solid line) or linear interpolation (blue dashed line). Inset panel shows cross correlation of 2 s of such data when interpolated. These data were gathered with vsync off and a Logitech MX-300 USB mouse.

behavior and generally are not written to operate in a *hard realtime* (in other words, with deterministic latency) manner. For example, drawing a single additional object may cause the hardware to pass a critical threshold for memory use and force a slow operation. A low-level solution which operated in hard realtime would have to bypass complex OpenGL libraries and implement routines to draw directly to the framebuffer to guarantee performance.

It is worth noting that because of this unavoidable variable latency listed above (pre-emptive multitasking operating systems and OpenGL implementations), the variable latency introduced by use of an interpreted language with garbage collection, such as Python, does not fundamentally worsen the situation. In other words, use of Python introduces no fundamental problem other than that of an additional potential source of variable latency to that already imposed by the OS and OpenGL.

MEASUREMENTS OF LATENCY

A high speed video camera was used to measure the absolute total latency between input (a standard USB computer mouse) and output (the position of a rectangle on the screen), as described in the methods Section “Timing of Visual Stimuli: Methods to Measure Latency.” **Figure 5** shows that latency can be reduced to around 15 ms, but that the vsync state plays a very significant role in total latency (**Table 1**). On a 140-Hz display (7.1 ms inter-frame interval), latency jumped by 17 ms or more when vsync was enabled. This is presumably due to the latency imposed by drawing in the middle of a refresh interval and waiting for that interval to be done combined with the additional latency described in Section “Low-level Hardware and Software Overview: Drawing in OpenGL.” Although the results would be interesting, these experiments were not repeated in triple buffering mode.

ONLINE DETECTION OF FRAME SKIPPING

Frame skipping is determined by measuring the interval between successive buffer swap commands using standard system calls to

Table 1 | Latency as estimated by the peak of the cross correlation between mouse location and displayed point location. Optimistic latencies were estimated using the cross correlation with the linearly interpolated display positions as plotted in **Figure 5** and described in Section “Timing of Visual Stimuli: Measurements of Latency.” Pessimistic latencies were also estimated with a cross correlation, but used a sample-and-hold function rather than linear interpolation to estimate display position.

Vsync	Mouse	Optimistic latency (ms)	Pessimistic latency (ms)
Off	Logitech MX-300 Optical USB	12.0	16.0
On	Logitech MX-300 Optical USB	35.0	38.5
Off	Dell DEL1 Optical USB	19.5	24.5
On	Dell DEL1 Optical USB	38.0	41.5

query the computers clock. If this value exceeds the known monitor inter-frame interval, a frame has been skipped. Stimuli generated by the Vision Egg are routinely presented for hours without skipping a frame when measured this way. The most likely occurrence of a skipped frame is at the immediate beginning of drawing a stimulus – presumably when some initialization occurs with the video system. Often this can be dealt with by initializing the video system in a non-critical task, such as drawing a black rectangle.

TRIGGER OUTPUT AND INPUT

It is often useful to trigger external hardware when a stimulus presentation begins. There are several ways to achieve this on typical personal computers. The parallel port can be used so that a pin goes from low to high voltage when the first frame of a stimulus is drawn. The Vision Egg has support for reading and writing to the parallel port, but because OpenGL operates in an asynchronous manner (see Section “Low-level Hardware and Software Overview: Drawing in OpenGL”), the parallel port cannot be updated at the exact instant the display begins a new frame. Instead, the parallel port can only be updated before the `swap_buffers()` command is given or after it returns. Better accuracy could be obtained by “arming” the trigger of a data acquisition device immediately before stimulus onset and triggering from the vertical sync pulse of a video cable. Ultimate verification can be done with a photodetector on a patch of screen that changes luminance at the onset of the experiment. This patch-of-screen is implemented in the `ephys_gui.pyw` application described in Section “High-level Software Overview: An Application for Electrophysiology.”

Some hardware used in experiments, such as fMRI machines, has intrinsic timing requirements and thus it is advantageous for the Vision Egg to act as a slave and to begin a stimulus upon receiving a digital pulse. Because of its realtime nature, it is straightforward to achieve temporal precision equivalent to the latencies described in Section “Timing of Visual Stimuli: Measurements of Latency,” although there might be slight differences in timing due to use of a parallel port for input rather than a USB mouse.

PRECISE CONTROL OF COLOR AND LUMINANCE METHODS TO MEASURE LUMINANCE

For the measurements described below, the Vision Egg version 1.0 was running on a dual Athlon 1400 Windows 2000 system with an nVIDIA GeForce 4 Ti 4200 graphics card and an

LG Electronics Flatron 915 FT + CRT monitor at a resolution of 640×200 at 200 Hz. Luminance measurements were made with a silicon photometer (OptiCal with LightScan software by Cambridge Research Systems, Ltd).

RESULTS OF LUMINANCE CALIBRATION

An 8-bit per color framebuffer allows specification of 256 luminance levels for each of the three color channels (see Section “Low-level Hardware and Software Overview: Hardware”). Each red, green, and blue value is used as an index into the appropriate color lookup table, which is used by the RAMDAC to produce an analog signal. Low contrasts or other effects may be achieved, even with an 8-bit per color framebuffer, by use of a 10-bit lookup table. Non-linearities of CRT displays are well understood (for review, see Brainard et al., 2002) with the most famous non-linearity being display luminance *gamma*. The lookup tables can compensate for this gamma property such that color specified is linearly proportional to the luminance produced on the display, and the Vision Egg includes the ability to calibrate and compensate automatically for this gamma property. Finally, some computers have framebuffers with >8 bits per color. In OpenGL and the Vision Egg, colors are specified as a floating point value between 0.0 and 1.0 so the same program benefits immediately from the improved hardware.

Photometric luminance measurements of the display made with full screen color values are shown in Figure 6. The most well known of the non-linearities of video displays is characterized by the gamma function

$$L = kp^{\gamma}, \quad (1)$$

with L being luminance (in cd/m^2), k being a scaling constant, p being the color value specified to OpenGL for each of the red,

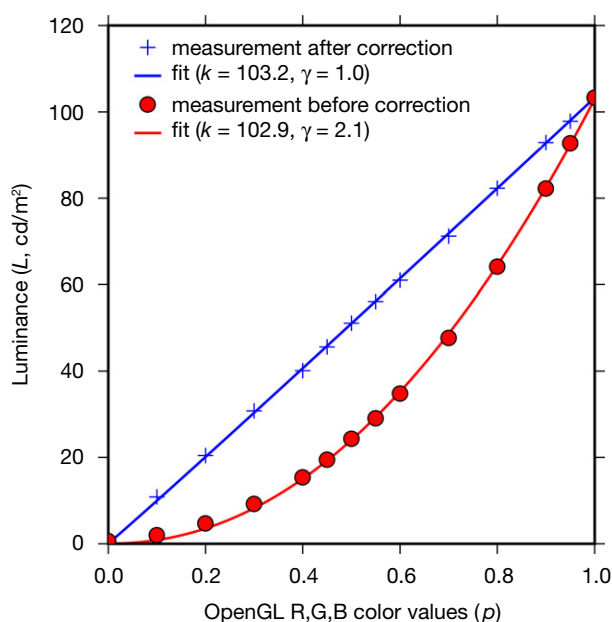


Figure 6 | **Luminance output of a CRT display is made linear with respect to commanded pixel value.** Color values specified in OpenGL units produce non-linear luminance relationship on an uncorrected display (red circles), but a corrected display has a linear relationship between specified and actual luminance (blue crosses). Lines are linear least squares fits to Eq. 1, Section “Precise Control of Color and Luminance: Results of Luminance Calibration,” with coefficients given in the legend.

green, and blue components of the screen, and gamma γ . In the example shown, the uncalibrated display system had $\gamma = 2.1$. By loading the appropriate values in the color lookup tables, a linear relation between specified color value and luminance output was achieved, with $\gamma = 1.0$.

DISCUSSION

IMPACT OF THE VISION EGG

Although usage for open source software is notoriously difficult to estimate, the number of downloads of the Vision Egg from SourceForge.net since the first release (November 2001) totals over 15,000. Another estimate is the number of papers citing use of the Vision Egg. To date, the total listed at the website is 14. The University of Bielefeld, Germany and the University of Adelaide, Australia have used the Vision Egg in undergraduate courses (Bart Geurten and David O’Carroll, personal communication).

Other software uses or incorporates the Vision Egg. For example, in this issue, (Spacek and Swindale, 2008), describe use of the Vision Egg as part of a system for high-throughput electrophysiology. Python based extensions called *BCPy2000* to the large project *BCI2000*, a general-purpose system for brain-computer interface (BCI) research, allow customizable experiment design using the Python scripting language (Schreiner, 2008; Jeremy Hill, personal communication). SR Research developed Pylink to interface their eye tracker to Python-based software, such as the Vision Egg, and they ship a Vision Egg based example to demonstrate gaze contingent control of a moving gradient.

Finally, perhaps the greatest impact of software packages such as the Vision Egg has simply been as a proof of concept that using OpenGL and Python for creating visual stimuli is possible. Several people have told me that they looked at the Vision Egg to see how something was done and then re-implemented it themselves. Such a spread of ideas is one of the benefits of open source, although the diversity of similar but different solutions can also be a challenge, particularly for those attempting to pick a solution without investing too much in an evaluation process.

COMPARISON TO SIMILAR OPEN SOURCE SOFTWARE

PsychoPy is another Python-based open source visual stimulus system (BSD license). The author, Peirce (2007) says, “For a good programmer, Vision Egg achieves its goals very well, providing a powerful and highly optimized system for visual stimulus presentation and interactions with hardware (including the ability to run experiments remotely across a network). Straw does, however, adhere very strongly to an object-oriented model of programming which can be harder for relatively inexperienced programmers, like most scientists, to understand. For instance, the temporal control of experiments in Vision Egg is predominantly though the use of presentation loops, whereby the user sets an object to run for a given length of time, attaches stimuli to it, attaches it to a screen and then tells it to go.” I believe the criticism is directed not so much toward object oriented programming (which is also employed at a fundamental level within PsychoPy) but rather Peirce’s concern is with the mainloop-and-callback mechanism of flow control described in Section “Mid-level Software Overview: Controlling Program Flow.” As mentioned in that section, and demonstrated in Figure 2, this is only optional, and the user may also maintain full control of program execution. Nevertheless, in the early development of the Vision Egg, this mainloop-and-callback style was present in all the demonstration scripts, and was intrinsic to the electrophysiology



applications envisioned, like that of Section “High-level Software View: An Application for Electrophysiology.” Indeed, it was a response to my own difficulties implementing psychophysics experiments with this style that I wrote demo scripts with their own flow control and began documenting the possibility.

Apart from the differences mentioned above in the style of programming, the most substantive differences today between the Vision Egg and PsychoPy are that the Vision Egg offers relatively simple perspective corrected stimuli utilizing the 3D nature of OpenGL, while PsychoPy has an automated luminance calibration utility and interfaces with Bits++ from Cambridge Research Systems, Ltd. Furthermore, the primary development platform of the Vision Egg is GNU/Linux, while it appears to be Windows for PsychoPy.

The Psychophysics Toolbox (Brainard, 1997; Pelli, 1997) has evolved greatly since the situation described in Section “Introduction: Historical Context.” There is a large overlap between the possibilities offered by the PsychToolbox and the Vision Egg. Although the PsychToolbox is now officially open source (GNU GPL license), the main language of implementation is MATLAB, a proprietary application. Thus, its appeal as an open source solution is limited. Nevertheless, a core developer, Mario Kleiner, tests PsychToolbox functions with Octave, an open-source MATLAB clone, and many useful functions are implemented in C and could be used from environments other than MATLAB. Due to its heritage, most of the demonstration scripts for the PsychToolbox use pre-rendered stimuli, but it is now capable of using OpenGL and generating complex stimuli in realtime.

For another comparison between Vision Egg, PsychoPy, and the PsychToolbox, see Peirce (2007).

TOWARD A DATABASE OF VISUAL STIMULI

An online database of scripts to generate stimuli used in visual neuroscience would be useful for realizing the benefits of open-source software described in Section “Introduction: Open Source Software and Python.” Other databases, such as of neuronal models (e.g., ModelDB and NeuronDB), biochemical reaction networks (e.g., SBML), and so on are proving useful in their fields. For visual neuroscience, Viperlib, an online visual perception library, might be a natural host for such a database of stimulus scripts for experiments. First, however, some serious technical issues must be solved. Although libraries like the Vision Egg and PsychoPy make it relatively easy to generate visual stimuli in a free way that is theoretically hardware independent, the issues of framerate, display luminance and position calibration, and synchronization with data acquisition and other hardware would all need to be addressed. Nevertheless, the availability of open source libraries and a number of publications based on them means that such endeavor could already be started.

CONCLUSION

The Vision Egg is a free and open-source programming library that allows scientists to produce arbitrary visual stimuli. Such stimuli can be specified in realtime without skipping frames, may involve traditional stimuli such as sinusoidal gratings, or may be more complex, 3D, and naturalistic scenes. Features such as perspective correction and realtime interpolation of image data for sub-pixel movement are part of OpenGL and thus occur in realtime at little or no extra programming or computational cost.

With the continued increase in power of conventional consumer graphics hardware, the use of such systems for vision science experiments will continue to become more common. This paper described a visual stimulus generation system that utilizes such hardware and addresses critical calibration issues in the luminance and time domains. Of course, such calibration also depends on the display device, which also has temporal, spatial, spectral, and polarization properties that need to be accounted for.

With powerful stimulus generation software and video cards now available, the greatest challenge of producing visual stimuli may now be finding an appropriate physical display device. CRTs are well understood (Bach et al., 1997; Brainard et al., 2002; Cowan, 1995) and would remain a popular stimulus presentation device, but are becoming increasingly more difficult to acquire as their production stops. LCD and DLP based devices are useful for many experiments (Packer et al., 2001). Finally, custom built LED devices may be constructed to address many issues faced with standard commercial technology (Lindemann et al., 2003; Reiser and Dickinson, 2008). Regardless of display technology, if the display device accepts standard inputs (e.g., VGA or DVI), a modular approach to stimulus generation may be used, and stimulus generation software such as the Vision Egg may be used.

CONFLICT OF INTEREST STATEMENT

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

ACKNOWLEDGEMENTS

I thank David O’Carroll for many discussions about, much feedback on, and an environment in which to create the Vision Egg. Many others have contributed to the Vision Egg over the years with bug reports and code submissions. Sol Simpson of SR Research, in particular, helped elucidate the latency issue described in Section “Low-level Hardware and Software Overview: Drawing in OpenGL.” Silicon Graphics, Inc. provided a loan of a high performance workstation. Thanks to Michael Dickinson for use of the high speed video camera. Work was partially supported by a Predoctoral fellowship from the Howard Hughes Medical Institute, who also graciously allowed a leave of absence to work for a summer in private industry, where I learned enough about realtime graphics to create the Vision Egg.

REFERENCES

- Bach, M., Meigen, T., and Strasburger, H. (1997). Raster-scan cathode-ray tubes for vision research – limits of resolution in space, time and intensity, and some solutions. *Spat. Vis.* 10, 403–414.
- Brainard, D. H. (1997). The psychophysics toolbox. *Spat. Vis.* 10, 433–436.
- Brainard, D. H., Pelli, D. G., and Robson, T. (2002). Display characterization. In *Encyclopedia of Imaging Science and Technology*, J. Hornak, ed. (New York, NY, Wiley), pp. 172–188.
- Cowan, W. B. (1995). Displays for vision research. In *Handbook of Optics*, Vol. 1: Fundamentals, Techniques, and Design, M. Bass, ed. (New York, NY, McGraw-Hill), pp. 27.21–27.44.
- Fry, S. N., Müller, P., Baumann, H. J., Straw, A. D., Bichsel, M., and Robert, D. (2004). Context-dependent stimulus presentation to freely moving animals in 3d. *J. Neurosci. Methods* 135, 149–157.
- Fry, S. N., Rohrseitz, N., Straw, A. D., and Dickinson, M. H. (2008). TrackFly: virtual reality for a behavioral system analysis in free-flying fruit flies. *J. Neurosci. Methods* 171, 110–117.
- Kern, R., Lutterklas, M., Petereit, C., Lindemann, J. P., and Egelhaaf, M. (2001). Neuronal processing of behaviourally generated optic flow: experiments and model simulations. *Netw. Comput. Neural Syst.* 12, 351–369.

- Lindemann, J. P., Kern, R., Michaelis, C., Meyer, P., van Hateren, J. H., and Egelhaaf, M. (2003). Flimax, a novel stimulus device for panoramic and highspeed presentation of behaviourally generated optic flow. *Vis. Res.* 43, 779–791.
- Packer, O., Diller, L. C., Verweij, J., Lee, B. B., Pokorny, J., Williams, D. R., Dacey, D. M., and Brainard, D. H. (2001). Characterization and use of a digital light projector for vision research. *Vis. Res.* 41, 427–439.
- Peirce, J. W. (2007). PsychoPy – psychophysics software in python. *J. Neurosci. Methods* 162, 8–13.
- Pelli, D. G. (1997). The VideoToolbox software for visual psychophysics: transforming numbers into movies. *Spat. Vis.* 10, 437–442.
- Reiser, M. B., and Dickinson, M. H. (2008). A modular display system for insect behavioral neuroscience. *J. Neurosci. Methods* 167, 127–139.
- Riemersma, T. (2000). Detecting vertical retrace. *Windows Dev. J.* 11.
- Schreiner, T. (2008). Development and Application of a Python Scripting Framework for bci2000. Thesis, Universität Tübingen.
- Spacek, M., and Swindale, N. (2008). Python for high-throughput electrophysiology. *Front. Neuroinform.*
- Straw, A. D., Rainsford, T., and O’Carroll, D. C. (2008). Contrast sensitivity of insect motion detectors to natural images. *J. Vis.* 8, 1–9.
- Straw, A. D., Warrant, E. J., and O’Carroll, D. C. (2006). A bright zone in male hoverfly (*Eristalis tenax*) eyes and associated faster motion detection and increased contrast sensitivity. *J. Exp. Biol.* 209, 4339–4354.
- Watson, A. B., and Pelli, D. G. (1983). QUEST: a Bayesian adaptive psychometric method. *Percept. Psychophys.* 33, 113–120.



ADVANTAGES OF PUBLISHING IN FRONTIERS



FAST PUBLICATION

Average 90 days
from submission
to publication



COLLABORATIVE PEER-REVIEW

Designed to be rigorous –
yet also collaborative, fair and
constructive



RESEARCH NETWORK

Our network
increases readership
for your article



OPEN ACCESS

Articles are free to read,
for greatest visibility



TRANSPARENT

Editors and reviewers
acknowledged by name
on published articles



GLOBAL SPREAD

Six million monthly
page views worldwide



COPYRIGHT TO AUTHORS

No limit to
article distribution
and re-use



IMPACT METRICS

Advanced metrics
track your
article's impact



SUPPORT

By our Swiss-based
editorial team