Model descriptions in Neuroscience: computational performance and collaboration

Pedro Garcia-Rodriguez, Andrew P. Davison, Lungsi Sharma, Shailesh Appukuttan

Model descriptions in neuroscience: computational performance and collaboration

Pedro Garcia-Rodriguez*, Andrew P. Davison*, Lungsi Sharma*, Shailesh Appukuttan[1]

[1]**Neurosciences Intégratives et Computationnelles, Centre National de la Recherche Scientifique, 91400 Saclay, France; andrew.davison@cnrs.fr**

Abstract This report reviews the state-of-the-art of model descriptions for large-scale neuronal network models in computational neuroscience, with a particular focus on issues of collaborative model development and of performance on large clusters and supercomputers.
After summarising the requirements for this class of models, and the capabilities of existing simulation tools and existing model description formats, we analyse the shortcomings of existing tools and languages, and make recommendations for future development in this domain; in particular: the use of mixed, standardised text and binary file formats (*e.g.* YAML/JSON with parallel HDF5); alternative/interoperable representations to support a wide range of use cases from algorithm-driven point-neuron networks to data-driven biophysically-detailed networks; and the development of conversion tools to allow gradual convergence without disruption to ongoing projects.

Introduction

In the earliest days of mathematical modelling applied to neuroscience, models consisted of a handful of equations, and solutions were obtained either analytically or with custom computer programs (or even by manual computation where the university mainframe was not available [1, 2]). As models have become more complex, while custom-written code using general purpose programming languages has remained an option [3], there has been a general move towards the use of simulators, i.e. modelling tools that are not specific to an individual model, with their own, domain-specific, high-level programming languages (early examples being NEURON [4], GENESIS [5], XPP [6]; reviewed in [7]).

A further level of abstraction was the development of simulator-independent model description languages, such as NeuroML [8, 9], SBML [10], NineML [11] and PyNN [12]. These languages, often declarative rather than imperative in style, aim to decouple the model specification from the model implementation, with the goals of facilitating reproducibility, cross-simulator verification, and model sharing.

As the ambition of computational neuroscience stretches out towards biologically-detailed, multi-scale, data-driven models of entire brain regions or even entire brains [13, 14, 15], the demands on simulators and on model representations increase, as computational, communication and memory efficiency become critical constraints, as does the requirement for interoperability with other domains of neuroscience, such as connectomics. Simulators, model description formats or languages that are entirely suitable for single neurons or medium-sized networks (up to a few hundred thousand neurons) may have unacceptable performance at larger scales.

In modern supercomputers, memory bandwidth/compute imbalance is becoming prominent and performance can no longer be precisely assessed in terms of number of operations. Other factors are now more critical, including data transfer between parallel processing units and their synchronization, and data movement through the levels of the existing memory hierarchies. Besides, there is a pressing need to optimize the interaction between threads/processes to make it possible that each one can proceed ahead for longer periods with minimal computational-expensive communication. More often, splitting the original problem into smaller pieces of work that can fit in the local (faster) memory is highly recommendable [16].

In order to meet the current challenges imposed by extreme computing, and benefit from those new hardware architectures without sacrificing numerical accuracy/stability, some traditional algorithms have been redesigned. For instance, eingenvalue and sparse matrix computations have been modified to include dimensionality reduction by means of projections onto smaller but highly informative sub-spaces. Besides, random sampling and splitting of the original matrix have been proposed in the field of numerical linear algebra for matrix reduction/factorization, diminishing so

inter-processors communication. Another innovative technique worth to mention is mixed-precision computing, where 64-bit arithmetic is conveniently interleaved with faster 32-bit or 16-bit calculations and precision recovery methods applied to lower-precision arithmetic for computation speedup.

On the other hand, uncontrolled use of random number generators across the available hardware often result in the impossibility to replicate a simulation or stochastic-based optimization results. A less known example is the loss of the expected associative character of a trivial addition operation when it goes through the multi-cores hardware and the complex computing scheduling in a CPU-based system with SIMD (single instruction, multiple data) vectorization pipeline. The result may be different if moved to an accelerator GPU-based hardware operating with SIMT (single instruction, multiple threads) or an hybrid architecture [17]. One of the reasons is that alternative groupings of the individual summands exist, depending of the number of processing units and (dynamic) task organization, which in turn results in floating-point calculations exposed to different round-off error propagations.

Structured, simulator-agnostic model description formats adapted to the performance requirements of modern and future supercomputers will be an essential tool in combatting this. In particular, declarative programming may become an important aspect of HPC systems in the near future, by means of data-flow scheduling techniques [16]. A less imperative coding style might let the system optimize the distribution of the computational tasks across the available hardware and decide a convenient execution pipeline with continuous I/O operations.

Simulation software developers have already begun to address scaling issues for petascale and ultimately exascale supercomputers [18, 19, 20, 21]. The purpose of this review is to consider the same issues for model representations, in particular for simulator-independent representations. The optimisations and customisations required for high-performance computing have long been the enemy of reproducibility [17, 16].

This review is limited to network models in which the individual elements are neurons, communicating with action potentials ("spikes"). We do not consider here either sub-cellular biochemical pathway modelling or models in which the state variables are firing rates or averaged measures of population activity. We begin by describing the more widely-used simulation tools and model representations in this domain. We then itemise the requirements for describing large-scale spiking neuronal network models, and for each requirement examine how well it is met by existing tools, with reference to performance and to ease of collaboration. The review concludes with recommendations for future development in model representation languages and formats.

Overview of simulators and of model representations

A great many tools for simulation in neuroscience have been developed and made available over the last three decades. We consider here those which, in our experience, are the most widely used at present for large-scale, spiking network simulations. We additionally consider a number of tools that either aim to allow model descriptions that are independent of any particular simulator, or that provide an alternative interface to a given simulator. Finally, we consider the model representations used by the Blue Brain Project (BBP) and by the Allen Institute for Brain Science (AIBS), two institutes that have developed internal standards to facilitate large-scale data-driven neuroscience model development by large teams. Here we briefly introduce each of the tools considered in this review.

Neural simulators

NEURON (`http://www.neuron.yale.edu`; [22]) is a simulator designed for networks of biophysically-detailed, multicompartmental neurons. Originally supporting individual desktop computers, it now also runs on large-scale HPC resources. Models of synapses and ion channels are generally written in a domain-specific language, NMODL, although it is also possible to use C or Python. Models of neurons and networks are constructed in Python, in a proprietary language, Hoc, or using a graphical user interface. NEURON supports both thread-based parallelism and distributed computation with MPI.

NEST (`http://nest-simulator.org`; [23]) is a simulator designed for large networks of point-neurons running on large-scale HPC resources, although it is also suitable for smaller networks running on desktop or laptop computers. Neuron and synapse models are written in C++, although a declarative domain-specific language (NESTML) for defining such models is in development. NEST supports both thread-based parallelism and distributed computation with MPI.

Brian (`http://briansimulator.org`; [24]) is a simulator with a focus on ease of use and ease of learning (*a simulator "should not only save the time of processors, but also the time of scientists"*). Models are written in Python, with simple textual representations of equations; as of version 2.0, Brian can make use of run-time generation and compilation of C++ code to accelerate simulations. Brian supports thread-based parallelism, but not distributed computation with MPI.

MOOSE (`https://moose.ncbs.res.in`; [25]), Multi-scale Object-Oriented Simulation Environment allows use of models that combines electrical and chemical signalling (Multi-scale). The conceptual model is mapped into classes. Models are build from these class instances with messages to connect them (Object-Oriented). It can visualize models, using in-house moogli (`http://moose.ncbs.res.in/moogli`; [25]). PyMOOSE is the the python interface. It also supports other model formats. As of version 3.2, these include SBML (`http://sbml.org`; [26]), NeuroML, GENESIS kkit and cell.p formats, and HDF5 and NSDF for writing data.

Model representations

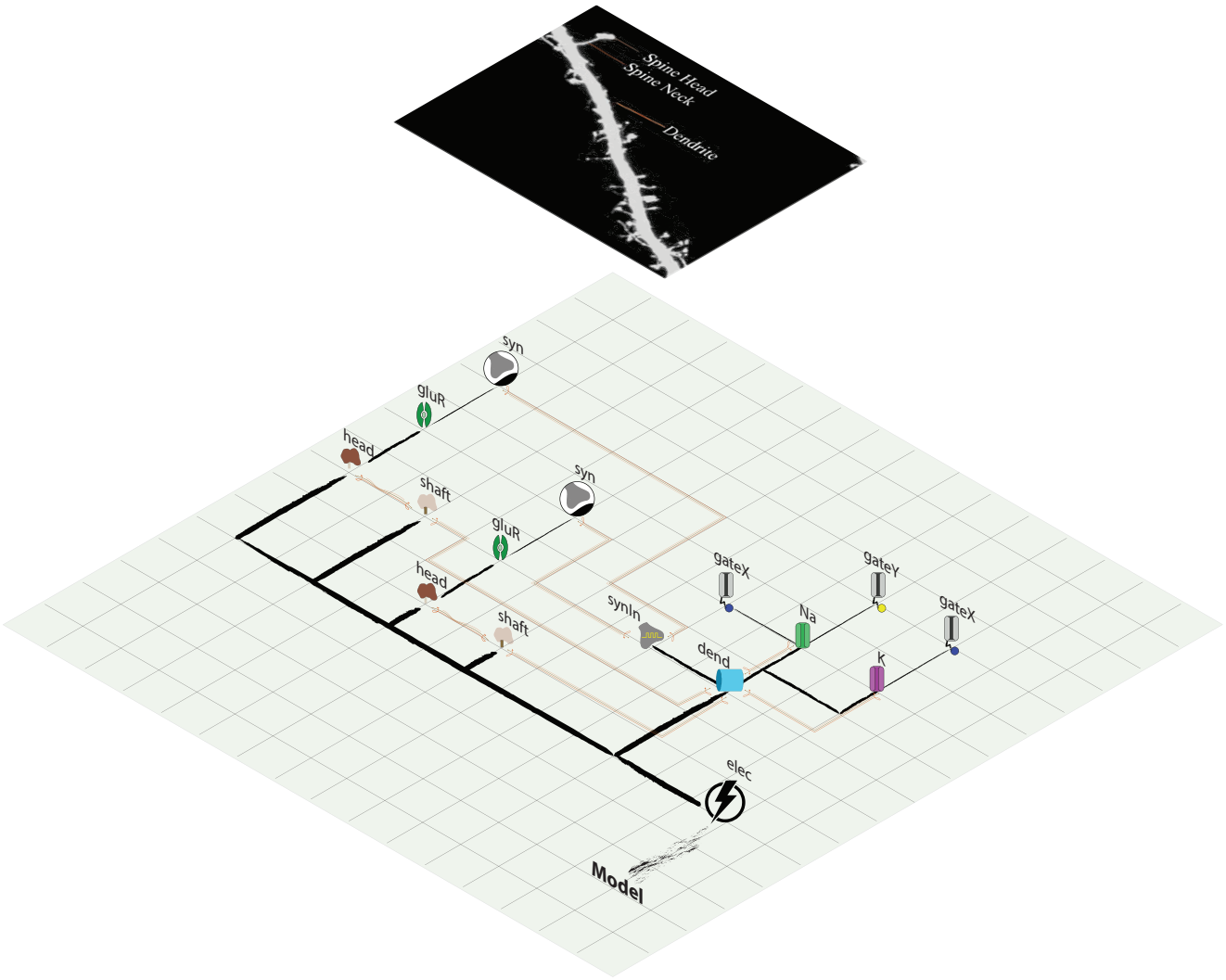
NeuroML (`http://neuroml.org`; [9]) is an XML-based model description language to express models with diverse detail spanning from abstract point-neuron to conductance-based neuron models to morphologically detailed, multi-compartmental neuron models, voltage- and calcium-dependent ion channels. It also supports 3D networks of those neuron models with (fixed or dynamic) synaptic connections between neural populations. As a declarative language, the emphasis is more on the description of the problem under consideration rather than in the specific sequence of operations to solve it, a characteristic of procedural languages. Dynamical behavior of model components were only available in text-based descriptions, not in machine-readable format, but latter versions were developed in conjunction with LEMS to circumvent this problem and avoid overlay verbose specifications (see below). NEURON is able to import and export single cell models from/to NeuroML, increasing reuse and portability of data-driven biophysically detailed models. NeuroConstruct [27] can generate native NEURON code from a NeuroML description. The latest version of MOOSE is NeuroML compliant as well. PyNN can export point-neurons network models to NeuroML. A full description of the elements in NeuroML v2 is available at http://www.neuroml.org/NeuroML2CoreTypes. Each NeuroML release includes W3C XML Schema Document (XSD, `http://www.w3.org/XML/Schema`) to validate NeuroML documents.

LEMS (`http://lems.github.io/LEMS/`; [28]) Low Entropy Model Specification (LEMS) is a declarative XML-based language to describe hierarchical mathematical models of physio-chemical systems. It is a domain-independent language to express machine-readable definitions of mathematical models of diverse complexity in a concise form, with little redundancy. Each model takes the form of a tree of XML elements which can only contains children elements of particular types. Model components definitions by means of generic component types are neatly separated from model implementation/instantiations (specific parameters setting). LEMS also specifies how the model evolves with time, where deterministic or probabilistic transitions between different dynamical regimes can be present. Representation of parameters and state variables as dimensional quantities with automatic handling of units and dimensions is central to LEMS, including consistency checking and transformation between different unit systems. LEMS provides the definitions used in the latest version of NeuroML to describe models in a concise way. LEMS definitions exist for the standard neuron models defined in PyNN and corresponding NeuroML elements for these. New model types which are not part of the core definitions can be created in LEMS, helping to the extensibility of NeuroML. A Java implementation for LEMS (`https://github.com/LEMS/jLEMS`) allows importing/exporting from multiple formats such as SBML, NEURON and Brian into LEMS formats using code generation. Any simulator using NeuroML should follow the LEMS definitions for models simulations in order to be NeuroML compliant. A LEMS-specific XSD exists to validate LEMS descriptions.

NineML (`http://nineml.net`; [11]) is a model description language for computational neuroscience. NineML addresses the issue of specifying large-scale simulations of complex neural connectomes in a simulator-independent way, but far from general-purpose languages descriptions which can lead to maintenance and sharing problems. It currently supports XML-based descriptions for networks of point-neuron/single-compartment models, *i.e.* where axons/dendrites are not explicitly represented. Similarly to LEMS, NineML separates model definition from cell/networks creation in "Abstraction" and "User" specification layers, respectively. Handling of units and dimensions in parameters and state variables is tackled the similar way as in LEMS/NeuroML v2. Tools are available to generate code for NEURON and NEST (*e.g.* pype9), MATLAB, and the neuromorphic SpiNNaker system. A Python library exists to ease the interaction with NineML's code generation tools, for serializations to XML, JSON, YAML and HDF5 formats, and for reading/writing and building/introspecting/manipulating NineML models (`https://github.com/INCF/nineml-python`). With overall goals largely similar to the ones of LEMS and NeuroML v2, they have been developed in parallel and it is likely they will merge in future. A closely related language to NineML is SpineML (`http://spineml.github.io`; [29]).

Blue Brain Project The Blue Brain Project uses NEURON together with Neurodamus as its simulation engine. As such, ion channels and other mechanisms are expressed using NMODL. For simulation purposes, Neurolucida ASCII format is used for morphologies. Passive electrical properties and ion channel distributions are specified in code, using NEURON's Hoc language.

Allen Institute for Brain Science The Allen Institute for Brain Science uses NEURON as its simulation engine, and NMODL for ion channel models. Morphologies are stored in SWC format. Passive electrical properties and ion channel distributions are specified in JSON.




```
model = moose.Neutral('/model')
elec = moose.Neutral(model.path + '/elec')
dend = moose.SymCompartment(elec.path + '/dend')
moose.connect(na, 'channel', dend, 'channel', 'OneToOne')   na = moose.HHChannel(dend.path + '/Na')
                                                            na_gateX = moose.HHGate(na.path + '/gateX')
                                                            na_gateY = moose.HHGate(na.path + '/gateY')
moose.connect(k, 'channel', dend, 'channel', 'OneToOne')    k = moose.HHChannel(dend.path + '/K')
                                                            k_gateX = moose.HHGate(k.path + '/gateX')
                                                            synIn = moose.SpikeGen(dend.path + '/synIn')
                                                            spine1shaft = moose.SymCompartment(elec.path + '/shaft1')
                                                            spine1head = moose.SymCompartment(elec.path + '/head1')
moose.connect(synIn, 'spikeOut', syn1, 'addSpike', 'Single')   spine1gluR = moose.SymCompartment(spine1head.path + '/gluR')
                                                               syn1 = moose.element(spine1gluR.path + '/syn')
                                                            spine2shaft = moose.SymCompartment(elec.path + '/shaft2')
                                                            spine2head = moose.SymCompartment(elec.path + '/head2')
moose.connect(synIn, 'spikeOut', syn2, 'addSpike', 'Single')   spine2gluR = moose.SymCompartment(spine2head.path + '/gluR')
                                                               syn2 = moose.element(spine2gluR.path + '/syn')
moose.connect(dend, 'cylinder', spine1shaft, 'proximalOnly', 'Single')
moose.connect(dend, 'cylinder', spine2shaft, 'proximalOnly', 'Single')
moose.connect(spine1shaft, 'distal', spine1head, 'proximal', 'Single')
moose.connect(spine2shaft, 'distal', spine2head, 'proximal', 'Single')
```

Figure 1. Illustration of model description in MOOSE. The model has five compartments; dendrite, head and shaft (neck) for the two spines.

Application programming interfaces (APIs)

libNeuroML (`https://github.com/NeuralEnsemble/libNeuroML`; [30]) is a Python API which provides a complete and direct mapping between the NeuroML Schema and a Python object model. NeuroML models can be read, parsed and saved by means of libNeuroML. As a procedural model specification that specifies a sequence of steps to be followed (control flow), libNeuroML can handle models whose design boundaries and concepts are not clear or even novel modelling approaches. For instance, libNeuroML can produce much shorter/efficient descriptions and multiple instances of a large network that can be shared, differently to a unique large instance when a NeuroML/JSON encoding is used. Furthermore, this Python library facilitates the integration with other Python packages for simulation, visualization and analysis. Its *arraymorph* Python module substantially simplifies some mathematical analysis (*e.g.* branching analysis) and usual transformations (*e.g.* transpositions) of neuron morphologies as well, by avoiding the recursive traversal of tree-based model data specifications. Additional helper methods present in libNeuroML support general operations with NeuroML models (*e.g.*, computing volumes/areas of cell's branches).

pyLEMS (`https://github.com/LEMS/pylems`; [30]) is a Python API that profits of the advantages of procedural programming for developing/modifying models created with LEMS. pyLEMS helps circumvent the difficulties to read/write declarative XML-based model descriptions, specially overly verbose expressions of complex models. Indeed, declarative languages can not properly deal with large and repetitive model components, but can be succinctly expressed by means of recursion/loops that are part of the characteristic control flow of procedural programming. In addition to accessing the LEMS data model, pyLEMS includes a functionality for validation of files against the LEMS Schema and facilitates the procedural Python creation of new LEMS model types not included in the core definitions of LEMS or NeuroML v2.

Neurodamus (`http://bluebrain.epfl.ch/`; [31]) is an extension framework for neural simulators, which allows the network communication component of a simulator to be replaced by alternative components; this allows linking simulators to separately developed components for monitoring, analysis or simulation control. Neurodamus currently supports the NEURON simulator.

NetPyNE (`http://www.neurosimlab.org/netpyne/`; [32]) is a Python package that aims to make it easier to develop large parallel simulations with NEURON, reducing the level of programming expertise required. NetPyNE uses NMODL for ion channel models, but all other model description components, including morphologies and ion channel distributions, use ordered, nested Python dictionaries, a format syntactically similar to JSON, and structurally similar to NeuroML.

PyNN (`http://neuralensemble.org/PyNN/`; [12]) is a Python API for simulator-independent modelling and simulation. A script written using the PyNN API is expected to run without modification and to give the same results (within limits of numerical accuracy due to different rounding errors, etc., on different implementations) on any supported simulator: at the present time the PyNN API has been implemented for NEST, NEURON, Brian and the two neuromorphic systems SpiNNaker and BrainScaleS. PyNN supports networks of point-neurons; support for multi-compartmental neurons is planned. PyNN supports distributed computation with MPI where the underlying simulator also supports it.

Requirements for model representations

Single neurons

Broadly speaking, we can divide approaches to single cell modelling into three categories:

1. point-neuron models, exemplified by the integrate-and-fire model, in which cell morphology is not represented explicitly, although it may be accounted for implicitly by adjusting the strength and timing of inputs according to their location on the nominal dendritic tree [33]. We may include in this category models with a few spatial compartments that attempt to capture the functional behaviour of dendrites without modelling them in detail [34, 35].

2. compartmental neuron models, in which the dendritic and, sometimes, the axonal trees are represented as chains of isopotential cylindrical or truncated-conical sections to which phenomenological models of ion channels and, sometimes, calcium biochemistry are added. Electrical transmission is simulated using the discretised cable equation; other transport mechanisms, such as longitudinal chemical diffusion, are generally not represented.

3. biochemically detailed models, which take into account both electrical and chemical transport, and model biochemical pathways within the cells in detail with mechanistic models. Such models have a short history compared to the previous two categories, and there is considerable diversity in the approaches taken.

In this report we will consider only the first two categories, due to the greater exploratory nature and lack of standardisation of the third category.

Point-neuron models

Point-neuron models are generally represented by a small number of ordinary differential equations with rules for event handling and for transitions between regimes (where different sets of equations apply in different regimes, such as a refractory regime following a spike). Some simulation environments (*e.g.* NEST, PyNN) have a pre-defined library of models; in this case the user need only specify the model name and parameters. Others (*e.g.* Brian) expect the user to also specify the equations.

Compartmental neuron models

Defining a compartmental neuron model requires the following components: (i) the cell morphology; (ii) a model for each ion channel and for calcium handling (together we will refer to these as mechanisms; some of these models may be dependent on global variables such as temperature); (iii) the passive electrical properties of the cell (*e.g.* membrane capacitance, axial resistance); (iv) the distribution of mechanisms on the dendritic and axonal trees. Synaptic receptors may be specified as part of the neuron model (in the same way as for ion channels), or as part of the network description.

In a network with biologically realistic connectivity, the number of synapses is approximately ten thousand times larger than the number of cells. This means that computations and I/O operations related to synapses are likely to dominate the overall performance of the simulation, and that therefore, while performance issues cannot be neglected since neuron models are in general considerably more complex than synapse models, the principal criterion for choosing a model representation for single neurons should be ease of collaboration (implying ease of model reuse, model sharing, ensuring reproducibility). NMODL and SWC are the most widely used current formats for multicompartmental models; continuing to use them, or adopting them, is clearly a good strategy in the short-to-medium term. It will be important to formalise and standardise use of these formats as much as possible, for example by agreeing on a subset of NMODL (avoiding verbatim C blocks, for example), using JSON Schema, and using the standardised version of SWC used by the `neuromorpho.org` database.

Nevertheless, in the long-term it seems worthwhile moving towards use of NeuroML/LEMS (or possibly NineML, depending on the development velocities of the two projects), since (i) this enables use of a single format (XML) and parser, rather than three separate formats (NMODL-SWC-JSON); (ii) NeuroML (and XML in general) support rich and fine-grained annotations, for example using semantic technologies such as RDF, which is likely to be of considerable importance given the role of models as knowledge integrators; (iii) the principal languages used in sub-cellular modelling, such as SBML, are also XML-based. The gap between these two approaches could be narrowed by NeuroML adopting alternative serialization formats to XML, such as JSON and HDF5.

Neuron morphology descriptions

The cell morphology is represented as a list of cylindrical or truncated-conical sections, typically parameterised by the three-dimensional spatial location of each end of the section, the diameter of each end, and the identity of the parent section (to which a given section is electrically attached). The representation of cell somata is sometimes more complicated. These data are often obtained from digitisation of cell structure ("reconstruction") using light microscopy, and so are provided in the file format produced by the reconstruction software. The most commonly used formats are Neurolucida (MBF Bioscience, Williston, USA) [36] and SWC [37] (see `http://www.neuronland.org/NLMorphologyConverter/FormatStatus.html` for an extensive list of such formats). Simulators such as NEURON and GENESIS have their own formats, although they also generally support other formats . This has seen some uptake ; however many modellers prefer to retain such data in its original format due to (i) specific annotations or non-standard representations of the soma; (ii) the greater simplicity of reading ASCII-based formats compared to XML.

NeuroML NeuroML has attempted to provide a standard format for representing morphology data [38]. As a declarative language, NeuroML can be helpful in model readability and avoidance of model fragmentation. Due to its good interchange format for different software tools, NeuroML can be useful in model interoperability and cross-simulator model validation as well. Indeed, NeuroML's latest version has been built on top of LEMS, a wider framework for model specification and exchange, incorporating tools for reading, writing, simulating and automatic conversion to multiple simulator native formats of most of NeuroML v2 components.

libNeuroML In this API every XML element in NeuroML corresponds to a Python class in such API, since the libNeuroML core object model is generated from a NeuroML schema via the generateDS tool (`https://bitbucket.org/dkuhlman/generateds`) which produces all the necessary type interfaces (Python data structures). Such a process ensures maintainability (rapid update to the latest version of the NeuroML Schema), backward-support (by creating a libNeuroML API for handling older versions of NeuroML) and flexibility to develop new NeuroML components (by

generating a custom copy of the Python API for the Schema changes being tested). Additional libNeuroML modules exist for loading, writing and to validate NeuroML files. XML (NeuroML) or other serialization formats such as JSON, HDF5 and SWC are possible.

Although slowest and least concise, XML serializations should be preferred since it is the format most widely supported by NeuroML compliant software tools. However, complex morphologies can be adequately handled by HDF5 serializations, but internally represented as well in a memory-efficient (NumPy-based) way, that allows handling large-scale morphological and circuit data. As a number of web-based tools and frameworks are optimized to work with JSON files, such a format may be preferred to transmit data over networks. In conjunction with a special libNeuroML module called *arraymorph*, the use of JSON serialization for complex morphologies leads to disk-usage optimization.

LEMS LEMS does not support a simulator-independence of the equations of multi-compartments neurons with more than one segment. LEMS is currently lacking a more sophisticated definition of space than a 3D cartesian coordinates, to accurately simulate for instance conventional cable equations, models with internal reaction-difussion systems or LFPs, as well as formats for complex noise representation and gap junctions.

LEMS model conversion to general-purpose languages such as C, MatLab, Modelica and XPP is also possible, increasing the possibility to explore the performance of dissimilar hardware (*e.g.* SpiNNaker and GPUs), which may lower the memory footprint and speed-up the large-scale simulations. Note, however, that support for some NeuroML v2 components is not supported.

Furthermore, object-oriented related programming principles such as *encapsulation* (containment of components with hierarchical relationships) to encode the concept that one model component is part of another (*e.g.* a gate dynamics is part of an ion channel) and *inheritance* for type refinement or extension to link together similar model types can be found as part of LEMS. Particularly, the child relationship (or containment) of LEMS XML-elements permits that every component has access to just the attributes exposed by its enclosing component higher up the hierarchy. Such a way fixates the internal relationships between model components, whose connections specification get embedded in this nested hierarchical principles as well.

Computer simulations of neuron models

LEMS LEMS is sufficiently generic to provide model descriptions across domains as different computational neuroscience and system biology. A mature Java implementation (`https://github.com/LEMS/jLEMS`) is the reference simulator implementation of the LEMS language, which is complemented with other Java libraries to import/export several formats such as SBML, NEURON and Brian, enabling to fill the gap between separate disciplines dedicated to dissimilar levels of biological description and facilitating cross-simulator model validations, to ease advances in complex multi-scale neuronal modeling. A tool jNeuroML (`https://github.com/NeuroML/jNeuroML`) currently encompasses the functionalities of all these Java packages. jLEMS includes Runge-Kutta four-order methods to integrate the dynamical equations, that in combination with pyLEMS support verification tests of the same model description relative to expected dynamical behaviour. Unfortunately, Runge-Kutta algorithms requires flattening the hierarchical structure of LEMS by removing child elements and adding corresponding scope parameters when this is possible.

NEURON A Python interface is now preferred instead of Hoc original simulator's interpreter, to ensure more portability and compatibility with other neural softwares [39]. Indeed, Python data structures precisely describe neural models in a simulator independent way, for instance in terms of dictionaries [19]. The use of tupples as dictionary keys allows concise descriptions of datasets reducing string manipulation and excessive use of long nested dictionaries. The easiness of Python data structures conversion to other formats facilitates data storage/retrieval/communication and posterior analysis and visualisation of the results on outside platforms, *e.g.* HDF5 files can be imported in MATLAB and MAT files can be created from Python. HDF5 and MAT formats reduce more disk space and saving time than even JSON or pickle serializations of spike-raster data (*e.g.* see Fig 4 of [19]).

Fixed time-step integration algorithms are preferred over adaptive techniques to integrate neuronal dynamics, due to the incessant arrival of events (spikes) from other cells in the network precluding them from showing more frequent larger periods of quiescent (or bursting) activity that can be integrated with longer (or shorter) time scale.

Ion channels descriptions

Ion channel models are represented with a variety of domain-specific, largely declarative, modelling languages, *e.g.* NMODL for NEURON. Due to the difficulty of converting such models between simulators, the International Neuroinformatics Coordinating Facility (INCF) and the NeuroML community have both led efforts to develop standardised formats (NineML and LEMS respectively), which can easily be converted into the representations needed for particular simulators through techniques such as code generation. These formats have so far seen limited uptake; the main barriers appear to be the dominance of the NEURON simulator, the very large number of models already available in NMODL format, and the difficulty in automatically converting NMODL files (which may contain blocks of verbatim C code) into other formats.

The spatial distribution of ion channel and other mechanisms is usually unknown experimentally except at a coarse level of detail, although gene expression studies may radically change this situation in the near future Therefore, while it would be possible to specify the density of ion channels in each electrical compartment individually, as data, it is more common to specify this in code, dividing the soma, dendrites and axons into regions in which the density of each ion channel is homogeneous, or varies continuously with position according to some simple relation.

libNeuroML Serialization into XML or JSON formats preserve all the original NeuroML model data, differently to HDF5 and SWC which are only suited to serialize morphological structure of digitally reconstructed neurons. For instance, any information about the distribution of ion channels or synapses relative to the cell's morphology would be lost if a SWC serialization is applied. With the same drawback, HDF5 provides a low memory footprint and allows rapid I/O operations.

A NeuroML v2/LEMS example is shown in Figure 2. In particular, note that XML lends itself to adding additional annotations of unambiguous scope, such as the neuroLexId attributes, which refer to the terms 'Neuronal Cell Body', 'Dendrite' and 'Spine' in the NeuroLex lexicon [40]. Note also that ion channel models may be of type "conductance density" or "population of individual conductances".

NineML The language supports all levels of modelling from point-neuron/synapse models up to network dynamics and connectivity specification. Hierarchical descriptions allow to build a single component, out of several smaller components, facilitating code reuse and maintainability. However, hierarchical nesting to express complex structures as in LEMS is not supported, requiring that components be explicitly connected by ports, favouring flatter over more structured model descriptions. This also hampers a compact description as additional scoping rules are needed to obtain biologically plausible configurations.

Computer simulations of ion channels

PyLEMS Description of ion channels by a kinetic-scheme are available in LEMS, but are not yet stable in pyLEMS. pyLEMS allows the simulations of most models expressed in LEMS as well by means of a forward Euler integration method. Differently to jLEMS, kinetic schemes are not supported in pyLEMS. LEMS models can be exported to other native formats of faster simulation platforms by means of pyLEMS and jLEMS.

NEURON The kernel has recently been upgraded to reduce the computational cost during integration of multi-compartment neuron dynamics [21]. The new compute engine called coreNEURON is now able to re-group similar gating variables describing different ion channels during the setup phase of the simulation pipeline, which permits an optimal distribution of the calculations across several cores/processors.

Network modelling

Defining a neuronal network model requires the following to be specified: (i) the number of cells of different types to be instantiated; (ii) the position of each cell in space; (iii) the locations and types of synaptic connections between neurons.

The choice of a network specification format or data model is driven primarily by performance considerations. Since networks of neurons may contain many millions or trillions of synapses, and be simulated on many thousands of processors, computational and I/O efficiency is paramount. Unfortunately, the optimal approach depends on the characteristics of the model. Let us consider two extremes. In one extreme scenario every neuron and every synapse is a different model, with a different set of equations, and the connectivity is determined by an explicit list. In another, all neurons and synapses are of the same type, with the same equations and the same parameters, and the connectivity is determined by a simple random rule. The optimal specification in the former case will be based on very large data files, in the latter on a very few parameter values. More realistic models will lie between these extremes, but there is nevertheless a large gap between models in which each neuron has a unique morphology and we have a large number of electrical behaviours, and models with a few populations of neurons, in which each population is defined by a single mathematical model, with only the parameters of equations varying between members of a population. For synapse models, the situation is closer to the second situation for neurons—a limited number of mathematical models, with only the parameters varying from connection to connection.

Populations

As noted above, performance issues for neuron populations are likely to be less critical due to the dominance of simulation performance by the much greater number of synapses. Nevertheless, performance cannot be neglected, and for this reason HDF5 should be preferred to both CSV and XML. Careful thought should be given to designing an HDF5 structure suitable both for point-neuron and morphologically realistic populations.

```
<neuroml xmlns="http://www.neuroml.org/schema/neuroml2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.neuroml.org/schema/neuroml2  ../Schemas/NeuroML2/NeuroML_v2beta4.xsd"
      id="NML2_FullCell">

<!-- Example of a multicompartmental cell with biophysics in NeuroML 2 -->

    <include href="NML2_SimpleIonChannel.nml"/> <!-- Contains ionChannel NaConductance -->
    <ionChannelHH id="pas" conductance="10pS"/> <!--For use in example cell below-->

    <cell id="SpikingCell" metaid="HippoCA1Cell">

        <morphology id="SpikingCell_morphology">

            <segment id ="0" name="Soma">
                <!-- no parent => root segment -->
                <proximal x="0" y="0" z="0" diameter="10"/>
                <distal x="10" y="0" z="0" diameter="10"/>
            </segment>
            <segment id ="1" name="Dendrite1">
                <parent segment="0"/>
                <!-- no proximal => use distal of parent -->
                <distal x="20" y="0" z="0" diameter="3"/>
            </segment>
            <segment id ="2" name="Dendrite2">
                <parent segment="1"/>
                <distal x="30" y="0" z="0" diameter="1"/>
            </segment>
            <segment id ="3" name="Spine1">
                <parent segment="2" fractionAlong="0.5"/>
                <proximal x="25" y="0" z="0" diameter="0.1"/>
                <distal x="25" y="0.2" z="0" diameter="0.1"/>
            </segment>

            <!-- segmentGroups: used for placing ion channels -->

            <segmentGroup id="soma_group" neuroLexId="sao1044911821">
                <member segment="0"/>
            </segmentGroup>
            <segmentGroup id="dendrite_group" neuroLexId="sao1211023249">
                <member segment="1"/>
                <member segment="2"/>
                <member segment="3"/>
            </segmentGroup>
            <segmentGroup id="spines" neuroLexId="sao1145756102">
                <member segment="3"/>
            </segmentGroup>

        </morphology>

        <biophysicalProperties id="bio_cell">

            <membraneProperties>
                <channelPopulation id="naChansDend" ionChannel="NaConductance" segment="2" number="120000"
    erev="50mV" ion="na"/>
                <channelDensity id="pasChans" ionChannel="pas" condDensity="3.0 S_per_m2" erev="-70mV" ion=
    "non_specific"/>
                <channelDensity id="naChansSoma" ionChannel="NaConductance" segmentGroup="soma_group"
    condDensity="120.0 mS_per_cm2" erev="50mV" ion="na"/>
                <specificCapacitance segmentGroup="soma_group" value="1.0 uF_per_cm2"/>
                <specificCapacitance segmentGroup="dendrite_group" value="2.0 uF_per_cm2"/>
            </membraneProperties>

            <intracellularProperties>
                <resistivity value="0.1 kohm_cm"/>  <!-- Used for specific axial resistance -->
            </intracellularProperties>

        </biophysicalProperties>

    </cell>
</neuroml>
```

Figure 2. Example of a multi-compartmental neuron in NeuroML

```
<network id="MultiCompCellNetwork">
        <population id="pop0" type="populationList" component="MultiCompCell">
            <instance id="0">
                <location x="0" y="0" z="0"/>
            </instance>
            <instance id="1">
                <location x="100" y="0" z="0"/>
            </instance>
            <instance id="2">
                <location x="200" y="0" z="0"/>
            </instance>
        </population>
        ...
</network>
```

Figure 3. A fragment of a network model description in NeuroML

NeuroML NeuroML represents cell positions as an explicit list of XML nodes, as shown in Figure 3. The model type, a reference to a cell model defined previously, as discussed above, is an attribute of the parent node. This avoids repeating the reference for every cell in the population; however, the format is otherwise rather verbose. An HDF5 representation of NeuroML is under development to avoid the problem of verbosity with XML.

Blue Brain Project The BBP uses a CSV-based format to hold the specifications of the neurons in the network, one line per cell, containing the position and orientation of each neuron, a reference to the neuron model definition (a Hoc template, as described above), and assorted metadata such as the cortical layer in which the cell body is situated. This is close to the extreme "each neuron an individual" scenario outlined above, except that the number of cell types (for each of which there is a Hoc template) is smaller than the number of cells.

Allen Institute for Brain Science The AIBS also uses a CSV-based format with one line per cell, containing position, orientation and a reference to the model type.

NetPyNE NetPyNE represents information about a given population of cells in a nested dictionary structure. The cell type/model type of a population is a reference to the neuron model nested dictionary described above. It does not appear to be possible in NetPyNE to explicitly specify the position of each neuron; rather cells must be positioned on a grid .

NEURON There is a need for fixed time-step integration algorithms for neuronal dynamics, as mentioned above. This implies that all the cells move forward at the same pace which forces to uniformly distribute the neurons across the nodes to avoid unbalanced memory bandwith and CPU loading. Typically a round-robin (card dealing) approach that sequentially assigns neuron GIDs to each node is used, in case of networks of neurons with similar level of complexity and to randomly distribute populations of similar neuron types of comparable size in the case of hybrid networks as well. Populations types of dissimilar sizes may be better tackled by splitting the round-robin algorithm across each type. Finally, in case of few cells with extreme complexity (*e.g.* large number of compartments with reaction-diffusion mechanisms), its allocation to an individual node may be recommendable to reduce load unbalance on the computational resources [41].

Connections

Given the potentially very-large size of the data tables needed to specify synaptic connectivity in realistic neuronal networks (up to 100 trillion or so entries), it is clear that memory- and I/O-efficient storage is needed. A strong candidate for this, one already adopted by many modelling projects and proven for use in highly-parallel, HPC scenarios by physics simulations, is HDF5. However, the HDF5 API is very extensive, to give maximal flexibility, and it is by no means a minor task to design an optimal specification for the use of HDF5, particularly as the sophistication of our synaptic modelling (and the diversity of synaptic models) increases. Formats that do not currently use HDF5 for network specification (*e.g.*, NeuroML, NetPyNE) are likely need to be adopted in order to scale to large network sizes.

Connection Set Algebra (CSA) CSA is a general formal approach for representing connectivity structure in neural network models, small-scale to large-scale [42]. The concise notation allows scalable and efficient representation. In CSA, a connection set is a collection (list) of ordered pairs (tuple) such that the first element of the pair represents the source of the connection and the second element represents the target. A connection set is therefore a Cartesian product. One may also think of it as a connection matrix. In the software implementation connection sets are by default

infinite.

Masks are considered pure connection sets without any value associated with it. In other words, they are functions that maps non negative integers to a Boolean value. The result is infinite connection sets. The `full` function in the `csa` python package is a mask that returns an infinite connection set with all possible connections (see Figure 4). `block` function is a mask operator that results in representing a set such that each element is an independent block of connection set.

There are different ways to create a finite connection set. The `cross` function performs the Cartesian product of two sets (lists) of finite number of non-negative integers. Thus, returning a finite connection set with all possible connections (see Figure 4 for other variations using `cross`). Another approach is to explicitly define the finite set as a list of desired tuples.

For getting connection set based on geometry CSA does this using three main function types: geometric function, distance metric function and distance dependent masking. The geometric functions (e.g. `random2d` and `grid2d`) transforms the index representation into positions in a unit square (normalized). The distance metric like `euclidMetric2d` defines the Euclidean metric on the grid. Finally, the distance dependent masks (e.g. `disc` and `gaussian`) returns a connection set based on the distance between i and j following a condition defined by the mask. For instance, `disc(r)` connects all i, j such that the measured distance is within a boundary, $d(i, j) < r$ (see Figure 4). CSA can also be used to create connections such that projections occur between different coordinate systems [42].

NeuroML NeuroML represents connections as an explicit list of XML nodes.

PyLEMS The flexibility of Python allows, for instance, that probabilistic rules are used to create a neuronal connectivity patterns, differently to NeuroML that focuses in the description of single instances of model entities. Complex connectivity patterns, including heterogeneous connectivity parameters can easier be coded in PyLEMS than in LEMS.

NetPyNE NetPyNE represents connectivity information in a nested dictionary structure. It is possible either to provide an explicit list of connections or to specify an algorithm for constructing connections based on properties (such as type and location) of the pre- and post-synaptic neurons.

NEURON Each model cell is assigned a unique global identifier (GID) which remains the same regardless the cells distribution setup on the computing system, and to unequivocally identify them during cells creation/placing, stimulation protocols or network wiring.

Blue Brain Project In their model of the cortical column [13], the BBP uses a single model of short-term synaptic plasticity. It is therefore possible to use a fixed-width table of parameters for each neuron ($N \times 19$, where N is the number of synapses per neuron, and 19 is the number of parameters per connection, including the synaptic delay and the parameters of the plasticity model). The HDF5 format is used to store the set of tables. For reasons of efficiency, multiple redundant variants of this file are used for different purposes in the model building/simulation/validation pipeline, for example differing in whether it is structured by afferent or efferent connections.

Allen Institute for Brain Science The AIBS uses a sparse (CSR) representation of the connection and synapse properties matrices, with one dataset per connection property, stored in HDF5. The synapse type is uniquely determined by the types of the pre- and post-synaptic neurons.

Computer simulations of network models

NEURON A *Vector* class of the Hoc interpreter for NEURON eases the recording of state variables by direct pointer access to memory locations, a functionality not present in Python, and randomizers that produce pseudo-random seed-reproducible sequences with longer periods than the ones obtainable in Python. Array indexing provided by NumPy functions is used for fast random access to neuronal state variables in large datasets. Python pickle routines facilitate intermediate saves at fixed times to disk during the simulation of large neural networks that otherwise would overburden the RAM available.

Neuron GIDs are particularly useful to specify independent random number streams for each model cell, which in turn makes it possible to distinguish and re-compute specific simulations based on just the saved spike trains from external nodes and validate the results of the re-simulation, a technique which reduces bottlenecks created by excessive run-time data saving [19].

NEURON optimises various phases of the simulation pipeline. It has included traditional techniques to simulate complex systems on clusters and HPCs resources, such as load balancing, and the use of a main (master) node to handle the data traffic from several cores in moderate network sizes/computing systems and other serial tasks (*e.g.* timing notices). Diverse techniques are used with MPI for run-time spikes passing between cores or nodes involved in the dynamics computation on each node subnet forming the whole network.

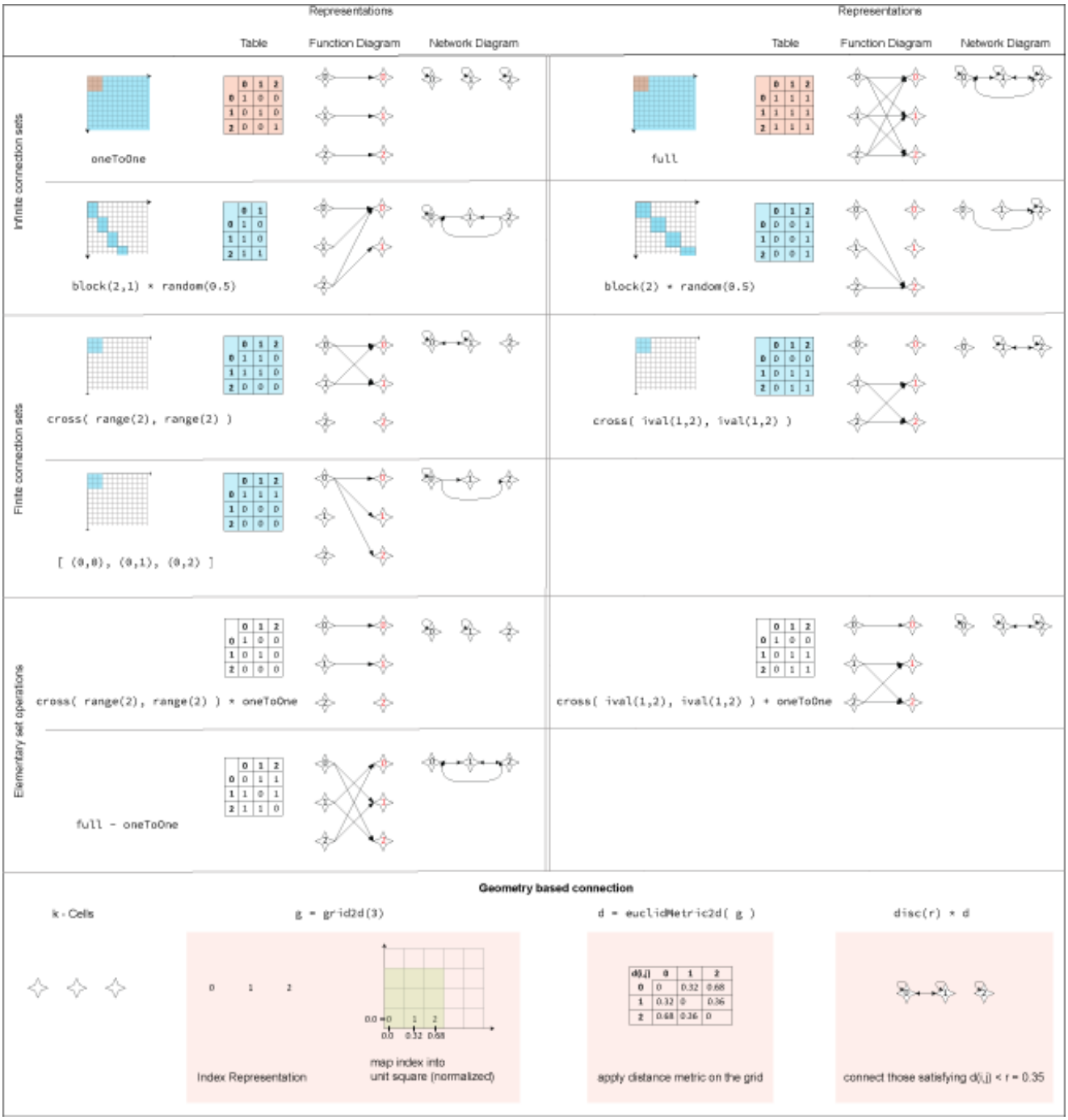


Figure 4. Illustration of CSA. Each cell shows a function command available in the `csa` Python package. The table shows only a connection matrix of specified size for infinite sets. The can be visualizes in `csa` using the command `show( defined_connections_set, 3, 3)` for a 3×3 table. The function diagram shows the function representation of the table. One may think of domain as the source of the connection and its image as the connection target. Notice that the `block` function is a mask operator that results in groups of independent connections, i.e., a subset of the overall connection set. The bottom cell demonstrates the generations of connection based on geometry.

In non-serial implementations of network simulations, connections are defined as having a source and a target which in general can be housed by different cores/nodes of the cluster/HPC system. While the former only specifies threshold/spikes detection mechanisms at specific pre-synaptic cell locations, the latter part of the connection contains the post-synaptic side specification specifying the weights and axonal/synaptic delays. By default, all the spikes generated by a cell are sent to every other computing node, although it is possible to constrain a particular cell to project inside just its local node or provide a list of spike target nodes. However, all-to-all communication seems to not be overburdening unless the number of nodes significantly exceeds the average number of connections per cell [43].

Each processor/core integrates its own subnet dynamics over an interval equal to the minimum pre-synaptic (inter-processor) spike generation to post-synaptic spike delivery connection delay [44]. Synchronization intervals between cores or nodes are (globally) upper-bounded by the shortest inter-processors delay, and the spikes exchange happens only afterwards across all nodes. Future implementations may benefit from different inter-node synchronization time windows, which suits the interactions of different part of the nervous system (*e.g.* spinal cord and brain regions).

Regardless cell complexity, network size or connectivity pattern, usually run-time simulation of fixed-size networks with NEURON linearly scale down with quantity of cores/processor and linearly increase with network's size or cell's complexity (*e.g.* number of compartments) on a single processor. Approximately, a constant run-time is expected when network size increase proportionally to the number of processors [44]. Deviations from any of such ideal execution behaviors may happen when the faster cache memory of the processor is not profited adequately, for instance if just 5-10 cells are houses per node [19], resulting in a larger spikes-mediated communication overhead between the nodes and a slight loss of performance. In case of networks with random delays, inter-node spike exchange time is usually proportional to the number of spikes delivered [44]. Parallel implementations of moderately large networks can show the best performance when split into subnets with complexity fitting in the cache memory of the hosting processor and little use of main memory.

A more precise tuning of a network model can be achieved by configuring NEURON at built time with NEOSIM and NeoCortical simulator programs. Those programs handle more optimally the spike distribution in the network, by means of algorithms carefully tailored to the actual inter-processor connectivity patterns and more efficient use of MPI functionalities.

Data Storage Formats

The field of Information Technology offers a variety of file formats that can be used for data storage and retrieval. These vary in how the data is represented, thereby affecting their performance with regards to parameters such as speed of storage, retrieval and file sizes. We review some of these below, and later compare their performances on the above mentioned parameters.

Text / ASCII: Data can be stored a plain-text file with no specific structure. This would require the target script to be tailor-made for being able to utilize the stored data.

Comma Separated Values (CSV), Tab Separated Values (TSV): These are plain-text file formats that are especially suited to storing tabular data, i.e. data in the form of rows and columns. Each line represents a specific row or record, while columns (or attributes) within each row are separated by a specific delimited. As indicated by their names, for CSV files the delimiter is comma (e.g. see listing 1), while TSV files make use of tabs (e.g. see listing 2).

```
"", "height_mean","height_std"
"Layer 1", "118.271988 um", "10.3625812 um"
"Layer 2", "93.00521788 um", "12.20023253 um"
\caption={}
```

Listing 1. Example of data stored in CSV format

```
""    "height_mean"    "height_std"
"Layer 1"    "118.271988 um" "10.3625812 um"
"Layer 2"    "93.00521788 um"     "12.20023253 um"
```

Listing 2. Example of data stored in TSV format

XLS, XLSX: These are spreadsheet formats developed by Microsoft and made popular through the widespread usage of Microsoft Excel. The older XLS format is a proprietary format owned by Microsoft, while the more recent XLSX makes use of Open XML (see below for XML). Like CSV, TSV, these formats are useful for storing tabular data that can be organized in rows and columns, with the additional feature of tabs to collate multiple such spreadsheets. Fig. 5 shows a snapshot of the tabular representation of data typical of spreadsheet formats such as XLS and XLSX.

eXtensible Markup Language (XML): XML is a markup language that defines a system of rules for annotating documents such that the language syntax is distinguishable the actual data. The primary purpose of this file format was

	A	B	C
1		**Mean Height**	**Mean STD**
2	**Layer 1**	118.271988 um	10.3625812 um
3	**Layer 2**	93.00521788 um	12.20023253 um

Figure 5. Example of data stored in XLS/XLSX spreadsheet format.

targeted at efficient data transfer across the Internet, in a format that is human-readable as well as machine-readable. Listing 3 shows an example of this format.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Layer 1>
    <height>
        <mean>118.271988 um</mean>
        <std>10.3625812 um</std>
    </height>
</Layer 1>
<Layer 2>
    <height>
        <mean>93.00521788 um</mean>
        <std>12.20023253 um</std>
    </height>
</Layer 2>
```

Listing 3. Example of data stored in XML format

JavaScript Object Notation (JSON), YAML (YAML Ain't Markup Language): JSON and YAML are both data serialization languages whereby data structures (objects) are converted into a format appropriate for storage and transmission. Like XML, they are widely used for data transfer across the internet. One important distinction between these and XML is in the usage of minimal syntax to represent the data, thereby making them less verbose. The data is presented in the form of attribute-value pairs. Listing 4 shows an example of data represented in JSON format, while listing 5 shows the same data in YAML format.

```
{
   "Layer 1":{
      "height":{
         "mean":"118.271988 um",
         "std":"10.3625812 um"
      }
   },
   "Layer 2":{
      "height":{
         "mean":"93.00521788 um",
         "std":"12.20023253 um"
      }
   }
}
```

Listing 4. Example of data stored in JSON format

```
---
Layer 1:
  height:
    mean: 118.271988 um
    std: 10.3625812 um
Layer 2:
  height:
    mean: 93.00521788 um
    std: 12.20023253 um
```

Listing 5. Example of data stored in YAML format

Hierarchical Data Format v5 (HDF5): HDF was designed to store large amounts of data in an efficient manner. The current version of HDF is v5 and simply termed HDF5. HDF5 differs significantly in its design and API from its predecessor, HDF4. HDF5 allows files to contain binary data and allows direct access to parts of the file without requiring parsing of the entire file. HDF5 files are not natively in a human-readable form, but can be converted into other formats which allow this. Fig. 6 illustrates the storage of data in HDF5 format.

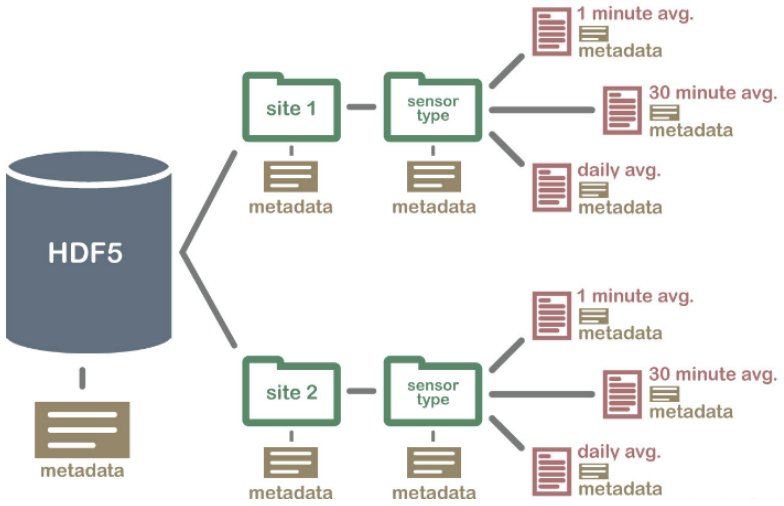


Figure 6. **Representation of data stored in HDF5 format.**
Source: `https://www.neonscience.org/intro-hdf5-r-series`

Table 1. **Comparison of characteristics of various data storage formats**

	self-descriptive	open-source	human-readable
Text	?	Y	Y
CSV, TSV	Y	Y	Y
XML	Y	Y	Y
XLS / XLSX	Y	N / Y	Y
JSON / YAML	Y	Y	Y
HDF5	Y	Y	N

Comparison of Formats

Here we have attempted to provide a quick overview and comparison of the various data storage formats. Table 1 compares certain selection criteria, and we find that most of them possess similar traits. We also ran a simple performance analysis wherein we created dummy data within Python and compared the data write and read times for the various file formats. The analysis was performed on Python 3.6.9 with dummy data generated with a structure similar to that employed for representing internal nodes in SONATA format. Each entry consisted of <"*node_type_id*", "*node_id*", "*rotation_angle_yaxis*", "*x*", "*y*", "*z*", "*ei*", "*model_processing*", "*model_type*", "*model_template*", "*morphology*", "*model_name*">, and the number of entries was varied between 100 and 900,000.

Fig. 7 shows the performance comparison of the various formats for writing the generated data to a local file. In accordance with common programming conventions in Python, the generated data was loaded into a pandas.DataFrame object and this was subsequently written to files with desired data storage formats. It should be noted that pandas didn't offer a built-in method for writing to XML and YAML. We therefore implemented the most commonly advised and adopted approaches to writing these. It is evident from fig. 7 that this significantly affected their performance. The lack of support for these formats can also be construed as a lack of their popularity for data storage. XLSX was also found to yield poor performance, despite having built-in support. All other formats performed comparably well, with JSON and HDF5 formats faring the best.

The performance comparison of the various file formats for reading the data stored above is illustrated in fig. 8. As in the case of WRITE operation, we had to custom implement the desired READ functionality for XML and YAML. YAML and XLSX continued to exhibit poor performance, whilst XML performed well and comparable to most other formats. The best performance was obtained with the use of HDF5 storage format.

Fig. 10 shows how the various file formats compare with regards to the amount of storage space required for saving data. It is notable that XLSX offers the best performance on this front, and this could be attributed to it being a compressed file format. Further, the poor WRITE and READ performance of this format could be an outcome of its compression and decompression requirements. XML, JSON and YAML are seen to demand the most storage space. CSV, TSV and TEXT require very similar storage requirements, owing to their similarity in data representation. HDF5 interestingly shows poor performance for smaller datasets (10k entries), but for larger datasets it offers performance benefits.

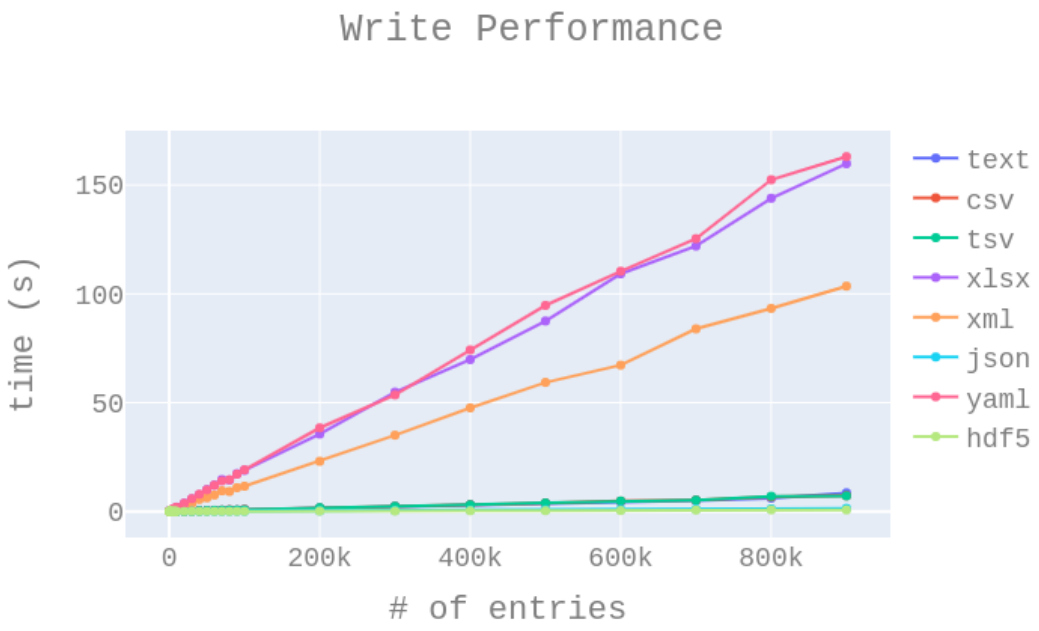


Figure 7. **Performance comparison of various file formats for WRITE operation.**

Figure 8. **Performance comparison of various file formats for READ operation.**

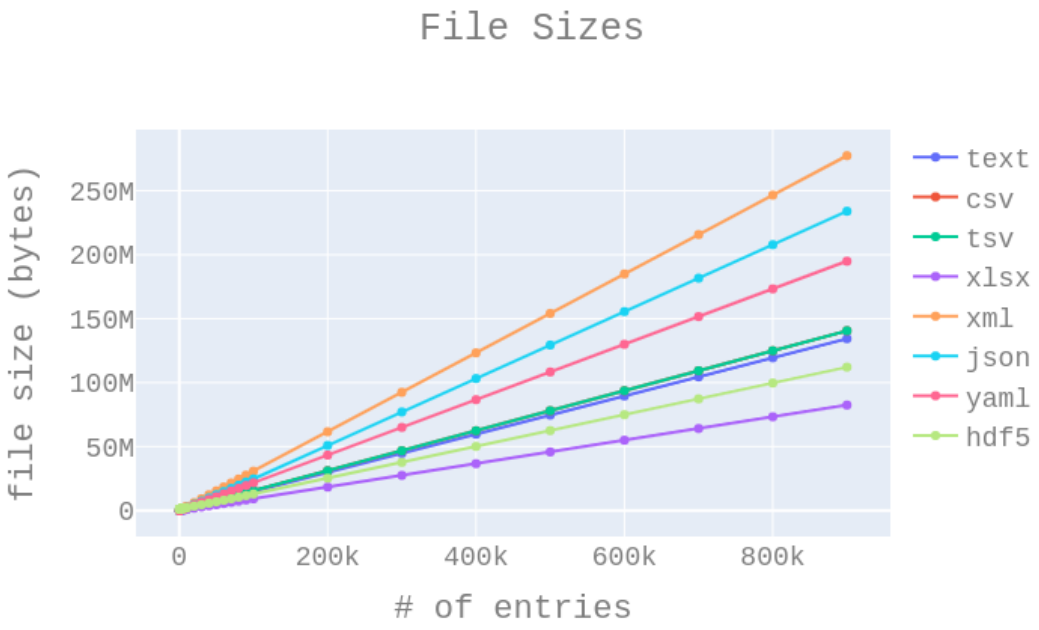


Figure 9. **Performance comparison of various file formats with regards to storage size.**

Model representation across platforms/languages

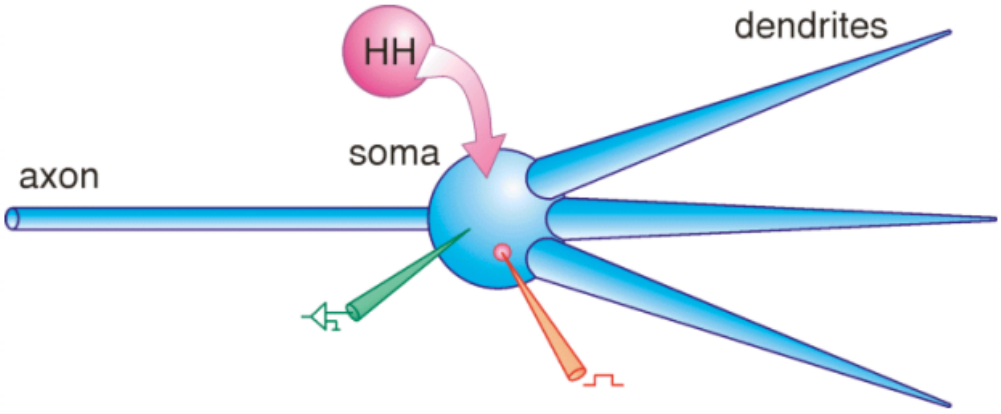


Figure 10. **Conceptual model to be implemented in various simulators.**
Source: `https://www.cns.nyu.edu/~rinzel/CompModelsS09/Remme_Neuron%20demo_03_23_2009.pdf`

Implementation of model in NEURON

```
create soma, dend[3]
connect axon(0), soma(0)

for i=0,2 {
 connect dendrite[i](0), soma(1)
}

soma {
    nseg = 1
    L = 10
    diam = 10
    insert pas
    e_pas = -65
    g_pas = 0.001
    insert hh
```

```
    gnabar_hh = 0.120
}

axon {
    nseg = 20
    L = 300
    diam = 0.5

    insert pas
    e_pas = -65
    g_pas = 0.001

    insert hh
    gnabar_hh = 0.120
}

for i=0,2 dend[i] {
    nseg = 5
    L = 200
    diam(0:1) = 2:0.1

    insert pas
    e_pas = -65
    g_pas = 0.001
}

objref stim
soma stim = new IClamp(0.5)
stim.del = 5
stim.dur = 1
stim.amp = 1

dt = 0.05
tstop = 20
```

Listing 6. Implementation of model in NEURON

Discussion

The aim of this report is to review the adequacy of model representation standards for community models. We take a broad definition of the word "standard", taking it to mean any representation either used by a large number of scientists (whether in separate institutions or within a single large institution such as the Allen Institute for Brain Science) or developed and promoted as a standard by an international body. By "community" model, we mean a model developed collaboratively by more than one research group, and for which the code is shared openly with the scientific community no later than the date of publication of any article describing the model.

We have reviewed the state of the art in describing large-scale networks of point-neurons and/or of biophysically- and morphologically-detailed neurons. Such descriptions may directly use the native programming interface of a specific simulator (Hoc, Python, NMODL for NEURON, Python/C++/NESTML for NEST, Python for Brian), may extend or adapt a specific simulator (Neurodamus, NetPyNE), or may be simulator-independent (PyNN, NeuroML, LEMS, NineML).

Based on the discussion presented in the subsections developed above, in the following we present a number of recommendations.

Recommendations

- Data used in model building should be stored in standard binary formats so as to combine interoperability with performance. In particular, we recommend HDF5; this has the further advantage of being designed for parallel access.
- XML-based model description languages such as NeuroML and NineML should develop additional, interoperable serialisations, for example to JSON, YAML and/or HDF5, to allow more performant data access and more efficient data storage.
- All model description formats should aim to support:
  - both point-neuron and multicompartmental neuron models
  - both algorithmic and data-driven connectivity (and hybrids)

Given this, a representation that aims to maximise both ease-of-use and performance should combine text-based (YAML, JSON) and binary (HDF5) formats.

- Declarative programming may become an important aspect of HPC systems in the near future, by means of dataflow (task) scheduling techniques [16]. It forces a more serial than programmatic style of coding, so taking some control away from the model developer. However, declarative coding might let the system optimize the distribution of the computational tasks across the available hardware and decide a convenient execution pipeline. Serial pieces of code with no side-effects (*e.g.* access to global variables) would be treated as functions whose input/output parameters may determine data dependencies and would fix, at run-time, the multi-core execution of the model components by means of continuous I/O operations between memory and disk. Examples of such schedulers have already been developed for other contexts in Barcelona Supercomputer Center, INRIA and University of Tennessee Knoxville, to name a few.

- Large neuroscience modelling organisations, such as the BBP, HBP, AIBS, and large modelling communities, such as the OpenSourceBrain/NeuroML/OpenWorm community, should endeavour to converge on common formats. To maintain the productivity of existing modelling and simulation pipelines, it will be necessary to develop and maintain tools for conversion between formats. The PyNN package, with extensions for multicompartmental models, is a possible basis for such conversion tools, since it is designed to support "non-purist" model representations that mix standard, simulator-independent formats with native representations for model elements that are not supported by standard formats.

- Furthermore, it is desirable the deployment of HPC software stacks, including simulator kernels enabling automate setting of available algorithms and existing tunable hardware's parameters (*e.g.* amount of shared memory) [16]. This would allow an in-order execution according to the considered hardware (CPUs, GPUs, or hybrid) and vectorization pipeline (SIMD or SIMT). Genetic algorithms and other optimization algorithms might be embedded in those kernels to find the most optimal setting.

- Modern linear algebra techniques already redesigned to target HPC systems may be included into future versions of simulator kernels, to reduce inter-processors communication and main memory access. Solving the cable PDEs equations which describe the APs propagation, by means of finite-differences linearisation, would profit from those improved eingenvalue and sparse matrix computation methods that reduce the problem to be solved by dimensionality reduction, random sampling and matrix splitting.

Grant information

This project received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 720270 (Human Brain Project).

Acknowledgements

A.P. Davison would like to thank Eilif Muller (BBP) and Sergey Gratiy (AIBS) for their presentations at the HBP CodeJam Workshop #7 (`http://neuralensemble.org/meetings/CodeJam7/`). Their talks, entitled respectively "The BBP data model, BluePy and friends" and "Model representation approaches for the Allen mouse visual column", provided much information about the formats used by these two large projects.

References

[1] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. ISSN 0022-3751. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/`.

[2] Christof J Schwiening. A brief historical perspective: Hodgkin and Huxley. *The Journal of Physiology*, 590(Pt 11):2571–2575, June 2012. ISSN 0022-3751. doi: 10.1113/jphysiol.2012.230458. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3424716/`.

[3] Roger D. Traub, Diego Contreras, Mark O. Cunningham, Hilary Murray, Fiona E. N. LeBeau, Anita Roopun, Andrea Bibbig, W. Bryan Wilent, Michael J. Higley, and Miles A. Whittington. Single-Column Thalamocortical Network Model Exhibiting Gamma Oscillations, Sleep Spindles, and Epileptogenic Bursts. *Journal of Neurophysiology*, 93(4):2194–2232, April 2005. ISSN 0022-3077, 1522-1598. doi: 10.1152/jn.00983.2004. URL `http://jn.physiology.org/content/93/4/2194`.

[4] M Hines. A program for simulation of nerve equations with branching geometries. *Int. J. Biomed. Comput.*, 24(1):55–68, 1989. ISSN 0020-7101. URL `http://www.ncbi.nlm.nih.gov/pubmed/2714879`.

[5] J Bower and D Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. Springer, New York, 1997.

[6] Ermentrout, Bard. Simulating, Analyzing, and Animating Dynamical Systems: A Guide to XPPAUT for Researchers and Students. In *SIAM 2002*, Philadelphia, USA, 2002.

[7] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James Bower, Markus Diesmann, Abigail Morrison, Philip Goodman, Frederick Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3): 349–398, 2007. ISSN 0929-5313. URL `http://dx.doi.org/10.1007/s10827-007-0038-6`.

[8] Nigel H. Goddard, Michael Hucka, Fred Howell, Hugo Cornelis, Kavita Shankar, and David Beeman. Towards NeuroML: Model Description Methods for Collaborative Modelling in Neuroscience. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 356(1412):1209–1228, 2001. doi: 10.1098/rstb.2001.0910. URL `http://rstb.royalsocietypublishing.org/content/356/1412/1209.abstract`.

[9] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. *PLoS Comput Biol*, 6(6):e1000815, 2010. doi: 10.1371/journal.pcbi.1000815.

[10] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novère, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, March 2003. ISSN 1367-4803. doi: 10.1093/bioinformatics/btg015. URL `https://academic.oup.com/bioinformatics/article/19/4/524/218599/The-systems-biology-markup-language-SBML-a-medium`.

[11] Ivan Raikov, Robert Cannon, Robert Clewley, Hugo Cornelis, Andrew Davison, Erik De Schutter, Mikael Djurfeldt, Padraig Gleeson, Anatoli Gorchetchnikov, Hans Plesser, Sean Hill, Mike Hines, Birgit Kriener, Yann Le Franc, Chung-Chan Lo, Abigail Morrison, Eilif Muller, Subhasis Ray, Lars Schwabe, and Botond Szatmary. NineML: the network interchange for neuroscience modeling language. *BMC Neuroscience*, 12(Suppl 1):P330, 2011. ISSN 1471-2202. doi: 10.1186/1471-2202-12-S1-P330. URL `http://www.biomedcentral.com/1471-2202/12/S1/P330`.

[12] Andrew Davison, Daniel Brüderle, Jochen Martin Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2(11), 2009. doi: 10.3389/neuro.11.011.2008.

[13] Henry Markram, Eilif Muller, Srikanth Ramaswamy, Michael W. Reimann, Marwan Abdellah, Carlos Aguado Sanchez, Anastasia Ailamaki, Lidia Alonso-Nanclares, Nicolas Antille, Selim Arsever, Guy Antoine Atenekeng Kahou, Thomas K. Berger, Ahmet Bilgili, Nenad Buncic, Athanassia Chalimourda, Giuseppe Chindemi, Jean-Denis Courcol, Fabien Delalondre, Vincent Delattre, Shaul Druckmann, Raphael Dumusc, James Dynes, Stefan Eilemann, Eyal Gal, Michael Emiel Gevaert, Jean-Pierre Ghobril, Albert Gidon, Joe W. Graham, Anirudh Gupta, Valentin Haenel, Etay Hay, Thomas Heinis, Juan B. Hernando, Michael Hines, Lida Kanari, Daniel Keller, John Kenyon, Georges Khazen, Yihwa Kim, James G. King, Zoltan Kisvarday, Pramod Kumbhar, Sébastien Lasserre, Jean-Vincent Le Bé, Bruno R. C. Magalhães, Angel Merchán-Pérez, Julie Meystre, Benjamin Roy Morrice, Jeffrey Muller, Alberto Muñoz-Céspedes, Shruti Muralidhar, Keerthan Muthurasa, Daniel Nachbaur, Taylor H. Newton, Max Nolte, Aleksandr Ovcharenko, Juan Palacios, Luis Pastor, Rodrigo Perin, Rajnish Ranjan, Imad Riachi, José-Rodrigo Rodríguez, Juan Luis Riquelme, Christian Rössert, Konstantinos Sfyrakis, Ying Shi, Julian C. Shillcock, Gilad Silberberg, Ricardo Silva, Farhan Tauheed, Martin Telefont, Maria Toledo-Rodriguez, Thomas Tränkler, Werner Van Geit, Jafet Villafranca Díaz, Richard Walker, Yun Wang, Stefano M. Zaninetta, Javier DeFelipe, Sean L. Hill, Idan Segev, and Felix Schürmann. Reconstruction and Simulation of Neocortical Microcircuitry. *Cell*, 163(2):456–492, October 2015. ISSN 0092-8674. doi: 10.1016/j.cell.2015.09.029. URL `http://www.sciencedirect.com/science/article/pii/S0092867415011915`.

[14] Michael Hawrylycz, Costas Anastassiou, Anton Arkhipov, Jim Berg, Michael Buice, Nicholas Cain, Nathan W. Gouwens, Sergey Gratiy, Ramakrishnan Iyer, Jung Hoon Lee, Stefan Mihalas, Catalin Mitelut, Shawn Olsen, R. Clay Reid, Corinne Teeter, Saskia de Vries, Jack Waters, Hongkui Zeng, Christof Koch, MindScope, Costas Anastassiou, Anton Arkhipov, Chris Barber, Lynne Becker, Jim Berg, Barg Berg, Amy Bernard, Darren Bertagnolli, Kris Bickley, Adam Bleckert, Nicole Blesie, Agnes Bodor, Phil Bohn, Nick Bowles, Krissy Brouner, Michael Buice, Dan Bumbarger, Nicholas Cain, Shiella Caldejon, Linzy Casal, Tamara Casper, Ali Cetin, Mike Chapin, Soumya Chatterjee, Adrian Cheng, Nuno da Costa, Sissy Cross, Christine Cuhaciyan, Tanya Daigle, Chinh Dang, Bethanny Danskin, Tsega Desta, Saskia de Vries, Nick Dee, Daniel Denman, Tim Dolbeare, Ashley Donimirski, Nadia Dotson, Severine Durand, Colin Farrell, David Feng, Michael Fisher, Tim Fliss, Aleena Garner, Marina Garrett, Marisa Garwood, Nathalie Gaudreault, Terri Gilbert, Harminder Gill, Olga Gliko, Keith Godfrey, Jeff Goldy, Nathan Gouwens, Sergey Gratiy, Lucas Gray, Fiona Griffin, Peter Groblewski, Hong Gu, Guangyu Gu, Caroline Habel, Kristen Hadley, Zeb Haradon, James Harrington, Julie Harris, Michael Hawrylycz, Alex Henry, Nika Hejazinia, Chris Hill, Dijon Hill, Karla Hirokawa, Anh Ho, Robert Howard, Jaclyn Huffman, Ram Iyer, Tim Jarsky, Justin Johal, Tom Keenan, Sean Kim, Ulf Knoblich, Christof Koch, Ali

Kriedberg, Leonard Kuan, Florence Lai, Rachael Larsen, Rylan Larsen, Chris Lau, Peter Ledochowitsch, Brian Lee, Chang-Kyu Lee, Jung-Hoon Lee, Felix Lee, Lu Li, Yang Li, Rui Liu, Xiaoxiao Liu, Brian Long, Fuhui Long, Jennifer Luviano, Linda Madisen, Veronica Maldonado, Rusty Mann, Naveed Mastan, Jose Melchor, Vilas Menon, Stefan Mihalas Maya Mills, Catalin Mitelut, Kenji Mizuseki, Marty Mortrud, Lydia Ng, Thuc Nguyen, Julie Nyhus, Seung Wook Oh, Aaron Oldre, Doug Ollerenshaw, Shawn Olsen, Natalia Orlova, Ben Ouellette, Sheana Parry, Julie Pendergraft, Hanchuan Peng, Jed Perkins, John Phillips, Lydia Potekhina, Melissa Reading, Clay Reid, Brandon Rogers, Kate Roll, David Rosen, Peter Saggau, David Sandman, Eric Shea-Brown, Adam Shai, Shu Shi, Josh Siegle, Nathan Sjoquist, Kimberly Smith, Andrew Sodt, Gilberto Soler-Llavina, Staci Sorensen, Michelle Stoecklin, Susan Sunkin, Aaron Szafer, Bosiljka Tasic, Naz Taskin, Corinne Teeter, Nivretta Thatra, Carol Thompson, Michael Tieu, Dmitri Tsyboulski, Matt Valley, Wayne Wakeman, Quanxin Wang, Jack Waters, Casey White, Jennifer Whitesell, Derric Williams, Natalie Wong, Von Wright, Jun Zhuang, Zizhen Yao, Rob Young, Brian Youngstrom, Hongkui Zeng, and Zhi Zhou. Inferring cortical function in the mouse visual system through large-scale systems neuroscience. *Proceedings of the National Academy of Sciences*, 113(27):7337–7344, July 2016. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.1512901113. URL `http://www.pnas.org/content/113/27/7337`.

[15] Katrin Amunts, Christoph Ebell, Jeff Muller, Martin Telefont, Alois Knoll, and Thomas Lippert. The Human Brain Project: Creating a European Research Infrastructure to Decode the Human Brain. *Neuron*, 92(3):574–581, November 2016. ISSN 1097-4199. doi: 10.1016/j.neuron.2016.10.046.

[16] J. Dongarra, S. Tomov, P. Luszczek, J. Kurzak, M. Gates, I. Yamazaki, H. Anzt, A. Haidar, and A. Abdelfattah. With Extreme Computing, the Rules Have Changed. *Computing in Science & Engineering*, 19(03):52–62, may 2017. ISSN 1521-9615. doi: 10.1109/MCSE.2017.48.

[17] K. Diethelm. The Limits of Reproducibility in Numerical Simulation. *Computing in Science Engineering*, 14(1):64–72, January 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2011.21.

[18] Susanne Kunkel, Maximilian Schmidt, Jochen M. Eppler, Hans E. Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann, and Moritz Helias. Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics*, 8, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00078. URL `http://journal.frontiersin.org/article/10.3389/fninf.2014.00078/abstract`.

[19] William W. Lytton, Alexandra H. Seidenstein, Salvador Dura-Bernal, Robert A. McDougal, Felix Schürmann, and Michael L. Hines. Simulation neurotechnologies for advancing brain research: Parallelizing large networks in NEURON. *Neural Comput.*, 28(10):2063–2090, October 2016. ISSN 0899-7667. doi: 10.1162/NECO_a_00876. URL `https://doi.org/10.1162/NECO_a_00876`.

[20] Aleksandr Ovcharenko, Pramod Kumbhar, Michael Hines, Francesco Cremonesi, Timothée Ewart, Stuart Yates, Felix Schürmann, and Fabien Delalondre. Simulating Morphologically Detailed Neuronal Networks at Extreme Scale. In *Parallel Computing: On the Road to Exascale*, pages 787 – 796. 2016. URL `http://doi.org/10.3233/978-1-61499-621-7-787`.

[21] P. Kumbhar, M. Hines, J. Fouriaux, A. Ovcharenko, J. King, F. Delalondre, and F. Schürmann. CoreNEURON: An Optimized Compute Engine for the NEURON Simulator. *arXiv e-prints*, January 2019.

[22] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, 2006.

[23] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[24] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.*, 2:5, 2008. ISSN 1662-5196. doi: 10.3389/neuro.11.005.2008. URL `http://www.ncbi.nlm.nih.gov/pubmed/19115011`.

[25] Subhasis Ray and Upinder S Bhalla. PyMOOSE: Interoperable Scripting in Python for MOOSE. *Front. Neuroinform.*, 2:6, 2008. ISSN 1662-5196. doi: 10.3389/neuro.11.006.2008. URL `http://www.ncbi.nlm.nih.gov/pubmed/19129924`.

[26] Sarah M Keating Aurélien Naldi Martijn P van Iersel Nicolas Rodriguez Andreas Dräger Finja Büchel Thomas Cokelaer Bryan Kowal Benjamin Wicks Emanuel Gonçalves Julien Dorier Michel Page Pedro T Monteiro Axel von Kamp Ioannis Xenarios Hidde de Jong Michael Hucka Steffen Klamt Denis Thieffry Nicolas Le Novère Julio Saez-Rodriguez Claudine Chaouiya, Duncan Bérenguier and Tomáš Helikar. Sbml qualitative models: a model representation format and infrastructure to foster interactions between qualitative modelling formalisms and tools. *BMC Systems Biology*, 7, 2013. ISSN 1752-0509. doi: 10.1186/1752-0509-7-135. URL `http://www.biomedcentral.com/1752-0509/7/135`.

[27] Padraig Gleeson, Volker Steuber, and R. Angus Silver. neuroConstruct: a tool for modeling networks in 3d space. *Neuron*, 54 (2):219–235, 2007. doi: 10.1016/j.neuron.2007.03.025.

[28] Robert C Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. LEMS: A language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in Neuroinformatics*, 8(79), 2014. ISSN 1662-5196.

[29] Paul Richmond, Alex Cope, Kevin Gurney, and David J. Allerton. From model specification to simulation of biologically constrained networks of spiking neurons. *Neuroinformatics*, 12(2):307–323, 2014. ISSN 1539-2791. doi: 10.1007/s12021-013-9208-z. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4003408/`.

[30] M Vella, RC Cannon, S Crook, A Davison, G Ganapathy, HPC Robinson, RA Silver, and P Gleeson. libNeuroML and PyLEMS: using Python to combine imperative and declarative modelling approaches in computational neuroscience. *Frontiers in Neuroinformatics*, 8(38), 2014.

[31] James G. King, Michael Hines, Sean Hill, Philip H. Goodman, Henry Markram, and Felix Schürmann. A Component-Based Extension Framework for Large-Scale Parallel Simulations in NEURON. *Frontiers in Neuroinformatics*, 3, April 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.010.2009. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2679160/`.

[32] S Dura-Bernal, BA Suter, SA Neymotin, CC Kerr, A Quintana, P Gleeson, GMG Shepherd, and WW Lytton. NetPyNE: a Python package for NEURON to facilitate development and parallel simulation of biological neuronal networks. In *CNS 2016*, 2016.

[33] Christian Rössert, Christian Pozzorini, Giuseppe Chindemi, Andrew P. Davison, Csaba Eroe, James King, Taylor H. Newton, Max Nolte, Srikanth Ramaswamy, Michael W. Reimann, Marc-Oliver Gewaltig, Wulfram Gerstner, Henry Markram, Idan Segev, and Eilif Muller. Automated point-neuron simplification of data-driven microcircuit models. *arXiv:1604.00087 [q-bio]*, March 2016. URL `http://arxiv.org/abs/1604.00087`. arXiv: 1604.00087.

[34] P. F. Pinsky and J. Rinzel. Intrinsic and network rhythmogenesis in a reduced Traub model for CA3 neurons. *Journal of Computational Neuroscience*, 1(1-2):39–60, June 1994. ISSN 0929-5313.

[35] V. Booth and J. Rinzel. A minimal, compartmental model for a dendritic origin of bistability of motoneuron firing patterns. *Journal of Computational Neuroscience*, 2(4):299–312, December 1995. ISSN 0929-5313.

[36] J. R. Glaser and E. M. Glaser. Neuron imaging with Neurolucida–a PC-based system for image combining microscopy. *Computerized Medical Imaging and Graphics: The Official Journal of the Computerized Medical Imaging Society*, 14(5):307–317, October 1990. ISSN 0895-6111.

[37] R. C. Cannon, D. A. Turner, G. K. Pyapali, and H. V. Wheal. An on-line archive of reconstructed hippocampal neurons. *Journal of Neuroscience Methods*, 84(1-2):49–54, October 1998. ISSN 0165-0270.

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

[39] Michael L Hines, Andrew P Davison, and Eilif Muller. NEURON and Python. *Front. Neuroinform.*, 3:1, 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.001.2009. URL `http://www.ncbi.nlm.nih.gov/pubmed/19198661`.

[40] Stephen D. Larson and Maryann Martone. NeuroLex.org: an online framework for neuroscience knowledge. *Frontiers in Neuroinformatics*, 7, 2013. ISSN 1662-5196. doi: 10.3389/fninf.2013.00018. URL `http://journal.frontiersin.org/article/10.3389/fninf.2013.00018/abstract`.

[41] Michael L. Hines, Hubert Eichner, and Felix Schürmann. Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors. *Journal of Computational Neuroscience*, 25(1):203–210, Aug 2008. ISSN 1573-6873. doi: 10.1007/s10827-007-0073-3. URL `https://doi.org/10.1007/s10827-007-0073-3`.

[42] Mikael Djurfeldt. The Connection-set Algebra—A Novel Formalism for the Representation of Connectivity Structure in Neuronal Network Models. *Neuroinformatics*, 10(3):287–304, 2012. ISSN 1539-2791. doi: 10.1007/s12021-012-9146-1. URL `http://dx.doi.org/10.1007/s12021-012-9146-1`.

[43] Michael Hines, Sameer Kumar, and Felix Schürmann. Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. *Frontiers in Computational Neuroscience*, 5:49, 2011. ISSN 1662-5188. doi: 10.3389/fncom.2011.00049. URL `https://www.frontiersin.org/article/10.3389/fncom.2011.00049`.

[44] M. Migliore, C. Cannia, W. W. Lytton, H. Markram, and M. L. Hines. Parallel network simulations with NEURON. *Journal of Computational Neuroscience*, 21:119–129, 2006.