



libNeuroML and PyLEMS: using Python to combine procedural and declarative modeling approaches in computational neuroscience

Michael Vella^{1*}, Robert C. Cannon², Sharon Crook³, Andrew P. Davison⁴, Gautham Ganapathy², Hugh P. C. Robinson¹, R. Angus Silver⁵ and Padraig Gleeson^{5*}

¹ Department of Physiology, Development and Neuroscience, University of Cambridge, Cambridge, UK

² Textensor Limited, Edinburgh, UK

³ School of Mathematical and Statistical Sciences and School of Life Sciences, Arizona State University, Tempe, AZ, USA

⁴ Unité de Neurosciences, Information et Complexité, CNRS, Gif sur Yvette, France

⁵ Department of Neuroscience, Physiology and Pharmacology, University College London, London, UK

Edited by:

Eilif Benjamin Muller, Ecole Polytechnique Fédérale de Lausanne, Switzerland

Reviewed by:

Jochen Martin Eppler, Research Center Jülich, Germany
Marcel Stimberg, École Normale Supérieure, France

*Correspondence:

Michael Vella, Department of Physiology, Development and Neuroscience, University of Cambridge, Downing Street, Cambridge CB2 3EG, UK
e-mail: vellamike@gmail.com;
Padraig Gleeson, Department of Neuroscience, Physiology and Pharmacology, University College London, Gower Street, London, WC1E 6BT, UK
e-mail: p.gleeson@ucl.ac.uk

NeuroML is an XML-based model description language, which provides a powerful common data format for defining and exchanging models of neurons and neuronal networks. In the latest version of NeuroML, the structure and behavior of ion channel, synapse, cell, and network model descriptions are based on underlying definitions provided in LEMS, a domain-independent language for expressing hierarchical mathematical models of physical entities. While declarative approaches for describing models have led to greater exchange of model elements among software tools in computational neuroscience, a frequent criticism of XML-based languages is that they are difficult to work with directly. Here we describe two Application Programming Interfaces (APIs) written in Python (<http://www.python.org>), which simplify the process of developing and modifying models expressed in NeuroML and LEMS. The libNeuroML API provides a Python object model with a direct mapping to all NeuroML concepts defined by the NeuroML Schema, which facilitates reading and writing the XML equivalents. In addition, it offers a memory-efficient, array-based internal representation, which is useful for handling large-scale connectomics data. The libNeuroML API also includes support for performing common operations that are required when working with NeuroML documents. Access to the LEMS data model is provided by the PyLEMS API, which provides a Python implementation of the LEMS language, including the ability to simulate most models expressed in LEMS. Together, libNeuroML and PyLEMS provide a comprehensive solution for interacting with NeuroML models in a Python environment.

Keywords: NeuroML, LEMS, model specification, standardization, API, modeling, Python, SWC

INTRODUCTION

In neuroscience, models based on detailed anatomy and electrophysiology have been used for many years to help explore and understand neural systems. Historically, these models have been expressed using a variety of programming languages, tools, and techniques, leading to a high degree of fragmentation (Cannon et al., 2007). In scientific modeling, domain-specific modeling languages have been developed to address this fragmentation, aid with model exchange, and provide language features, such as built-in methods and classes, which simplify modeling in that particular domain. This is achieved by formalizing common concepts with a standardized set of language expressions and rules. Another key benefit of these languages is that they provide a common format that allows different software tools to process the same model description. When considering models of physical systems, scientists tend to think in terms of the relevant core components of those systems (such as neurons, synapses, and ion channels in the case of neural systems) and the interactions

among them. Declarative modeling languages are useful for expressing such conceptual models, as they free the modeler from describing the implementation details of the model, allowing them to focus on the scientific problem. NeuroML is a declarative, XML-based model description language for computational neuroscience, which has been developed as part of an international, collaborative initiative (Goddard et al., 2001; Gleeson et al., 2010). In the latest version of NeuroML (version 2.0 or v2), the structure and behavior of ion channel, synapse, cell, and network model descriptions are based on underlying definitions provided in LEMS, a domain-independent language for expressing hierarchical mathematical models of physical entities (Cannon et al., 2012).

One potential limitation of such a declarative model description language is that the commonly used format for serializing such models, XML, can be difficult to read and write and may be overly verbose, especially for large, complex models. There can be difficulties too when a model includes novel mechanisms that

were not considered during the design of the description language or when constructing large, repetitive models, which can be expressed more tersely in a procedural language by means of recursions or loops. Here we describe libNeuroML and PyLEMS, which address these issues for the NeuroML and LEMS languages, respectively. A wide range of models, from point neurons to morphologically detailed, conductance based cell and network models, can be created, parsed, and saved using the libNeuroML API. PyLEMS is a Python API for creating and working with models and model components specified directly in LEMS.

While there are a number of Python applications for reading, modifying and writing XML such as lxml (<http://lxml.de>)—the advantage of libNeuroML and PyLEMS over these more generic tools is that they contain sub-tools and optimizations specific to the modeling of neural systems. Here we present a brief overview of the current state of NeuroML and LEMS, describe the motivation for developing procedural APIs for these languages, outline the design considerations for libNeuroML and PyLEMS, and provide a number of examples of the usage of these libraries. libNeuroML and PyLEMS allow a user to enjoy the benefits provided by the domain-specificity and rigor of NeuroML and LEMS, while facilitating the use of the Python programming language for procedural model descriptions.

OVERVIEW OF NeuroML AND LEMS

The current scope of NeuroML covers abstract, point neuron models [e.g., leaky integrate and fire models or two-variable spiking neuron models (Izhikevich, 2003; Brette and Gerstner, 2005)], conductance based neuron models, morphologically detailed, multicompartmental neuron models, voltage, and calcium dependent ion channel models, both fixed and plastic synapse models, and models for networks of neurons positioned in 3D with synaptic connections among populations of cells. **Figure 1A** gives an overview of the elements allowed in a NeuroML file, and **Figure 1B** shows an example of a NeuroML serialization of a model. NeuroML is being developed by an international consortium of contributors, where the formal specification for the latest version is being developed by the NeuroML Editorial Board (<http://www.neuroml.org/editors.php>).

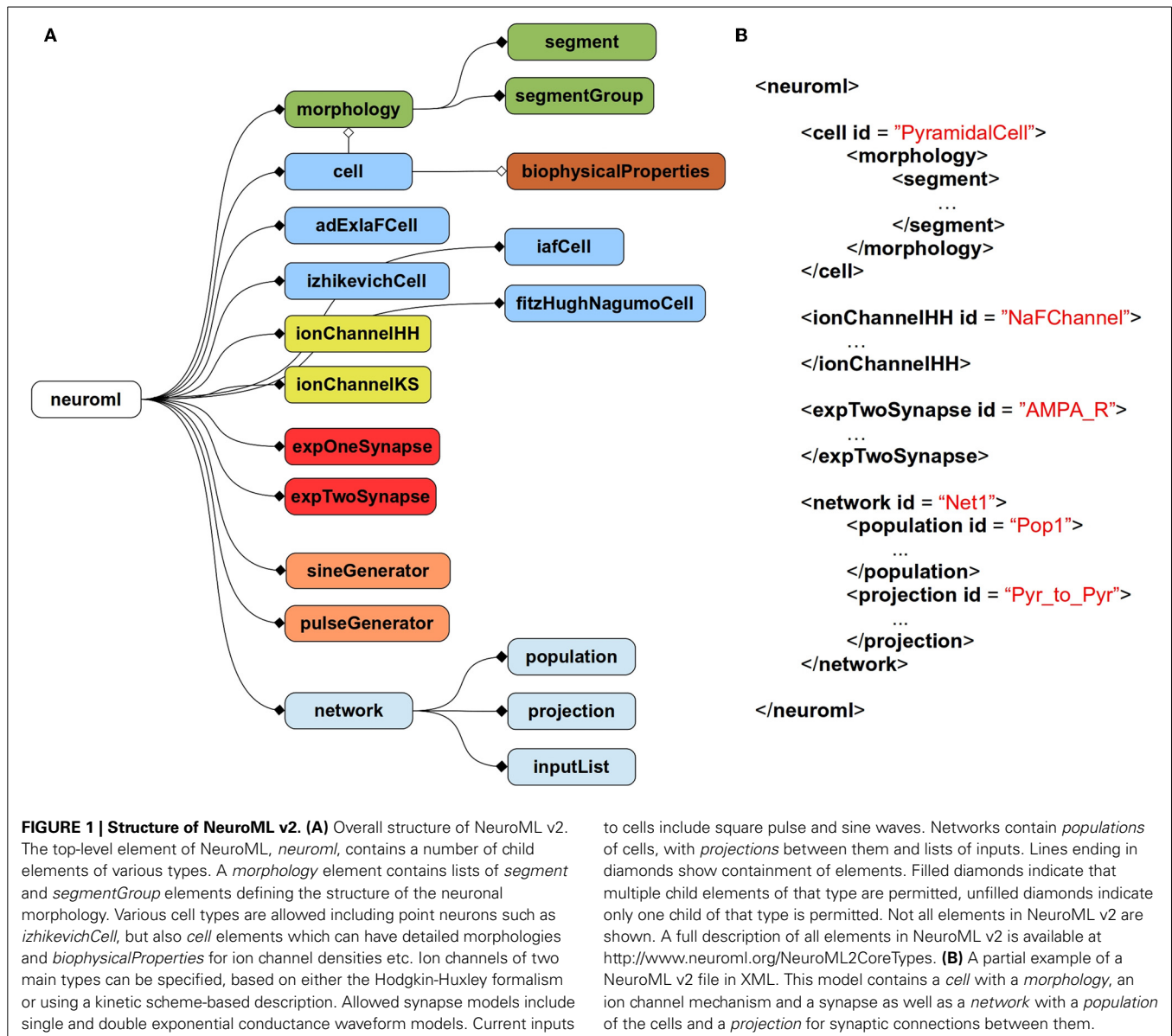
NeuroML v1.x (Gleeson et al., 2010) focused on conductance-based cell models, often with a corresponding multicompartmental representation of neuronal morphology. For these earlier versions, the mathematical descriptions of model components, such as ion channel models based on the Hodgkin-Huxley formalism (Hodgkin and Huxley, 1952), are specified in user documentation (see supplementary information of Gleeson et al., 2010). Modelers or application developers wishing to utilize or support a feature of NeuroML were required to familiarize themselves with the relevant documentation for that component and ensure compliance for any model description or software application (**Figure 2A**). A disadvantage of this approach is the possibility for ambiguity in the documentation. NeuroML v2 was designed in conjunction with a new XML-based language called Low Entropy Model Specification language (LEMS), which can be used for creating fully machine-readable definitions of the structure and behavior of the model components (**Figure 2B**).

The elements in NeuroML v2 have corresponding structural and mathematical definitions described in LEMS.

The LEMS language is used to formally describe the components of models of physical systems, which may contain hierarchical relationships. These components can have parameters, which are fixed, and state variables, which vary according to defined relationships. In LEMS, and hence also in NeuroML, all parameters and state variables are dimensional quantities rather than relying on an implicit set of units. Whenever a quantity is expressed in a model, dimensionally correct units must be provided. It is the responsibility of the implementation to check the units and convert to a consistent internal set of units for calculations. Another important concept in LEMS is that of containment of components, encoding the concept that one model element is part of another (e.g., a population of cells is part of a network, a gate is part of an ion channel). Another key LEMS concept is the ability to declare a prototype *ComponentType*, which defines the generic structure and dynamics for a broad class of models. Models can then be instantiated as *Components* by providing a set of parameters for a specific instance. An example of the *ComponentType* concept is shown in **Figure 2Bi** for the spiking cell model of Izhikevich (2003), a widely-used model in computational neuroscience, which exhibits a diverse range of physiologically-realistic spiking behaviors by changing a small set of parameters in the model. Instances of the model (*Components*), such as the cell defined on the left in **Figure 2Bi**, are specified by providing specific values for the parameters. NeuroML and LEMS use Fortran-like “gt” and “lt” inequality operator symbols instead of “>” and “<.” This is done because in XML the symbols “>” and “<” are used in the declaration of XML tags.

While any modeler is free to describe a model in LEMS, the NeuroML initiative has developed a set of curated LEMS definitions (e.g., Cells.xml and Synapses.xml in **Figure 2B**) for commonly used models, which form the basis for NeuroML v2. Similar classes of model types can be linked together by using the type extension mechanism of LEMS. For example, any object which produces a current extends the type *basePointCurrent*, while all synaptic models extend *baseSynapse*. This approach also provides for user defined extensions to the core NeuroML language.

LEMS provides complete, machine readable model definitions for a broad range of cell, ion channel and synapse models in NeuroML v2 (though not yet for multicompartmental cell models, see Discussion). Each NeuroML release includes a W3C XML Schema Document (XSD, <http://www.w3.org/XML/Schema>), which can be used to validate NeuroML documents, i.e., check whether all required elements and attributes are present. Simulators and other applications that aim to support the language can choose to base their import/export functions on the structure of the language specified by the Schema and associated documentation, as was the case in NeuroML v1.x. However, any simulator utilizing NeuroML should be designed to ensure that simulated behavior complies with LEMS definitions in order for that application to be NeuroML-compliant. The LEMS *ComponentType* definitions are also defined in XML format, and a LEMS-specific Schema document exists for the purpose of validation.



DECLARATIVE AND PROCEDURAL MODEL DESCRIPTIONS IN COMPUTATIONAL NEUROSCIENCE

Procedural programming languages require a description of the sequence of steps to be executed (or control flow) in a computer program in terms of sequential commands. On the other hand, declarative languages provide the information needed for computation without directly expressing the control flow. In this section the comparative advantages and disadvantages of these paradigms are described in the context of modeling in neuroscience.

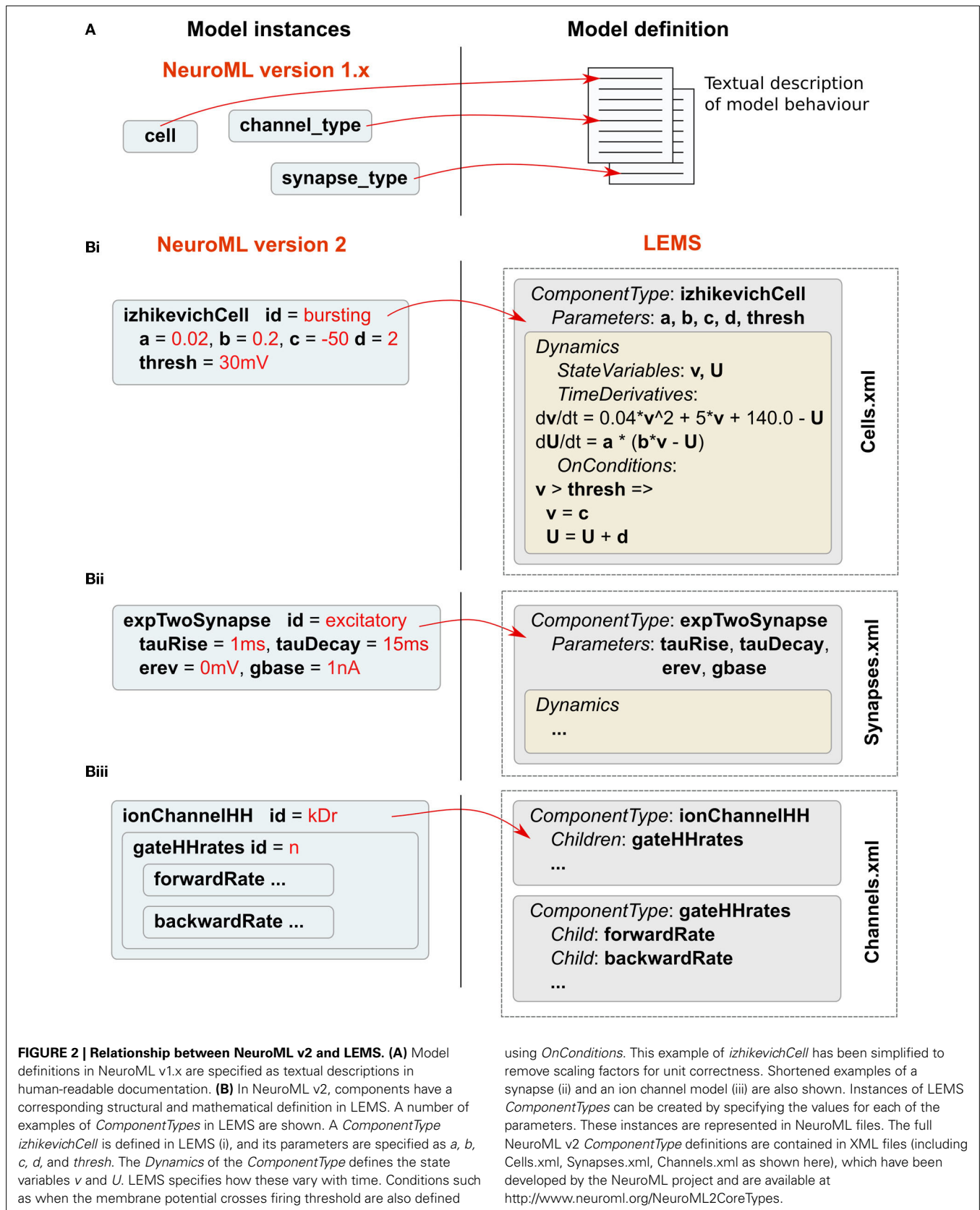
ADVANTAGES OF DECLARATIVE MODEL DESCRIPTIONS

Declarative specification of models can be of benefit in three principal respects: model readability, model interoperability/validation, and the avoidance of fragmentation.

Domain-specific declarative languages are generally easier to read and understand because they shift focus onto describing the

nature of the problem being solved rather than the details of the specific sequence of operations used to solve that problem. A neuroscientist with a background in biophysics and a minimal amount of programming experience can read, understand and modify a NeuroML model with relative ease. However, this may not be true for an equivalent model written in a general-purpose procedural language such as Python or C. Moreover, declarative formats like XML also allow for an easier transformation into more readable presentations such as HTML websites.

Declarative languages for computational modeling such as NeuroML provide a good interchange format for different software tools by ensuring model completeness and facilitating machine-parsing of the models. The domain-specificity of a particular declarative language means that relationships between model elements can be formalized (e.g., in an XML Schema), providing a fixed framework for defining models. This facilitates



model validation and typically makes it simpler to diagnose errors, although it should be noted that this form of validation indicates little about the scientific veracity of the model (Crook et al., 2013).

As a consequence of improved readability and software interoperability, efforts such as NeuroML also reduce the fragmentation in a scientific discipline. The development of common formats for specification of models accelerates progress by encouraging model sharing and reuse, as can be seen with SBML (the Systems Biology Markup Language Hucka et al., 2003 and MathML Miner, 2005).

ADVANTAGES OF PROCEDURAL MODEL DESCRIPTIONS

Despite the advantages of writing models in a declarative manner, procedural specification of models does have two important benefits. First, procedural languages are generally more practical for describing models when there are aspects of the model that can not be described with the base components of a declarative modeling language. When the boundaries and concepts for model design are not clear, a procedural language often will provide a needed degree of flexibility. As an example, if one were designing a multicompartmental model where a particular membrane conductance was only present on dendrites more distal than the second branch point from the soma, it would probably be easier to describe this concisely using a general-purpose language like Python or C, since NeuroML does not natively support defining regions of the cell in this way. A second advantage of developing libraries to allow procedural model description development is that it is easier to integrate with other libraries such as those providing visualization utilities or analysis routines. The libSBML API (Bornstein et al., 2008) for the SBML language has been an important factor in the widespread support for that language among systems biology applications (Sauro and Bergmann, 2008).

Thus, both procedural and declarative paradigms play an important role in computational neuroscience. When working within a widely used modeling formalism, such as Hodgkin-Huxley type conductance based models, having the ability to easily export it to a declarative format is useful and important. In such a situation, declarative modeling allows for ease of development, model interchange among software tools, and model reproducibility. When working with novel modeling approaches or when integrating with scientific or visualization libraries, it is useful to use a procedural approach.

libNeuroML AND PyLEMS

To facilitate procedural model description development for the NeuroML and LEMS languages we have developed libNeuroML and PyLEMS. These Python modules can be imported into a Python script to allow loading of XML files in their respective formats, parsing and editing of the models using APIs which closely follow the structure of the XML languages, and saving in valid XML. libNeuroML, which parses and saves NeuroML v2, has added functionality to use optimized representations of large models, both internally and as serialization formats. PyLEMS has the additional ability to simulate the dynamical behavior of LEMS models. **Figure 3** shows examples of XML from NeuroML

and LEMS and the Python code that can be used to create the equivalent entities. More detailed overviews of the APIs for libNeuroML and PyLEMS are shown in **Figures 4, 5**, respectively. A core aim of libNeuroML and PyLEMS is to provide production-quality, easy-to-use utilities for the manipulation of NeuroML and LEMS, using tools and standards familiar to Python programmers that are also easy to use for those less experienced with Python. With this core aim in mind, libNeuroML and PyLEMS have been implemented with the design goals described below.

DESIGN GOALS SHARED BY BOTH libNeuroML AND PyLEMS

Python naming conventions and adherence to Python PEP8 style guide

libNeuroML and PyLEMS strongly adhere to naming conventions that are widely used in the Python community and codified in the PEP8 Style Guide for Python Code (<http://www.python.org/dev/peps/pep-0008/>). In the case of libNeuroML, this adherence to convention is enforced by automated conversion from NeuroML to Python naming conventions, which occurs during the automatic generation of the libNeuroML object model from the corresponding NeuroML Schema (see below). NeuroML elements have their names modified in the libNeuroML object model to use standard Python naming conventions; for instance, *izhikevichCell* (element) and *q10Settings* (attribute) in NeuroML become **IzhikevichCell** (class) and **q10_settings** (field), respectively in libNeuroML.

Automated XML validation

libNeuroML provides a validation utility to ensure that NeuroML documents are well-formed (following the basic syntactic rules of XML) and valid (following the structure defined in the NeuroML Schema). A similar utility exists in PyLEMS for validating against the LEMS Schema. Note that while the NeuroML *ComponentType* definitions as well as XML generated by PyLEMS are valid according to this Schema, PyLEMS is flexible enough to parse invalid LEMS files (e.g., with reordered elements) as long as they follow the correct containment rules for LEMS elements.

Ease of installation

libNeuroML and PyLEMS utilize the standard distutils (<http://docs.python.org/2/library/distutils.html>) tool for packaging Python programs, making installation standard and simple. Both APIs require the lxml (<http://lxml.de>) Python package. Additionally, libNeuroML requires numpy (<http://www.numpy.org>), jsonpickle (<https://pypi.python.org/pypi/jsonpickle>), pymongo (<https://pypi.python.org/pypi/pymongo>), and PyTables (<http://docs.mongodb.org/ecosystem/drivers/python>), and PyTables (<http://www.pytables.org>) packages. Both APIs are currently tested and stable for Python versions 2.6 and 2.7. All of these packages can be obtained from the Python Package Index (<https://pypi.python.org/pypi>). The full source code for the libraries can be obtained from <https://github.com/NeuralEnsemble/libNeuroML> and <https://github.com/LEMS/pylems>.

ADDITIONAL DESIGN GOALS FOR libNeuroML

Auto-generation from NeuroML Schema

In libNeuroML every element (such as *cell*, *network* or *pulseGenerator*, see **Figure 1A**) in the NeuroML Schema corresponds to

Declarative XML specification

Procedural Python specification

NeuroML

```
<segment id="0" name="Soma">
  <proximal x="0" y="0" z="0" diameter="30"/>
  <distal x="30" y="0" z="0" diameter="30"/>
</segment>
```

```
import neuroml
p = neuroml.Point3DWithDiam(x=0, y=0, z=0,
                             diameter=30)
d = neuroml.Point3DWithDiam(x=30, y=0, z=0,
                             diameter=30)

soma = neuroml.Segment(id=0,
                       name='Soma',
                       proximal=p,
                       distal=d)
```

LEMS

```
<ComponentType name="simpleIafCell">
  <Parameter dimension="voltage"
             name="threshold"/>
  <Parameter dimension="voltage"
             name="reset"/>
  <Parameter dimension="time"
             name="tau"/>
  <Parameter dimension="voltage"
             name="leakReversal"/>
  <Exposure dimension="voltage"
            name="v"/>
  <Dynamics>
    <StateVariable dimension="voltage"
                  exposure="v"
                  name="v"/>
    <TimeDerivative variable="v"
                   value="(leakReversal - v) / tau"/>
    <OnCondition test="v .gt. threshold">
      <StateAssignment value="reset"
                      variable="v"/>
    </OnCondition>
  </Dynamics>
</ComponentType>
```

```
import lems.api as lems
model = lems.Model()

iaf_tau = lems.ComponentType('simpleIafCell')
model.add(iaf_tau)

iaf_tau.add(lems.Parameter('threshold',
                           'voltage'))
iaf_tau.add(lems.Parameter('leakReversal',
                           'voltage'))
iaf_tau.add(lems.Parameter('reset',
                           'voltage'))
iaf_tau.add(lems.Parameter('tau',
                           'time'))

iaf_tau.add(lems.Exposure('v',
                          'voltage'))

iaf_tau.dynamics.add(lems.StateVariable('v',
                                         'voltage',
                                         'v'))

iaf_tau.dynamics.add(lems.TimeDerivative('v',
                                          '(leakReversal - v) / tau'))

condition = lems.OnCondition('v .gt. threshold')
condition.add(lems.StateAssignment('v', 'reset'))
iaf_tau.dynamics.add(condition)
```

FIGURE 3 | Examples of NeuroML and LEMS models specified in XML and equivalent models specified with Python using the libNeuroML and PyLEMS APIs, respectively. The XML code on the left is generated

automatically by the Python code on the right. The NeuroML example describes a soma segment as a 3D cylinder. The LEMS example illustrates how a simple integrate and fire cell can be defined.

a concrete class and therefore can be instantiated as an object by calling `neuroml.<NameOfClass>()`. This is possible because the libNeuroML core object model is auto-generated from a NeuroML Schema file via the generateDS tool (<https://bitbucket.org/dkuhlman/generateds>), where the generated object model defines all necessary type interfaces. Therefore, libNeuroML provides a complete and direct mapping between the NeuroML Schema and its internal Python object model with several advantages:

1. *Maintainability.* libNeuroML can be rapidly updated to reflect the latest NeuroML Schema with little or no knowledge of

implementation or architecture. This allows libNeuroML to be a core part of the regular NeuroML release process. New versions of the NeuroML Schema will always be released along with a libNeuroML version which reflects those Schema changes.

2. *Backward—support.* This feature allows users to create a Python API for handling NeuroML even if they are working with older versions of the NeuroML representation format, including versions that existed before libNeuroML. However, the user must auto-generate this libNeuroML version via the generateDS tool as described in libNeuroML documentation.

libNeuroML Python API: **modules**, *Classes*, *Classifications* and usage **examples****neuroml**

| | | |
|----------------|--------------------|--------------------------|
| AdExIaFCell | IaFCell | IF_cond_alpha |
| IzhikevichCell | IaFRefCell | IF_cond_exp |
| | IaFTauCell | IF_curr_alpha |
| | IaFTauRefCell | IF_curr_exp |
| | FitzHughNagumoCell | EIF_cond_alpha_isfa_ista |
| | | EIF_cond_exp_isfa_ista |

| | | |
|-----------------|-----------------------|-------------------------|
| Cell | BiophysicalProperties | IntracellularProperties |
| Morphology | MembraneProperties | Resistivity |
| Segment | SpecificCapacitance | Species |
| SegmentGroup | ChannelDensity | |
| Point3DWithDiam | ChannelDensityNernst | |
| | ChannelDensityGHK | |

| | |
|---------------|------------------------|
| ExpOneSynapse | BlockingPlasticSynapse |
| ExpTwoSynapse | PlasticityMechanism |
| | BlockMechanism |

| | |
|--------------|------------|
| IonChannelHH | HHRate |
| GateHHRates | HHTime |
| GateHHTauInf | HHVariable |

| | | |
|----------------|---------------|-----------------------|
| PulseGenerator | RampGenerator | SpikeGenerator |
| SineGenerator | VoltageClamp | SpikeGeneratorPoisson |
| | | SpikeGeneratorRandom |
| | | SpikeArray |

| | | |
|------------|------------|-----------|
| Population | Projection | InputList |
| Instance | Connection | Input |
| Location | | |

NeuroMLDocument

neuroml.loaders

| | |
|---------------|------------------|
| NeuroMLLoader | JSONLoader |
| SWCLoader | ArrayMorphLoader |

neuroml.writers

| | |
|---------------|------------------|
| NeuroMLWriter | ArrayMorphWriter |
| JSONWriter | |

neuroml.utils*Point neuron models*

```
izh = IzhikevichCell(id='izh', a='.02', b='.2',
                    c='-65.0', d='6', v0='-70mV', thresh='30mV')
iaf = IaFTauCell(id='iafTau', leakReversal='-50mV',
                thresh='-55mV', reset='-70mV', tau='30ms')
```

Multicompartmental, conductance based neuron models

```
cell = Cell(id='cell0')
cd = ChannelDensity(cond_density='50mS_per_cm2',
                  ion_channel='NaF', erev='55mV', ion='na')
cell.membrane_properties =
    MembraneProperties().channel_densities.append(cd)
```

Synapses

```
e2syn = ExpTwoSynapse(id='gaba', tau_decay='12ms',
                     tau_rise='3ms', gbase='1nS', erev='-70mV')
```

Ion channels

```
ic = IonChannelHH(id='NaF', conductance='10pS',
                 species='na')
m = GateHHRates(id='m', instances='3')
m.forward_rate = HHRate(type='HHExpRate',
                       rate='0.07per_ms', midpoint='-65mV', scale='-20mV')
```

Inputs

```
p = PulseGenerator(id='p', delay='10ms',
                  duration='100ms', amplitude='50 pA')
v = VoltageClamp(id='v', delay='5ms', duration='5ms',
                target_voltage='0mV', series_resistance='10ohm')
```

Networks

```
net = Network(id='net1')
pop = Population(id='p1', component='cell0', size=10)
net.populations.append(pop)
```

Top level class for constructing models

```
nml_doc = NeuroMLDocument()
nml_doc.append(net)
```

File readers/importers

```
nml_doc2 = NeuroMLLoader.load('file.nml')
```

Export formats

```
JSONWriter.write(nml_doc, 'file.nml')
```

Helper methods

```
validate_neuroml2(nml_file)
```

FIGURE 4 | An overview of the structure of the libNeuroML API. The API is divided into four main modules (left), the largest of which (neuroml) consists of Python classes generated from the NeuroML XML Schema Document. There is roughly a one to one correspondence between the NeuroML elements and Python classes. These classes can be split into a

number of broad classifications (right) based on the types of models they represent. Examples of Python code using the classes is also shown on the right. Extra modules have been created to facilitate loading or writing NeuroML (in XML or other serialization formats) and for validating NeuroML files.

- Flexibility.** It is possible for a user to modify the NeuroML Schema in order to develop new components or change existing ones. A custom copy of libNeuroML can then be generated for further testing. Such modifications can then be proposed for inclusion in the language through discussions on the NeuroML mailing list.
- Saving to valid NeuroML.** A key feature of generateDS and, by extension, libNeuroML is the ability to save XML files which are valid against the Schema used to generate the API.
- Automatic conversion of names to Python format.** While the convention in NeuroML is to use “camel case” for naming elements and their attributes, the generated Python class and

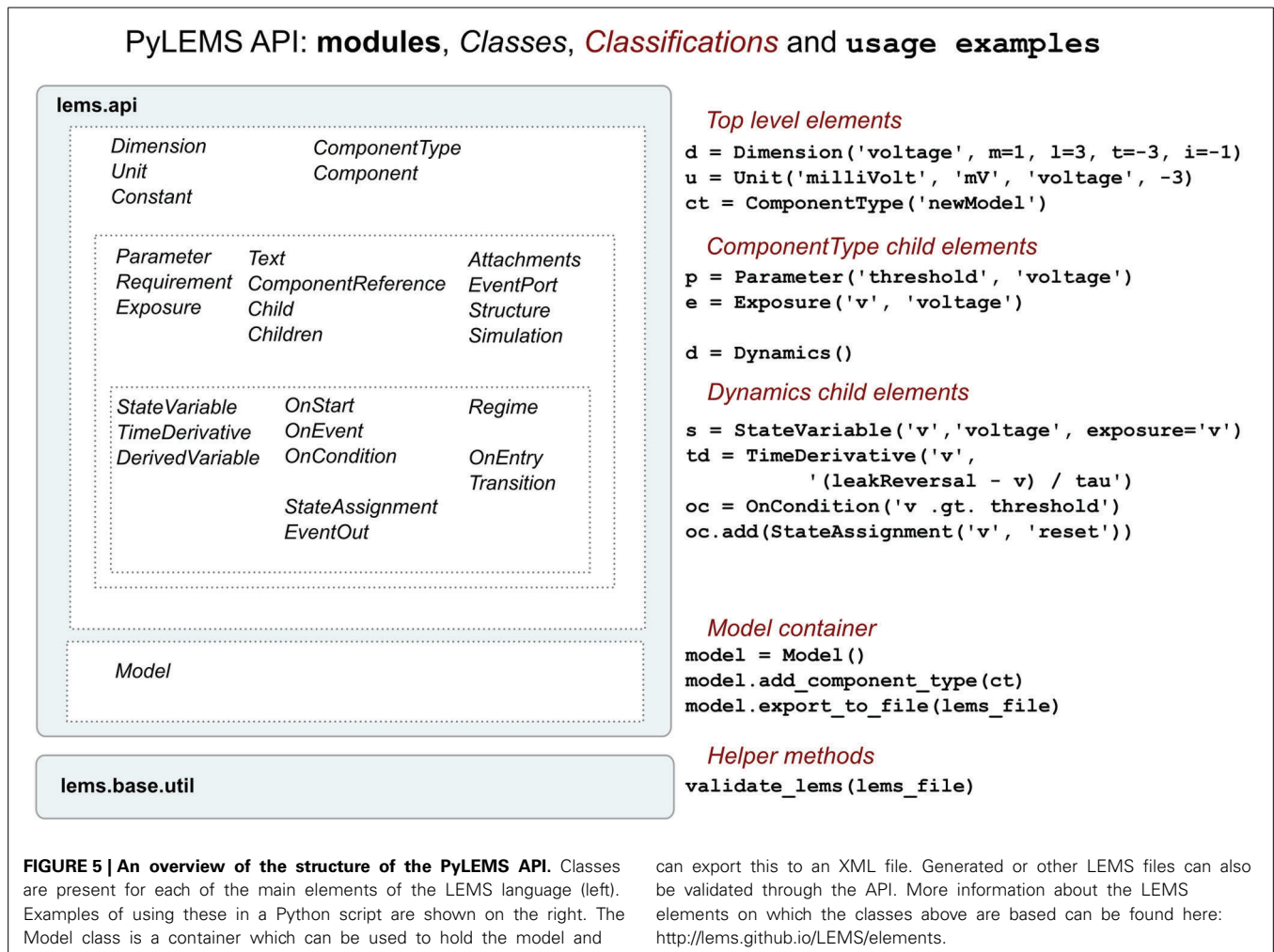


FIGURE 5 | An overview of the structure of the PyLEMS API. Classes are present for each of the main elements of the LEMS language (left). Examples of using these in a Python script are shown on the right. The Model class is a container which can be used to hold the model and

can export this to an XML file. Generated or other LEMS files can also be validated through the API. More information about the LEMS elements on which the classes above are based can be found here: <http://lems.github.io/LEMS/elements>.

method names automatically are converted to Python naming conventions.

One notable disadvantage of an auto-generated API is that it contains syntax which is less usable than it might be for a “hand-written” API. However, we have contributed updates to the generateDS tool to increase usability and readability of the API, and these will continue to be improved.

Serialization and database support

This section describes the different serialization formats provided by libNeuroML. In this section the term NeuroML specifically refers to the XML-based file format rather than NeuroML as a model description language. libNeuroML supports serialization of models into four different storage formats: NeuroML (XML), JavaScript Object Notation (JSON) (<http://www.json.org>), HDF5 (<http://www.hdfgroup.org/HDF5>), and SWC (Cannon et al., 1999). Of these four serialization formats, two (NeuroML and JSON) are lossless, which is to say that they preserve all of the NeuroML model data, and the other two (HDF5 and SWC) are lossy and are only able to serialize a subset of the data, namely the morphological structure of detailed neuronal reconstructions. In

the case of SWC however, the format is only suited to serializing a small subset of morphological data and is intrinsically unable to serialize complex models, due to the nature of the SWC format as described below.

In addition to file-based serialization, libNeuroML provides the ability to store data in a MongoDB (<http://www.mongodb.org>) database via an intermediate JSON document. All serialization and deserialization operations, including database operations, are carried out through the **neuroml.writers** and **neuroml.loaders** modules of libNeuroML, respectively (Figure 4).

NeuroML (XML). The de-facto libNeuroML serialization output is the standard XML-based format in which NeuroML models are written. Small to medium-sized NeuroML models generally should be written in this format, as these documents can be edited with standard text editors and validated against the NeuroML Schema independently of libNeuroML. The XML serialization is the format most widely supported by NeuroML compliant software tools (http://www.neuroml.org/tool_support.php). It is recommended that unless there is a clear reason to do otherwise, users should use XML

serialization until they encounter performance bottlenecks. NeuroML serialization, though the slowest and least concise, permits the use of NeuroML-compliant tools (such as morphology viewers) and the ability to easily track changes across file versions.

JSON. JSON is an open, text-based format for human-readable data interchange. Any file serialized by libNeuroML to JSON format can be loaded by libNeuroML without any loss of information and thus can be re-serialized as NeuroML. The **jsonpickle** (<https://github.com/jsonpickle/jsonpickle>) module is used to serialize NeuroML documents in JSON format. There are three primary advantages of JSON serialization:

1. Some users report that JSON is easier to read and understand than XML.
2. Many tools exist to facilitate the use of JSON in situations where data is transmitted over networks. This is particularly true for web-based applications since a number of such tools and frameworks are optimized for working with JSON.
3. When used in conjunction with the libNeuroML **arraymorph** module, large-scale morphological reconstructions require substantially less space on disk when stored as JSON-serialized documents.

SWC. SWC is a tree-based representation used for storing morphological reconstructions of neurons, and it is the data storage format of the NeuroMorpho.org database of reconstructed neurons (Ascoli et al., 2007). In SWC, each node contains diameter and position information, and the conical frustum between two nodes is treated as a segment of morphology. This allows, for instance, reconstructions of dendritic arborizations of theoretically-unlimited complexity. libNeuroML can import the **Morphology** component of a cell in SWC format. However, any other NeuroML component data such as information about the distribution of channels or synapses with regard to the morphology would be lost when serializing in this format.

HDF5. HDF5 (Hierarchical Data Format 5, <http://www.hdfgroup.org/HDF5>) is a set of file formats and tools for storing and organizing large amounts of numerical and hierarchical data. HDF5 serialization is provided by libNeuroML, although presently the level of serialization support which is provided only extends to morphological reconstructions of neurons. The main advantage of HDF5 support is the relatively low memory footprint and support for rapid read/write operations, which is demonstrated in the benchmarks section of this article.

MongoDB. MongoDB (<http://www.mongodb.org/>) is a document-oriented “NoSQL” database which departs from the traditional table-based relational database paradigm toward JSON-like documents with dynamic schemas. Its document-oriented approach is particularly suitable for storing NeuroML documents. The MongoDB support provided by libNeuroML is particularly useful for users wishing to store large amounts of data on a server, such as for use with a website. Since MongoDB

is compatible with NeuroML’s **arraymorph** module, large models can be stored in a MongoDB database at a fraction of the disk requirements for equivalent NeuroML documents. One MongoDB limitation is that JSON files exceeding 16 MB can not be stored; large documents (more than 100,000 segments) should therefore be stored as individual files.

Serialization file sizes. For reconstructed morphologies, the file sizes of the different serialization formats vary linearly with the number of segments in the morphology with NeuroML requiring 316 B/segment, JSON requiring 62 B/segment, and HDF5 requiring 41 B/segment. For a typical reconstructed morphology of 2000 segments, a NeuroML (XML) file would require 617 KB, and the corresponding JSON and HDF5 files would require 80 and 41 KB, respectively.

OPTIMIZED INTERNAL REPRESENTATION (ARRAYMORPH)

The **arraymorph** module of libNeuroML provides highly optimized representations of neuronal morphologies to increase read/write speed and reduce memory footprint. Neuronal morphologies are instantiated as a non-NeuroML **ArrayMorphology** type, which inherits from the standard NeuroML **Morphology** type but uses Numpy (<http://www.numpy.org/>) arrays to represent morphologies in a way that is transparent to the user and is highly-influenced by the SWC format. With the **arraymorph** module, it remains possible to manipulate NeuroML segment objects, and all helper methods and properties continue to work as with the standard NeuroML **Morphology** type. However, low-level access to the arrays is also possible, although the user must understand the internal details of implementation. The rationale for development of this module stems from several drawbacks of both Python and XML that can be summarized as follows:

- Python requires a relatively large memory footprint when instantiating objects. Representing every component in a NeuroML file in-memory by an instantiated object can demand an unfeasibly-large amount of memory. A typical reconstructed morphology of 1600 segments requires 6.9 MB in memory; a network of over 1000 cells would therefore require over 1 GB of memory.
- XML serialization results in performance bottlenecks when read/write operations are conducted on large XML files because NeuroML is a relatively verbose format, and the whole file must be loaded for even one element to be accessed.
- Because many NeuroML documents are mainly morphological reconstructions with a very small amount of metadata, a SWC-like format can store most of the same data with a much smaller performance overhead.
- Recent projects in the field of computational neuroscience such as the Blue Brain Project (Markram, 2006) and the Brain Activity Map Project (Alivisatos et al., 2012) require increasingly large amounts of morphological reconstruction data.
- Some mathematical analysis methods and transformations that can be performed on reconstructed morphologies,

such as morphology transpositions or branching analysis, can be performed on SWC-like flat representations more efficiently than by recursive traversals of tree-based data structures.

- An optimized internal morphology offers the advantage of faster read/write operations while simultaneously hiding a lot of the underlying complexity from the user.
- The SWC-like nature of the **ArrayMorphology** class provides a conceptual bridge between the component-based representation idea underlying NeuroML and the node-based morphology representation underlying SWC.

ADDITIONAL DESIGN GOALS FOR PyLEMS

Simulation of LEMS models

When defining a new language like LEMS, which aims to encapsulate a broad range of dynamical model descriptions, it is imperative to execute the models and test the results against expected behavior. This has been a core principle for the development of jLEMS (<https://github.com/LEMS/jLEMS>), a Java based reference implementation of LEMS developed in parallel with LEMS. A feature of LEMS is not considered stable by the design team unless an example can be executed with jLEMS. Like jLEMS, PyLEMS is not only an API for the language, but also can be used to simulate the dynamical behavior of the models it parses. While both jLEMS and PyLEMS feature only simple numerical integration techniques (forward Euler and additionally, in the case of jLEMS, Runge Kutta fourth-order), both implementations are required to produce the same model behavior for a given model specification. While some of the advanced features of LEMS (e.g., kinetic scheme based ion channel descriptions) are not yet considered stable in PyLEMS, this dual test for LEMS model behavior is invaluable in the development of the language.

METHODS AND RESULTS

IMPLEMENTATION OF XML BINDINGS FOR libNeuroML

The generateDS Python package is used for automatic generation of NeuroML XML-bindings in libNeuroML from the NeuroML Schema. Full details of this conversion process are available in the libNeuroML documentation (http://libneuromldev.readthedocs.org/en/latest/implementation_of_bindings.html).

BENCHMARKS

libNeuroML contains a benchmarking module which presently supports benchmarks for writing NeuroML model data to disk in a variety of serializations. All benchmarks described here were run on a DELL PowerEdge R815 containing a 64-core AMD Opteron 6272 processor, 7.2K RPM Hard Drive and 128 GB memory (16 × 8 GB 1600 MHz RDIMMs). During benchmarking a range of synthetic morphologies of various sizes were generated (sample size of 10) using the **arraymorph** module and written to disk. Write-to-disk times were recorded. The benchmark results (**Figure 6**) indicate that NeuroML (XML) serialization of NeuroML documents with libNeuroML is relatively slow in terms of read/write operations. However, NeuroML documents are readable by humans and a wide variety of software

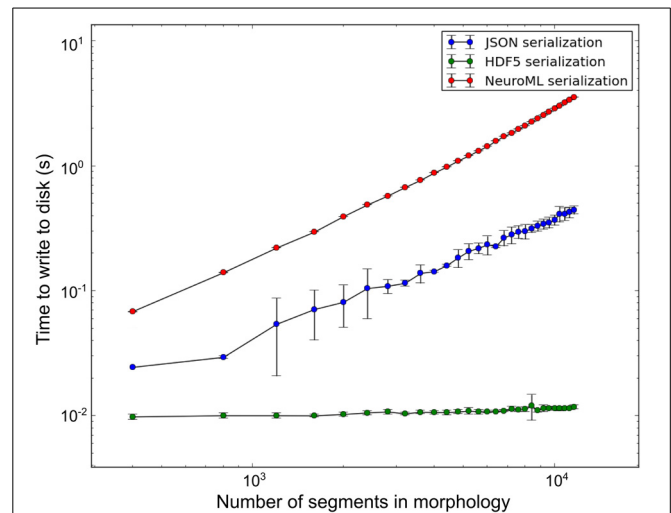


FIGURE 6 | Disk write time for synthetic morphologies in different serialization formats. Error bars represent one standard deviation (sample size = 10).

tools, making this serialization format advisable for most use cases. Benchmark results also show that HDF5 write operations are approximately 100 and 600 times faster than corresponding JSON and XML write operations, respectively, suggesting that if performance bottlenecks are encountered, HDF5 should be investigated for storing morphological reconstruction data. JSON serialization offers a good compromise between the speed of writing to disk provided by HDF5 and the readability and full NeuroML language support of XML serialization. The higher write speeds of JSON when compared to XML are achieved due to use of the **arraymorph** module and the fact that JSON-serialized NeuroML is a less verbose than the XML equivalent.

TESTING, VALIDATION, AND CONTINUOUS INTEGRATION

libNeuroML development follows modern software development practices, including a stable release cycle, version control, testing, and continuous integration. Each libNeuroML stable release is part of the official NeuroML release cycle. The git (<http://git-scm.com>) source code management system is used to provide version control management and is used in conjunction with GitHub (<https://github.com>) to provide issue tracking functionality as well as a central repository for developers. The package is tested using a variety of unit and integration tests, and the Travis-CI (<https://travis-ci.org>) continuous integration platform is used to confirm that libNeuroML is correctly installing and that all tests are passed every time a change is applied to the software and pushed to GitHub. Test coverage is 91% [measured with the Python Coverage module version 3.7.1 (<https://pypi.python.org/pypi/coverage>)]. PyLEMS is also developed on GitHub and released as part of NeuroML release cycle. Basic unit testing and continuous integration on Travis-CI have been added to PyLEMS and will be expanded in the future.

USAGE EXAMPLES

EXAMPLES USING `libNeuroML`

Network example

`libNeuroML` can simplify the process of describing a model of a spiking neural network in NeuroML. All of the cell, synapse and channel models in NeuroML are available for use in networks, and the structure of a network can be generated in a procedural way, using all the capabilities of Python to encode the connectivity. **Figure 7** shows a network of two randomly connected populations of integrate and fire neurons, one of which receives current injections of random magnitude. The structure of the network can be saved in XML format (or JSON format as outlined previously), but an important point is that many of the diverse network connectivity options available can not be described efficiently in NeuroML, and a Python script using `libNeuroML` provides a compact, cross-platform encoding of the network. While the XML in the example in **Figure 7** is roughly the same length as the associated Python script, generally this will not be the case for larger networks. Also, the XML represents only one instance of a network that can be generated by the script. Modifying the script to specify the random seed will ensure reproducibility of the network connectivity, and so the `libNeuroML` version of the network can be distributed instead of the potentially very large XML or JSON serialization.

`PyNN` (Davison et al., 2009) is an API in Python that also offers the advantage of compact, procedural network descriptions. Currently, the cell models that can be defined in `PyNN` scripts are limited to a set of commonly-used point neuron models on simulators with `PyNN` backends. As outlined in the Discussion section of this article, we are actively working toward greater compatibility between `PyNN` and network descriptions in `libNeuroML`.

Multicompartmental cells

Aside from the ability to load and parse NeuroML documents containing morphological reconstructions of neurons, `libNeuroML` allows for the modification and analysis of these morphologies, as well as the generation of completely synthetic morphologies. **Listing 1** shows an example of a script in `libNeuroML` which loads a morphology file from an XML representation, extracts the morphology and calculates the total volume and area of the segments in the cell. In this listing, **volume** and **area** methods do not have trailing parentheses because they are Python getter methods modified by the Python **@property** decorator. This demonstrates how helper methods can be used for operations that will commonly be required on those classes (e.g., getting the surface area or volume of the conical frustum). For details on how such helper methods can be added to `libNeuroML` the reader is referred to the `libNeuroML` developer documentation (<http://libneuromldev.readthedocs.org/en/latest/#developer-documentation>).

OpenWorm and neuroConstruct

The OpenWorm project (<http://www.openworm.org>) is an international collaboration with the aim of creating a physically- and biologically-detailed computer model of a behaving multicellular organism, the nematode *Caenorhabditis elegans*. The nervous

system of this worm contains only 302 neurons, making it a very interesting model organism in experimental and computational neuroscience. The morphology of each of these neurons (as well as all other cells of the adult hermaphrodite) has been reconstructed in 3D at the VirtualWorm project (<http://caltech.wormbase.org/virtualworm>) and released into the public domain in Blender (<http://www.blender.org>) format, which is used for the creation of 3D applications. These have been converted to NeuroML format for the OpenWorm project and made available at <https://github.com/openworm/CElegansNeuroML>. Data on the connectivity between individual cells is available in a spreadsheet (`CElegansNeuronTables.xls`) in that repository and includes the numbers of known chemical or electrical connections between neurons of a particular type.

We have developed a `libNeuroML`-based Python script to analyze the integrity of this data that loads each neuron morphology, generates connections based the connectivity data, and saves the network file to NeuroML. When generating connections for each pair of cells where there exist *N* connections, the script chooses a variable number (100–5000 depending on number of segments in the cells) of random points on the presynaptic and postsynaptic cells and chooses the *N* closest pairs of points for the connections. The generated network file can be loaded into `neuroConstruct` (Gleeson et al., 2007), which can import NeuroML cells and networks for visualization (**Figure 8**). Due to the fact that all neurites are reconstructed, it should be possible to find points separated by a short distance on any connected pair of presynaptic and postsynaptic cells. The long connections between some neurons in **Figure 8B** (which could not be removed by increasing the number of random connections tested) highlights that these are unlikely to be real anatomical connections.

PyLEMS EXAMPLE

The NeuroML v2 *ComponentType* definitions specified in LEMS for multiple commonly-used cell, synapse and ion channel types (**Figure 2**) can be loaded by `PyLEMS`. Another important feature of using LEMS as the basis for NeuroML v2 model types is that a modeler can define a new model type in LEMS if that model is not already present in the core definitions, significantly increasing the extensibility of the NeuroML language. `PyLEMS` allows these new models to be created using Python and saved in a valid, standard format. **Figure 9** shows the code required to define a Hindmarsh and Rose spiking cell model (Hindmarsh and Rose, 1984) and the behavior of one state variable when simulated with `PyLEMS`. More details on the implementation of this model can be found on the Open Source Brain website (<http://www.opensourcebrain.org/projects/hindmarshrose1984>).

DISCUSSION

SUMMARY OF ADVANTAGES

NeuroML is a model description language which can represent biological concepts such as cells, ion channels and networks in a declarative format, while the corresponding mathematical and structural definitions of these elements can be defined in LEMS. The libraries described here, `libNeuroML` and `PyLEMS`, together provide a flexible toolkit for utilizing and extending the NeuroML language with Python. They complement the declarative model

specification benefits of the XML forms of these languages by offering APIs which can be used for procedural model development. While the libNeuroML API provides a Python object model which has a type interface defined by the NeuroML Schema, it also offers a memory-efficient array-based internal

representation for morphological neuron reconstructions and allows export of NeuroML models into different formats. libNeuroML also provides a range of helper methods that provide several common operations required when interacting with models expressed in NeuroML. In the same way that libNeuroML

Declarative XML specification

```
<neuroml id="IafNet">
  <expTwoSynapse id="syn0" erev="0mV" gbase="10nS"
    tauDecay="10ms" tauRise="0.5ms"/>
  <iafCell id="iaf0" reset="-65mV" C="1.0 nF"
    thresh="-50mV" leakConductance="10 nS"
    leakReversal="-65mV"/>
  <pulseGenerator id="input0" delay="0ms" duration="500ms"
    amplitude="0.602601nA"/>
  <pulseGenerator id="input1" delay="0ms" duration="500ms"
    amplitude="0.794995nA"/>
  <pulseGenerator id="input2" delay="0ms" duration="500ms"
    amplitude="0.751483nA"/>
  <pulseGenerator id="input3" delay="0ms" duration="500ms"
    amplitude="1.127460nA"/>
  <pulseGenerator id="input4" delay="0ms" duration="500ms"
    amplitude="0.834095nA"/>
  <network id="IafNet">
    <population id="Pop0" component="iaf0" size="5"/>
    <population id="Pop1" component="iaf0" size="5"/>
    <projection id="Proj0" postsynapticPopulation="Pop1"
      presynapticPopulation="Pop0" synapse="syn0">
      <connection preCellId="..Pop0[0]"
        postCellId="..Pop1[4]"/>
      <connection preCellId="..Pop0[1]"
        postCellId="..Pop1[0]"/>
      <connection preCellId="..Pop0[1]"
        postCellId="..Pop1[3]"/>
      <connection preCellId="..Pop0[2]"
        postCellId="..Pop1[3]"/>
      <connection preCellId="..Pop0[2]"
        postCellId="..Pop1[4]"/>
      <connection preCellId="..Pop0[3]"
        postCellId="..Pop1[1]"/>
      <connection preCellId="..Pop0[4]"
        postCellId="..Pop1[1]"/>
    </projection>
    <inputList id="il0" component="input0" population="Pop0">
      <input destination="synapses" id="0"
        target="..Pop0[0]"/>
    </inputList>
    <inputList id="il1" component="input1" population="Pop0">
      <input destination="synapses" id="1"
        target="..Pop0[1]"/>
    </inputList>
    <inputList id="il2" component="input2" population="Pop0">
      <input destination="synapses" id="2"
        target="..Pop0[2]"/>
    </inputList>
    <inputList id="il3" component="input3" population="Pop0">
      <input destination="synapses" id="3"
        target="..Pop0[3]"/>
    </inputList>
    <inputList id="il4" component="input4" population="Pop0">
      <input destination="synapses" id="4"
        target="..Pop0[4]"/>
    </inputList>
  </network>
</neuroml>
```

Procedural Python specification

```
import neuroml
from random import random

nml_doc = neuroml.NeuroMLDocument(id='IafNet')

# Define integrate & fire cell
iaf0 = neuroml.IafCell(id='iaf0', C='1.0 nF', reset='-65mV',
  thresh = '-50mV', leak_conductance='10 nS',
  leak_reversal='-65mV')
nml_doc.iaf_cells.append(iaf0)

# Define synapse
syn0 = neuroml.ExpTwoSynapse(id='syn0', gbase='10nS',
  erev='0mV', tau_rise='0.5ms', tau_decay='10ms')
nml_doc.exp_two_synapses.append(syn0)

# Create network
net = neuroml.Network(id='IafNet')
nml_doc.networks.append(net)

# Create 2 populations
size0 = 5
size1 = 5
pop0 = neuroml.Population(id='Pop0', size = size0,
  component=iaf0.id)
pop1 = neuroml.Population(id='Pop1', size = size1,
  component=iaf0.id)
net.populations.append(pop0)
net.populations.append(pop1)

# Create a projection between them
proj1 = neuroml.Projection(id='Proj0', synapse=syn0.id,
  presynaptic_population=pop0.id,
  postsynaptic_population=pop1.id)
net.projections.append(proj1)

for pre in range(0,size0):

  # Create a number of random amplitude current pulses
  pg = neuroml.PulseGenerator(id='input%i'%pre,
    delay='0ms', duration='500ms',
    amplitude='%fnA'%(0.5+random()))
  nml_doc.pulse_generators.append(pg)

  # Add these to cells
  input_list = neuroml.InputList(id='il%i'%pre,
    component=pg.id, populations=pop0.id)
  input = neuroml.Input(id=pre,
    target='..%s[%i]%(pop0.id, pre)',
    destination='synapses')
  input_list.input.append(input)
  net.input_lists.append(input_list)

  # Connect cells with defined probability
  prob_connection = 0.3
  for post in range(0,size1):
    if random() <= prob_connection:
      conn = neuroml.Connection(
        pre_cell_id='..%s[%i]%(pop0.id,pre)',
        post_cell_id='..%s[%i]%(pop1.id,post)')
      proj1.connections.append(conn)
```

FIGURE 7 | Example of creation of a network of integrate-and-fire neurons using libNeuroML (right) and the equivalent XML representation (left). The network consists of two populations of five cells each. The cell type is a simple integrate-and-fire cell using the NeuroML element *iafCell*. A prototype synapse component is created for this setting all of the

parameters, along with a prototype synapse based on *expTwoSynapse*. A projection is created between the two populations and for each cell in the presynaptic population, an input current is applied, and connections are made to the postsynaptic cells with probability 0.3. The XML on the left is an example of one network instance which can be generated from this script.

```

1  from neuroml import loaders
2
3  doc = loaders.NeuroMLLoader.load('Purk2M9s.nml')
4
5  cell = doc.cells[0]
6
7  #extract the morphology:
8  sample_morphology = cell.morphology
9
10 #sum the total volume:
11 total_volume = 0
12 total_area = 0
13
14 for segment in sample_morphology.segments:
15     total_volume += segment.volume
16     total_area += segment.area
17
18 print 'Total volume: {} um^3'.format(total_volume)
19 print 'Total area: {} um^2'.format(total_area)

```

List 1 | Example of loading a NeuroML multicompartmental cell morphology with libNeuroML and using helper functions to calculate some of the properties (area and volume) of the cell. Methods to calculate area and volume have been inserted into the code generated from the NeuroML Schema.

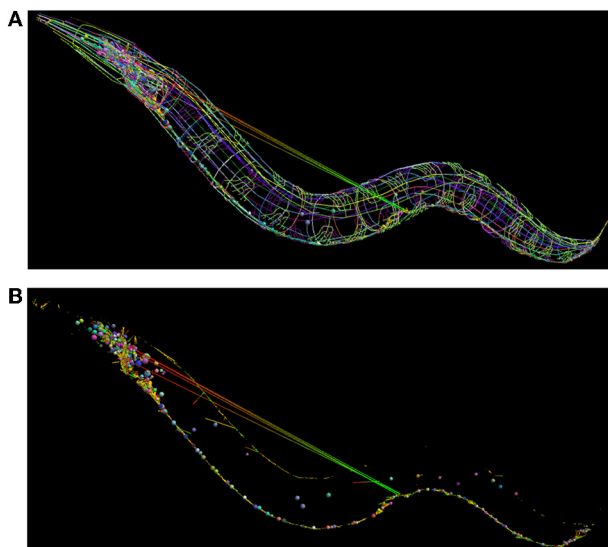


Figure 8 | Use of NeuroML and libNeuroML in the OpenWorm project. (A) The full set of 302 neurons in the OpenWorm model are available in NeuroML format and are visualized here. Each cell, together with its spherical soma, is a different uniform color. Connections between neurons are shown as straight lines varying in color from presynaptic point (green) to postsynaptic point (red). These connections were generated by a Python script using libNeuroML based on connectivity data for pairs of cells. The long straight connections from cells in the midpoint of the worm to the head indicate an error in the connections in the data. (B) Same network as (A), but only the somas of the cells, along with the synaptic connections are shown. The majority of connections are a short length due to the overlap between the neurites of the presynaptic and postsynaptic cells.

provides access via Python to NeuroML, access to the LEMS language is provided by PyLEMS. While Python interfaces exist for many neuronal simulators, giving access to their native object model (e.g., NEURON Carnevale and Hines, 2006) or allowing

new neuron models to be specified (e.g., Brian Goodman and Brette, 2008), this is the first effort at a comprehensive Python suite allowing standards based specification of neuronal model elements from ion channels and synapses to complex networks of cells in 3D, accompanied by programmatic access to underlying model definitions.

Another advantage of these Python APIs is that they provide a way to add functionality that is not provided in the declarative representation. In particular, the current version of NeuroML focuses on the expression of individual, instantiated networks and is less useful for expressing the many probabilistic rules which could be used for network creation. This can be overcome by implementing the network creation process in Python and instantiating the resulting model through the API. Greater support for declarative network templates, including network connectivity rules, is under development for NeuroML 2, but the language is unlikely to match the full flexibility available through a programmatic interface.

INTERACTION WITH PyNN AND OTHER INITIATIVES

A complementary initiative to create a Python API for simulator-independent model specification is PyNN. This API has traditionally concentrated on allowing scripts for large-scale network models with point neurons to be written once and used across multiple simulators with Python interfaces which support these types of model, such as NEST (Gewaltig and Diesmann, 2007), NEURON (Hines and Carnevale, 1997; Carnevale and Hines, 2006; Hines et al., 2009) and Brian (Goodman and Brette, 2008). While the focus of NeuroML v1.x was on multicompartmental, conductance-based cell models, the wider scope of NeuroML v2 has led to increased overlap with PyNN. There are LEMS definitions for the dynamical behavior of the standard PyNN neuron models and corresponding NeuroML elements for these (e.g., *IF_cond_alpha* and *EIF_cond_exp_isfa_ista* in Figure 4), and there is an experimental implementation which allows PyNN scripts to export a full model description in NeuroML v2 format. For example, whereas `from pynn import nest` at the start of a PyNN script indicates that the model should be executed in NEST, `from pynn import neuroml2` indicates that the script should save a declarative description of the model in NeuroML. In addition, code generation to allow all cell/synapse types in NeuroML to be used in PyNN scripts is under active investigation.

NineML (Raikov et al., 2011) is a language for describing new models of spiking neurons in a machine-readable format, which has been developed in parallel to LEMS. NineML shares a number of features with LEMS, allowing new models of spiking neurons to be specified (in the Abstraction Layer). However, a key advantage of LEMS is the close interaction with NeuroML 2, allowing modellers to choose to use either of the languages independently, or to make use of the curated sets of LEMS definitions of standard model types available for NeuroML 2. Creating hierarchical models is also difficult in NineML, but is a key feature of LEMS, required for specifying complex conductance based cell models and ion channels. SpineML (Richmond et al., 2013) is a language that has recently been derived from NineML and shares many of its design choices. LEMS to NineML and

```

import lems.api as lems

model = lems.Model()

hr = lems.ComponentType('hindmarshRoseCell')
model.add(hr)

hr.add(lems.Parameter('a', 'none'))
hr.add(lems.Parameter('b', 'none'))
hr.add(lems.Parameter('c', 'none'))
hr.add(lems.Parameter('d', 'none'))
hr.add(lems.Parameter('I', 'none'))
hr.add(lems.Parameter('r', 'none'))
hr.add(lems.Parameter('s', 'none'))
hr.add(lems.Parameter('x1', 'none'))

hr.add(lems.Constant('SEC', '1s', 'time'))

hr.add(lems.Exposure('x', 'none'))
hr.add(lems.Exposure('y', 'none'))
hr.add(lems.Exposure('z', 'none'))

hr.dynamics.add(lems.StateVariable('x', 'none', exposure='x'))
hr.dynamics.add(lems.StateVariable('y', 'none', exposure='y'))
hr.dynamics.add(lems.StateVariable('z', 'none', exposure='z'))

hr.dynamics.add(lems.DerivedVariable('P', value='y - a * x^3 + b * x^2 + I'))
hr.dynamics.add(lems.DerivedVariable('Q', value='c - d * x^2 - y'))
hr.dynamics.add(lems.DerivedVariable('R', value='s * ( x - x1 ) - z'))

hr.dynamics.add(lems.TimeDerivative('x', '(P - z) / SEC'))
hr.dynamics.add(lems.TimeDerivative('y', 'Q / SEC'))
hr.dynamics.add(lems.TimeDerivative('z', 'r * R / SEC'))

hr0 = lems.Component('hr0', hr.name, a='1', b='3', c='-3', d='5',
                    s='4', I='5', x1='-1.3', r='0.002')

model.add(hr0)

```

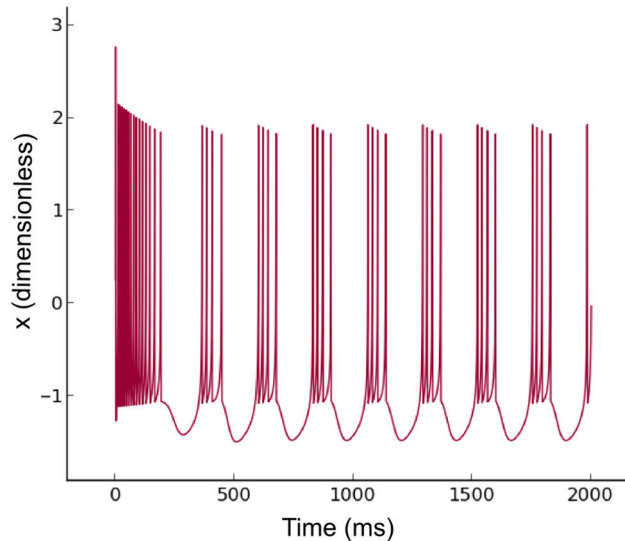


Figure 9 | The Hindmarsh and Rose spiking neuron model. A LEMS *ComponentType* for this model is built adding the three state variables (x , y , z) and the eight fixed parameters required for instances of the model. Time derivatives for the state variables are specified using derived variables to shorten the required expressions. One *Component*

with a particular set of parameters is created. This Python script produces valid LEMS XML which can be used by any LEMS-compliant simulator to simulate the model. The inset shows the behavior of the state variable x (corresponding to the model neuron's membrane potential) when the model is executed in PyLEMS.

SpineML conversion is a feature of the Java based jNeuroML tool (see below), which will assist interoperability between these languages.

libNeuroML AS AN OBJECT-MODEL FOR THIRD-PARTY APPLICATIONS

One potential use of libNeuroML is to provide an object model for third-party applications such as visualization, modeling, and simulation libraries. This is the route taken by the in-development Pyramidal project (<http://pyramidal.readthedocs.org>), which seeks to create an API for running multicompartmental simulations across different simulators. In contrast to libNeuroML, which loads the model elements into an internal object tree for manipulation, Pyramidal interacts with the underlying simulator to store the model in that application's native format. The advantages are that the modeler only deals with model elements defined in NeuroML and interacts with them through libNeuroML, while the simulator uses its own efficient, internal

representation, and scripts developed in this way are portable across simulators.

This approach of using libNeuroML as a library could be used for other applications which only use a subset of the elements of NeuroML, such as an application for visualizing or editing neuronal morphologies or an application for analyzing channel kinetics.

JAVA LIBRARIES FOR NeuroML AND LEMS

In addition to the Python APIs described here for working with NeuroML and LEMS, there are corresponding libraries in the Java language for reading, editing, writing, and validating NeuroML (<https://github.com/NeuroML/org.neuroml.model>) and LEMS (<https://github.com/LEMS/jLEMS>) documents. jLEMS is a more mature package than PyLEMS and is the reference simulator implementation of the LEMS language. These libraries are complemented by other Java packages for importing and

exporting multiple formats such as SBML, NEURON, and Brian into LEMS using code generation. A tool jNeuroML (<https://github.com/NeuroML/jNeuroML>) exists which bundles the functionality of these Java packages, allowing easy access to these features from command line. Options for allowing some of these code generation features to be accessible from Python are under investigation.

FUTURE WORK

Benchmarks indicate that HDF5 is a potentially powerful tool for improving the performance of libNeuroML read/write operations; however, currently, HDF5 can be used to serialize only a limited subset of components defined in NeuroML (reconstructed morphologies). A future avenue of research is the extension of libNeuroML HDF5 serialization to support full NeuroML documents. Additionally, none of the helper functions provided by libNeuroML assist the user with the creation of models of neural networks; we are planning to include this functionality in future versions of libNeuroML.

It should be noted that there are some limitations to the ability of LEMS to describe all models within the scope of NeuroML v2. Currently LEMS does not permit a fully machine-readable, simulator independent description of the equations to solve for neurons with more than one segment, i.e., compartmental modeling. This is planned for future extensions of LEMS.

CONCLUSION

NeuroML and LEMS are key languages which enable cross-simulator portability of models and increase the accessibility and transparency of model properties. Python libraries for reading, writing and manipulating models in these languages are an important step toward encouraging the wider use of these languages. Making stable versions of these libraries available to the computational neuroscience community is and will remain a core part of the release process for NeuroML.

ACKNOWLEDGMENTS

Michael Vella is funded by a Medical Research Council (MRC) Capacity Building Studentship. Michael Vella's initial contributions to libNeuroML were supported by the Google Summer of Code Program 2012, with the INCF as mentoring organization. The contributions of Sharon Crook, Gautham Ganapathy and Robert C. Cannon were supported in part by the US National Institute of Mental Health under grant R01MH061905. The content is solely the responsibility of the authors and does not necessarily represent the official views of the US National Institutes of Health. Contributions of Sharon Crook were also supported in part by grant R01EB014640 from the US National Institute of Biomedical Imaging and Bioengineering. The contributions of Robert C. Cannon, R. Angus Silver, and Padraig Gleeson were supported by the Wellcome Trust (086699/101445). R. Angus Silver is in receipt of a Wellcome Trust Principal Research Fellowship (095667) and an ERC Advanced Grant (294667). The contributions of Andrew P. Davison were supported in part by European Union grant FP7-269921 (BrainScaleS). We would like to thank Boris Marin for assistance in developing the Hindmarsh and Rose model example.

REFERENCES

- Alivisatos, A. P., Chun, M., Church, G. M., Greenspan, R. J., Roukes, M. L., and Yuste, R. (2012). The brain activity map project and the challenge of functional connectomics. *Neuron* 74, 970–974. doi: 10.1016/j.neuron.2012.06.006
- Ascoli, G. A., Donohue, D. E., and Halavi, M. (2007). NeuroMorpho.org: a central resource for neuronal morphologies. *J. Neurosci.* 27, 9247–9251. doi: 10.1523/JNEUROSCI.2055-07.2007
- Bornstein, B. J., Keating, S. M., Jouraku, A., and Hucka, M. (2008). LibSBML: an API Library for SBML. *Bioinformatics* 24, 880–881. doi: 10.1093/bioinformatics/btn051
- Brette, R., and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.* 94, 3637–3642. doi: 10.1152/jn.00686.2005
- Cannon, R., Gewaltig, M., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., et al. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138. doi: 10.1007/s12021-007-0004-5
- Cannon, R., Gleeson, P., Crook, S., and Silver, R. A. (2012). A declarative model specification system allowing NeuroML to be extended with user-defined component types. *BMC Neurosci.* 13(Suppl. 1):P42. doi: 10.1186/1471-2202-13-S1-P42
- Cannon, R. C., Wheal, H. V., and Turner, D. A. (1999). Dendrites of classes of hippocampal neurons differ in structural complexity and branching patterns. *J. Comp. Neurol.* 413, 619–633. doi: 10.1002/(SICI)1096-9861(19991101)413:4<619::AID-CNE10>3.0.CO;2-B
- Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge: Cambridge University Press.
- Crook, S. M., Plesser, H. E., and Davison, A. P. (2013). “Learning from the past: approaches for reproducibility in computational neuroscience,” in *20 Years of Computational Neuroscience*, ed J. M. Bower (New York, NY: Springer), 73–102.
- Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Müller, E., Pecevski, D. A., et al. (2009). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.* 2:11. doi: 10.3389/neuro.11.011.2008
- Gewaltig, M.-O., and Diesmann, M. (2007). NEST. *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430
- Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comp. Biol.* 6:e1000815. doi: 10.1371/journal.pcbi.1000815
- Gleeson, P., Steuber, V., and Silver, R. A. (2007). neuroConstruct: a tool for modeling networks of neurons in 3D space. *Neuron* 54, 219–235. doi: 10.1016/j.neuron.2007.03.025
- 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. doi: 10.1098/rstb.2001.0910
- 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
- Hindmarsh, J. L., and Rose, R. M. (1984). A model of neuronal bursting using three coupled first order differential equations. *Proc. R. Soc. Lond. B Biol. Sci.* 221, 87–102. doi: 10.1098/rspb.1984.0024
- Hines, M., Davison, A., and Müller, E. (2009). NEURON and Python. *Front. Neuroinform.* 3:1. doi: 10.3389/neuro.11.001.2009
- Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209. doi: 10.1162/neco.1997.9.6.1179
- Hodgkin, A. L., and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117, 500–544.
- Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., et al. (2003). The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531. doi: 10.1093/bioinformatics/btg015
- Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Trans. Neural Netw.* 14, 1569–1572. doi: 10.1109/TNN.2003.820440
- Markram, H. (2006). The Blue Brain Project. *Nat. Rev. Neurosci.* 7, 153–160. doi: 10.1038/nrn1848
- Miner, R. (2005). The importance of MathML to mathematics communication. *Not. AMS* 52, 532–538.

Raikov, I., Cannon, R., Clewley, R., Cornelis, H., Davison, A. P., De Schutter, E., et al. (2011). NineML: the network interchange for neuroscience modeling language. *BMC Neurosci.* 12(Suppl. 1):P330. doi: 10.1186/1471-2202-12-S1-P330

Richmond, P., Cope, A., Gurney, K., and Allerton, D. (2013). From model specification to simulation of biologically constrained networks of spiking neurons. *Neuroinformatics* 1–17. doi: 10.1007/s12021-013-9208-z. [Epub ahead of print].

Sauro, H. M., and Bergmann, F. T. (2008). Standards and ontologies in computational systems biology. *Essays Biochem.* 45, 211–222. doi: 10.1042/BSE0450211

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: 01 October 2013; accepted: 25 March 2014; published online: 23 April 2014.
Citation: Vella M, Cannon RC, Crook S, Davison AP, Ganapathy G, Robinson HPC, Silver RA and Gleeson P (2014) libNeuroML and PyLEMS: using Python to combine procedural and declarative modeling approaches in computational neuroscience. Front. Neuroinform. 8:38. doi: 10.3389/fninf.2014.00038

This article was submitted to the journal Frontiers in Neuroinformatics.

Copyright © 2014 Vella, Cannon, Crook, Davison, Ganapathy, Robinson, Silver and Gleeson. 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.