The SONATA Data Format for Efficient Description of Large-Scale Network Models

Kael Dai[1], Juan Hernando[2], Yazan N. Billeh[1], Sergey L. Gratiy[1], Judit Planas[2], Andrew P. Davison[3], Salvador Dura-Bernal[4], Padraig Gleeson[5], Adrien Devresse[2], Michael Gevaert[2], James G. King[2], Werner A. H. Van Geit[2], Arseny V. Povolotsky[2], Eilif Muller[2], Jean-Denis Courcol[2], Anton Arkhipov[1,6 *]

[1] Allen Institute for Brain Science, Seattle, WA, USA
[2] Blue Brain Project, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[3] Paris-Saclay Institute of Neuroscience UMR 9197, Centre National de la Recherche Scientifique/Université Paris Sud, France
[4] State University of New York Downstate Medical Center, Brooklyn, NY, USA
[5] Department of Neuroscience, Physiology and Pharmacology, University College London, UK
[6] Lead contact
[*] Correspondence: antona@alleninstitute.org

Abstract

Increasing availability of comprehensive experimental datasets in neuroscience and of high-performance computing resources are driving rapid growth in scale, complexity, and biological realism of computational network models. To support construction and simulation, as well as sharing of such large-scale models, a broadly applicable, flexible, and high-performance data format is necessary. To address this need, we have developed the Scalable Open Network Architecture TemplAte (SONATA) data format. It is designed for memory and computational efficiency and works across multiple platforms. The format represents neuronal circuits and simulation inputs and outputs via standardized files and provides much flexibility for adding new conventions or extensions. We provide reference Application Programming Interfaces and model examples to catalyze support and adoption of the format. SONATA format is free and open for the community to use and build upon with the goal of enabling efficient model building, sharing, and reproducibility.

Introduction

Modern neuroscience is undergoing a profound and rapid evolution thanks to systematic experiments providing comprehensive datasets on composition, connectivity, and *in vivo* activity of brain networks (e.g., (Gouwens et al., 2018a; Jiang et al., 2015; Kasthuri et al., 2015; Lee et al., 2016; Markov et al., 2012; Oh et al., 2014; Tasic et al., 2018; de Vries et al., 2018)). Several initiatives around the world are producing and sharing such comprehensive datasets and call for turning the data into new integrated knowledge via systematic analysis and modeling (Amunts et al., 2016; Bouchard et al., 2016; Hawrylycz et al., 2016; Koch and Jones, 2016; Martin and Chun, 2016; Vogelstein et al., 2016). These widening streams of complex data, together with increasing availability of high-performance computing resources, provide an opportunity and a challenge to the modeling community to contribute new insights via increasingly sophisticated, large-scale, biologically realistic models of brain circuits. To take full advantage of the incredibly rich new experimental data, it will be essential that models take into account the hundreds of cell types, intricate connectivity rules, and complex patterns of neuronal dynamics observed in experiments.

To deliver on these aspirations, the computational neuroscience community needs tools that are up to the challenge. A number of very successful software packages, such as NEURON (Carnevale and Hines, 2006), NEST (Gewaltig and Diesmann, 2007), Brian (Goodman and Brette, 2008), GENESIS (Bower and Beeman, 1997), MOOSE (Ray and Bhalla, 2008), and others provide excellent simulation engines. They supply programming environments that allow for development of very sophisticated models, but traditionally have not supported structured, simulator-independent input formats, where models are defined in standardized files, rather than as software code. Several tools have been developed that attempt to address these issues via interfaces to these simulation engines, for example the Blue Brain's Brion/Brain (https://github.com/BlueBrain/Brion), neuroConstruct (Gleeson et al., 2007), PyNN (Davison et al., 2009), NetPyNE (Dura-Bernal et al., 2019), Open Source Brain (Gleeson et al., 2018), and the Allen Institute's Brain Modeling ToolKit (https://alleninstitute.github.io/bmtk/; (Gratiy et al., 2018). These new tools improve user experience and make modeling more accessible. To achieve that, they typically need to operate with files representing data structures associated with inputs and outputs of models. Consequently, to support interoperability between tools and model reproducibility, a broadly applicable, flexible, and high-performance data format becomes necessary.

Promising solutions have been proposed to address this. They include the XML-based data format NeuroML (Cannon et al., 2014; Gleeson et al., 2010) and the PyNN language (Davison et al., 2009) along with the NSDF standard for simulator output (Ray et al., 2016). Several major challenges, however, remain and are felt particularly acutely in the case of large data-driven models, the scale and complexity of which have increased dramatically in recent years. One problem is a performance bottleneck: Storing data about models containing thousands of neurons and millions of synapses, or beyond, in verbose text-based files is very impractical in terms of storage space and may be challenging for reading and writing simulation data in a parallel computing environment. Hence, a performant binary format is needed. Another is that existing formats describe either static models or simulation outputs, but do not generally describe both of these in one complete data format. And, for broad adoption of a modeling data format, it needs to be flexible enough to represent a variety of model types (point neuron, biophysically detailed, etc.) and compatible with more specialized formats (e.g., SWC for neuronal morphologies (Cannon et al., 1998)), without compromising computational performance.

It should be noted that similar challenges exist in experimental neuroscience (see, e.g., (Koch and Reid, 2012). The situation is clearly improving due to initiatives for experimental data formats, such as NWB (Ruebel et al., 2019), BIDS (Gorgolewski et al., 2016), Loom (https://linnarssonlab.org/loompy), or spacetx-starfish (https://github.com/spacetx/starfish). Nevertheless, aspects of these formats are still being actively developed and redesigned and, being focused on experimental data, they are not suitable for some common aspects of large-scale simulations. All these challenges contribute to difficulties in closing the virtuous experiment/modeling loop and to the overall reproducibility crisis (Baker, 2016; Goodman et al., 2016; Koch and Jones, 2016)).

Here we present the SONATA (Scalable Open Network Architecture TemplAte) data format, which provides an open-source framework for representing neuronal circuits, simulation configurations, and simulation outputs. The format has been jointly developed by the Allen Institute and the Blue Brain Project to facilitate exchange of their large scale cortical models (e.g., (Arkhipov et al., 2018; Markram et al., 2015)), and more recently support for the format has been added by other simulation infrastructure developers. The format is designed for memory and computational efficiency, as well as to work across multiple platforms, and takes advantage of existing and widely used computer data formats such as CSV, JSON, and HDF5 (see Methods). As described below, SONATA's features provide for an extensive, flexible, and efficient ecosystem that is applicable to a wide variety of modeling approaches, including hybrid models (i.e., mixing model elements at different levels of resolution). We also showcase early adoption of the format by various existing modeling software environments. Full specification of the format can be found at the SONATA GitHub page (https://github.com/AllenInstitute/sonata), along with the open-source reference application programming interfaces (APIs) and model examples. To enable broad applications in the field, SONATA is freely available and open to the community to use and build upon.

Results

Overview of the SONATA format

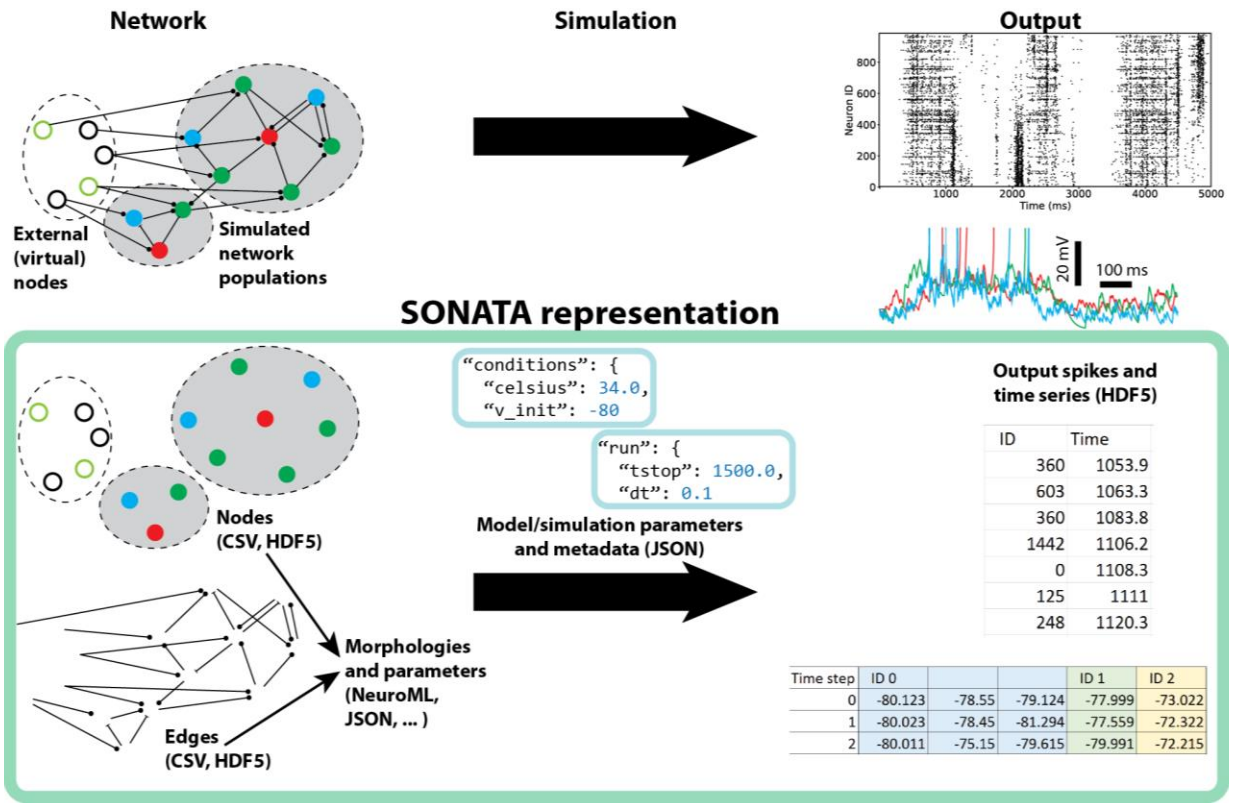
The SONATA format (**Fig. 1**) supports representations of neuronal network models and the output of their simulated activity (e.g. spike times, membrane potential traces). For the former, SONATA contains a network level representation of the individual components and topology of a brain circuit model. It uses existing tabular file formats, HDF5 and CSV, to build a graph representation of the model network of nodes (cells) and edges (synapses) and JSON files to describe metadata (see Methods for details about the HDF5, JSON and CSV formats). For the output, SONATA uses HDF5 to store events (spikes) and time series of continuous variables. Basing SONATA on these widely used formats ensures files can be read/written by existing libraries and applications and used on all major operating systems.

The format accommodates multiple cell and synapse model types and is designed to optimally handle a heterogeneous network. An important design consideration for SONATA has been the flexibility in defining model attributes. Thus, rather than enforcing specific names for the attributes, SONATA provides recipes for storing arbitrary attributes, with some attribute names being reserved for basic standardization and reproducibility.

The major object within the SONATA representation is the model network (**Fig. 1**), which consists of **nodes** of two types: explicitly simulated nodes and virtual nodes (the latter only providing inputs to the simulated system). In both cases, nodes are grouped in one or more **populations** for convenience. Nodes within and between populations are connected via **edges**. SONATA provides a standard for describing properties of individual nodes and edges, as well as types of nodes and edges, and supports reference to additional files containing detailed descriptions of various node/edge attributes, such as neuronal morphology, ion channel properties, synaptic kinetics, etc.

Simulations are performed on model networks, for which one needs to know locations of the files describing network properties and also parameters of simulation (e.g., the time step and temperature). These metadata and parameters are stored in the SONATA configuration ("config") files in JSON format (**Fig. 1**). Finally, the simulation output -- the spikes and time series -- is stored in HDF5 files structured according to the SONATA specification.

Below, we describe the details of these elements of the SONATA format. For the complete description, see **Online Documentation** (https://github.com/AllenInstitute/sonata/blob/master/docs/SONATA_DEVELOPER_GUIDE.md).


Network
Simulation
Output
Neuron ID
Time (ms)
External (virtual) nodes
Simulated network populations
20 mV
100 ms
SONATA representation
Output spikes and time series (HDF5)
Nodes (CSV, HDF5)
Model/simulation parameters and metadata (JSON)
Morphologies and parameters (NeuroML, JSON, ...)
Edges (CSV, HDF5)

```
"conditions": {
  "celsius": 34.0,
  "v_init": -80
```

```
"run": {
  "tstop": 1500.0,
  "dt": 0.1
```

ID	Time
360	1053.9
603	1063.3
360	1083.8
1442	1106.2
0	1108.3
125	1111
248	1120.3

Time step	ID 0			ID 1	ID 2
0	-80.123	-78.55	-79.124	-77.999	-73.022
1	-80.023	-78.45	-81.294	-77.559	-72.322
2	-80.011	-75.15	-79.615	-79.991	-72.215

Figure 1. Overview of the SONATA data format. (Top) A simulated model consists of one or more explicitly simulated network populations and external sources (virtual nodes) that provide inputs into the simulated populations. During and after simulation, output is created characterizing dynamics in the simulated model. (Bottom) The SONATA data format reflects the major components of simulation in dedicated file structures. Information about the model is stored in files (CSV and HDF5) describing nodes and edges of the network (left). Model metadata (e.g., path relations between files on disk) and information about simulation are stored in JSON configuration files (middle). The spiking and time series output is stored in a tabular format taking advantage of the HDF5 technology (right). In the case of time series (bottom right), multiple variables can be stored for individual nodes (in this example, node GID 0 has three variables stored), which can correspond, e.g., to multiple compartments of a neuron.

Node and Edge Types

Both nodes and edges can have **attributes** assigned to them, which can add details of the biological entity to be simulated (e.g. cell or synapse properties). One major benefit of the SONATA format is that it provides flexibility for describing such attributes in terms of storing them individually for each node or edge or more globally for whole subsets of nodes or edges. The latter is achieved via the concepts of **node types** and **edge types**. It is up to the user to decide which attributes are stored on a per-type basis and which should be stored individually for each node or edge. For example, **Table 1** describes a network consisting of five different node types, three of which (node_type_id 100, 101, and 102) are biophysically detailed models and two (node_type_id 103 and 104) are much simpler, point neuron models. Whereas the total number of nodes in this network can be many thousands, the five entries in **Table 1** succinctly describe many attributes of the nodes.

The lists of attributes and instructions for constructing individual nodes are determined by each node type's "model_type" (**Table 1**). A "biophysical" model_type corresponds to a multi-compartment, multi-channel model of an individual cell. The unique compartment sections, intracellular and membrane mechanisms required for building said cell model are described by the "model_template", which may be a reference to a NeuroML2 file or a template file in the hoc format as used in the NEURON simulator (and can be extended to represent other built-in types from a particular simulator software). The reserved "morphology" attribute is used to reference a morphology file (e.g., in the widely used SWC format) to build individual cell compartments in 3D space. The reserved "dynamics_params" file reference can be optionally used to initialize or overwrite electrophysiological attribute values defined by the template. In **Table 1**, node types 100 and 101 are built using custom hoc templates from the Allen Cell Types Database (http://celltypes.brain-map.org), which take parameter values form the JSON files in column "dynamics_params". Node type 102 uses a NeuroML template file (dynamics_params = NONE means that default values are used as set within the NeuroML model_template). Node types 103 and 104 are NEURON built-in IntFire1 point processes, again taking parameter values from the JSON files listed under "dynamics_params". Additional reserved model_type values are "single_compartment" or "virtual", and users may design their own new values.

Table 1: Examples of "node types" and "edge types". In a network model, all individual nodes belonging to a particular node type share the respective attributes, and likewise all edges belonging to the same edge type share attributes of that type.

Node types					
node_type_id	**model_type**	**name**	**model_template**	**morphology**	**dynamics_params**
100	biophysical	Scnn1a	ctdb:Biophys1.hoc	scnn1a_m.swc	472363762_fit.json
101	biophysical	Rorb	ctdb:Biophys1.hoc	rorb_m.swc	473863510_fit.json
102	biophysical	PValb1	nml:PV1.nml.xml	pv1_m.swc	NONE
103	point_neuron	i&f_exc	nrn:IntFire1	NONE	if1_exc.json
104	point_neuron	i&f_inh	nrn:IntFire1	NONE	if1_inh.json

Edge types			
edge_type_id	**model_template**	**dynamics_params**	**delay**
100	exp2syn	biophys_exc.json	2.0
101	exp2syn	biophys_inh.json	2.0

102	NONE	Instantaneous_exc.json	2.0
103	NONE	Instantaneous_inh.json	2.0

Edge types are described in similar ways (**Table 1**). The “model_template” attribute determines the synaptic model via a template file or a synaptic type defined in a particular simulator (e.g., NEURON’s exp2syn), whereas the optional “dynamics_params” initializes or overwrites the parameters of the synaptic mechanisms (e.g., time of rise and decay of synaptic conductance). Other reserved attributes include synaptic weight, delay, and the afferent and efferent locations of synapses. In **Table 1**, only the delays are specified; the other attributes may be specified individually for each edge (see below).

These node type and edge type attributes typically do not require much storage space, since the number of node/edge types in a network model is usually much smaller than the number of nodes/edges (importantly, the distinction between node/edge and node/edge type is up to the user). Therefore, the node/edge type files are stored in the plain-text CSV tabular format (making it easy for modelers to change and update the network *en-masse* through a text editor), whereas attributes of individual nodes and edges are stored in the binary HDF5 format, as described below.

Nodes

A network (see **Fig. 1**) consists of nodes, typically corresponding to cells or spike-generator objects. All nodes are listed, together with their individual attributes (such as the node’s coordinates), in “node tables”, stored as HDF5 files. As discussed, users decide which attributes to store in node-type CSV and which in node table HDF5. For example (**Fig. 2A**), one can store only the coordinates of neurons (x, y, z locations) in the node table with a pointer (the *node_type_id*) to the node types table for repeated information such as morphology (see example in **Table 1**). Alternatively, each neuron may have its own unique morphology (**Fig. 2B**), and in that case the node table contains both the coordinates and the morphology attribute.

SONATA also provides a hierarchy of possible divisions of nodes into *populations* and *groups*. A full network model may contain multiple node populations, with every node belonging to one of the populations. Within a given population, every node is assigned a unique node_id (an unsigned integer). Node populations will usually correspond to a different brain region or a subnetwork within the full network model, and may be created independently of each other. Note that a population may contain different node types (this concept is different from that used in NeuroML and PyNN, where a population is a set of cells of the same type). Within a population, there is one or more node groups, and every node is assigned to one of these node groups. Each node group uses a homogeneous collection of attributes to describe and instantiate each node (the implementation utilizes HDF5 groups and datasets; see **Online Documentation** for details).

Node populations and groups provide flexibility to SONATA applications. Different populations may be stored in different files, allowing modelers to mix, match, and reuse populations between simulations. For example, one may study one cortical area -- say, visual area V1 -- in one simulation and visual area V2 in another simulation, and then build a simulation of V1 and V2 together using two populations (one for V1 and the other for V2), without the need to create new nodes files. Node groups, on the other hand, permit hybrid simulations. For example, our V1 population may consist of both biophysically detailed compartmental neuron models and integrate-and-fire models, which have radically different sets of attributes. Thus, it is practical to store attributes of biophysical cells in one node group and point-neuron cells in the other. Note that nodes of multiple types may be stored in each node group, as long as all the nodes in the node group have the same lists of attributes (but attribute values can be different).

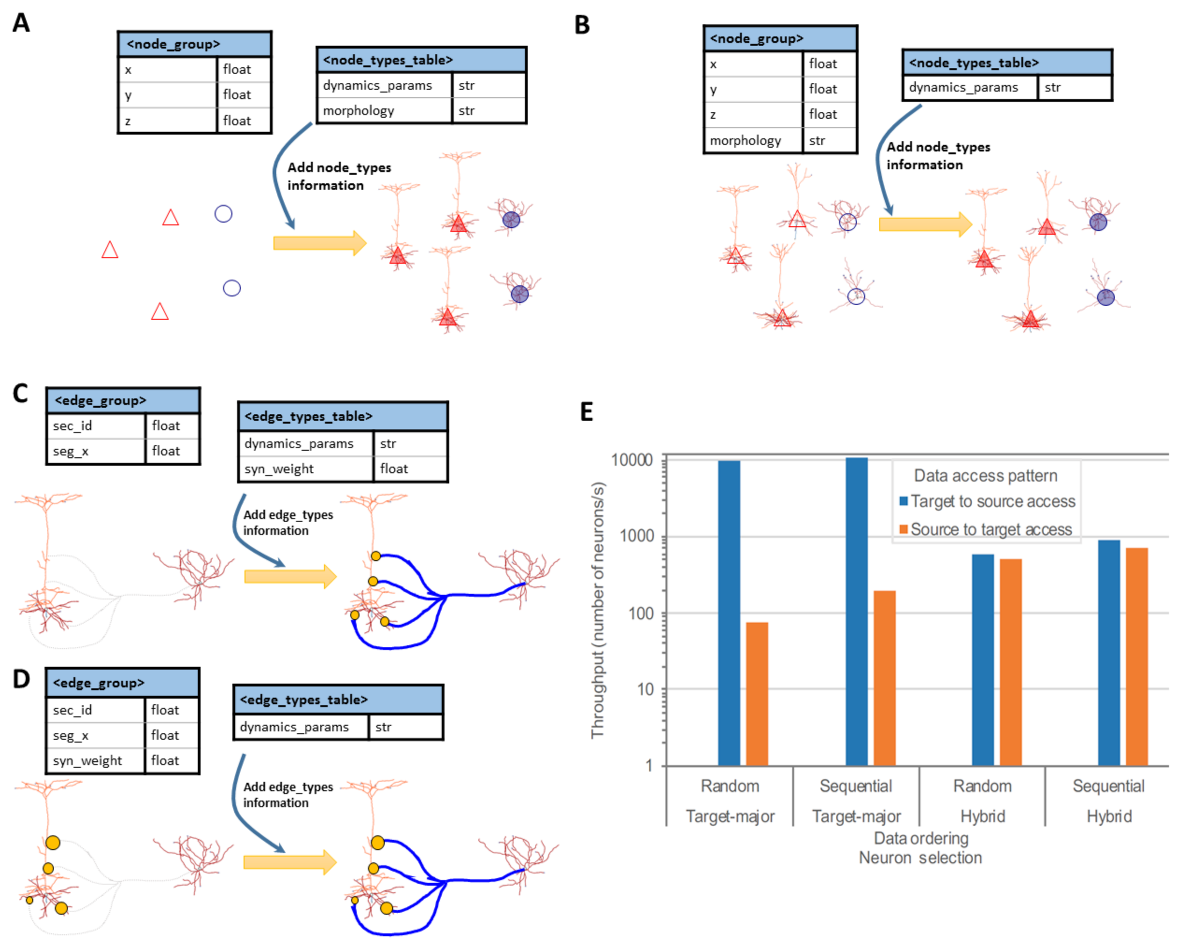


Figure 2. Nodes and edges in SONATA format. (A, B) Two examples are shown that demonstrate how for each node one can find its model attributes in either the node_group (for individually unique attributes) or the node_types table (for globally shared attributes). In (A), the

unique attributes are only the node locations (x, y, z), indicated by empty triangles and circles on the left. Morphology and dynamic parameters are shared among multiple nodes within a type. Hence, all red triangles share the same morphology, as do blue circles (right). In (B), the morphology is unique for each node. The dynamics_params is the only attribute specified at the type level; it is assigned to each node, as indicated by the triangles and circles being filled with color on the right. (C, D) Same for edges. In (C), the synapse locations are stored individually for each edge, whereas synaptic weights and dynamics_params attributes are stored at the edge type level, as indicated by the uniform circle size and colored connections on the right. ("dynamics_params" attributes here determine the dynamical properties of synapses, such as the time of rise and time of decay of synaptic conductance). (D) The synapse locations as well as synaptic weights are stored individually (hence different circle sizes), whereas the dynamics_params attributes are stored at the edge type level. (E) Throughput of accessing edge information for different ordering of edges in the SONATA files -- target-major or hybrid (see text for details) -- in a 45,000-cell model of Layer 4 of mouse V1 (Arkhipov et al., 2018). The target-to-source and source-to-target access patterns are illustrated with either random or sequential selection of target or source neurons.

Edges

An edge typically represents a synapse from one neuron to another. Like nodes, edges are grouped together into **edge populations** (**Fig. 2C, D**), and there can be multiple edge populations per network. Each edge population contains directed connections between nodes in one specified population to nodes in another population (the target and source populations can be the same). Each edge identifies the node_id of the source node and the node_id of the target. There may be multiple edges for a single source/target pair. As with nodes, each edge population consists of one or more edge groups. One edge group contains edges with the same list of attributes.

Continuing our example of a model of V1 and V2 above, one can use one edge population for all connections from V1 to V2, another for V2 to V1, another for V1 to V1, and one more for V2 to V2. The specific partition is again up to users, but has to be consistent with the partition of nodes into populations. Within the V1-to-V1 edge population, one may need to have two edge groups. One edge group would be used for connections to biophysically detailed cell models, containing, for example, attributes of synapse location on the dendritic tree of the target cell, local synapse strength, time delay specific to that particular edge, and many others. The other edge group would be used for connections to point-neuron models, perhaps containing only the synaptic weight. If instead of a hybrid network model a homogeneous network with only one level of resolution for all nodes is used, it is likely that only one edge group in each edge population will be sufficient.

The performance of navigating through an edge file in SONATA format is illustrated in **Fig. 2E** (see **Methods** for details), which shows the results of selecting 1000 neurons and accessing one arbitrary property of all the edges of the selected neurons in the 45,000-cell recurrently connected model of Layer 4 of mouse V1 (Arkhipov et al., 2018) (see below for the description of this model). To illustrate SONATA's performance and flexibility, we use examples of ordering the edges data in two different ways (see **Methods**): target-major (data is sorted according to the ID of the target neuron, increasing) and hybrid ordering (a mixture of blocks with target-major and source-major ordering). We also compare the impact of selecting 1000 neurons randomly or sequentially.

Fig. 2E shows that ordering has an impact on the performance of data access (whereas selecting neurons randomly or sequentially does not impact performance substantially). By using target-major ordering (or its symmetric source-major ordering) one can achieve optimal performance when accessing data in the same access pattern as the ordering, but accessing data in the opposite direction is much less efficient, by a factor of ~100. Ordering data in a hybrid manner is a compromise to get balanced performance between the source-to-target and target-to-source access patterns, but in this case the performance is not as good as the optimal performance for non-hybrid ordering. Due to such large discrepancies, the SONATA format specification leaves the choice of ordering open to users. Note that source-target pairs for each edge are always defined in the edge files in the same way; it is the indexing of these edges that may differ depending on user requirements. This means that the edges can always be read, but reading speed for a particular application will depend on the choice of indexing, and this choice should be made based on the desired application. Examples in **Fig. 2E** indicate that rather high performance can be achieved (close to 10,000 neurons processed per second for their edge attributes) in optimal cases, but users should take advantage of the flexibility of SONATA specification to use edge ordering that is most suitable for their needs. In situations where high performance for various access patterns is essential, solutions may include two or more copies of edge files with different orderings for different use cases.

Simulation Configuration

SONATA provides a framework for storing the information about location of the files describing the model, as well as parameters of the simulation and metadata. This information is stored in the *config* files that tie all the network, circuit, and output components together (**Fig. 1).** The SONATA configuration files, the primary *config*, the *circuit config*, and the *simulation config,* are JSON files containing key/value pairs and **Table 2** lists the keys required in each of these files (more detailed information is available in the **Online Documentation**).

The *circuit config* contains pointers to the files with the information about nodes and edges (including synaptic mechanisms) that describe the network being simulated (i.e. the piece of excitable tissue to be modeled). The *simulation config* on the other hand describes properties unique to a specific simulation run, such as the inputs the network receives, the simulation parameters (for example, duration, time-step), optional parameters such as the temperature, the outputs to be recorded (for instance spike times, membrane potentials, internal calcium concentrations, etc.), paths to writing the outputs, and others. Both the *simulation config* and the *circuit config* may contain a manifest block that defines the paths to be used/reused throughout the JSON file.

Finally, the primary *config* points to the two *configs* described above. Separating the *config* files in this manner gives users flexibility to mix and match models and simulations. For example, one can use a single *circuit config* file and multiple *simulation config* files to run many simulations of one model under different conditions. Alternatively, users can also study multiple circuits under identical conditions using multiple *circuit config* files and one *simulation config* file.

Table 2. Summary of the *config* files. Representative components are listed; additional entries can be used as described in the **Online Documentation.**

***Primary config:* Defines relative location of each part of a network simulation**	
Key	**Description**
network	Defines the network config file
simulation	Defines the simulation config file
***Circuit config:* Defines relative locations of circuit components**	
Key	**Description**
components	Directories for neuron morphologies, synaptic models, mechanisms, and neuron models
network/nodes	Specifies CSV file describing node types (key: node_types_file) and HDF5 file containing individual nodes (key: nodes_file)
network/edges	Specifies CSV file describing edge types (key: edge_types_file) and HDF5 file containing individual edges (key: edges_file)
***Simulation config:* Defines simulation conditions and inputs for the circuit**	
Key	**Description**
manifest	Convenient handle on setting variables that point to base paths
run	Specifies global parameters of the simulation run, such as total duration
conditions	Specifies optional global parameters with reserved meaning associated with manipulation
node_sets	Contains subsets of nodes that act as targets for different reports or stimulations, or can also be used to name and define the target subpopulation to simulate

inputs	Specifies the inputs to the network with a different block for every input (if more than one)
output	Configures the location where output reports should be written, and if output should be overwritten
reports	Defines the specifications of the output variables

Simulation output

In order to facilitate exchange of simulation results in addition to exchange of models, SONATA specifies file formats for representing simulation output, also referred to as **reports**.

Output Format Design

While some existing electrophysiology file formats could in principle be used for simulations (such as NWB:N (Ruebel et al., 2019)), they typically contain features that are optimal for experimental data collection, but counterproductive for simulations. First, electrophysiology experiments typically involve hundreds or thousands of units at most (Jun et al., 2017), whereas in large-scale simulations one can expect to work with networks that are orders of magnitude larger (10^4 neurons or more; see, e.g., (Arkhipov et al., 2018; Bezaire et al., 2016; Markram et al., 2015; Schmidt et al., 2018)). Second, the duration of wet lab recordings can easily be much longer than what is currently possible in simulations. And finally, experimental file formats typically emphasize storage of all possible metadata about the source experiment and the physical setup, which is an unnecessary complexity for typical simulations. These differences lead to substantial design limitations from the simulation standpoint, such as storing each unit as a separate group, which makes data access across units (neurons) very costly.

The SONATA simulation reports are designed to efficiently support three types of data: spike trains, time series for node elements (e.g., membrane voltage or Ca^{2+} concentration in cell compartments) and time series that are not associated with specific node elements (such as voltages recorded with extracellular probes). The file formats are based on HDF5.

The data stored in a spike train report consists of a series of node identifiers and spike times, stored in separate HDF5 datasets. For maximum flexibility, the standard allows the datasets to be sorted according to three different criteria: by node ID, by spike time, or unsorted.

A node element report consists of a set of variables which are sampled at a fixed rate for some elements of interest from a selected set of cells. Typically, the elements are electrical compartments, but other elements can be used as well, such as individual synapses. The time series associated with each element can be membrane voltage, synaptic current, or any other variable. In the report, a simulation **frame** is the set of all values reported at a given timestamp and a **trace** is the full time series of all values associated with one element (**Fig. 3A**). The requirements we followed in designing the node element report were: (i) support for large data sets both in total size (terabytes) and number of elements (millions of cells using multi-compartment models), (ii) random read access to specific frames and elements within a frame, (iii) high performance for different read access patterns (especially full frames and full cell traces) and (iv) high performance sequential parallel writing of full frames. Simplicity of the file format was also highly desirable and was preferred over performance in some aspects.

In the resulting design, data are stored in a single N×M matrix dataset, with rows being frames and columns being traces, whereas extra metadata provides a *mapping* between (cell, element) identifiers and columns within the frame (**Fig. 3A**). The format provides substantial flexibility, in particular permitting one to save different types and amounts of information for different cells. For example, one can choose to save membrane voltage and synaptic currents for all compartments and all synapses for a few cells, only somatic membrane voltage for several other cells, and nothing at all for all the other cells. All this diverse information will be stored easily in the SONATA output file. As such, this design was readily extended to represent also non-cell-element time series reports. In this case, instead of the cell elements, each row represents a channel storing a particular time series -- for example, an electrode at which the extracellular voltage is recorded.

Note that HDF5 provides a storage layout in which the dataset is split into fixed size "chunks" (see **Methods**). Chunking is essential for obtaining good performance with arbitrary access patterns, and for that reason is supported in SONATA. However, SONATA does not prescribe specific chunking, and taking advantage of chunking to optimize read/write performance for specific applications is up to the specific software implementations that use SONATA.

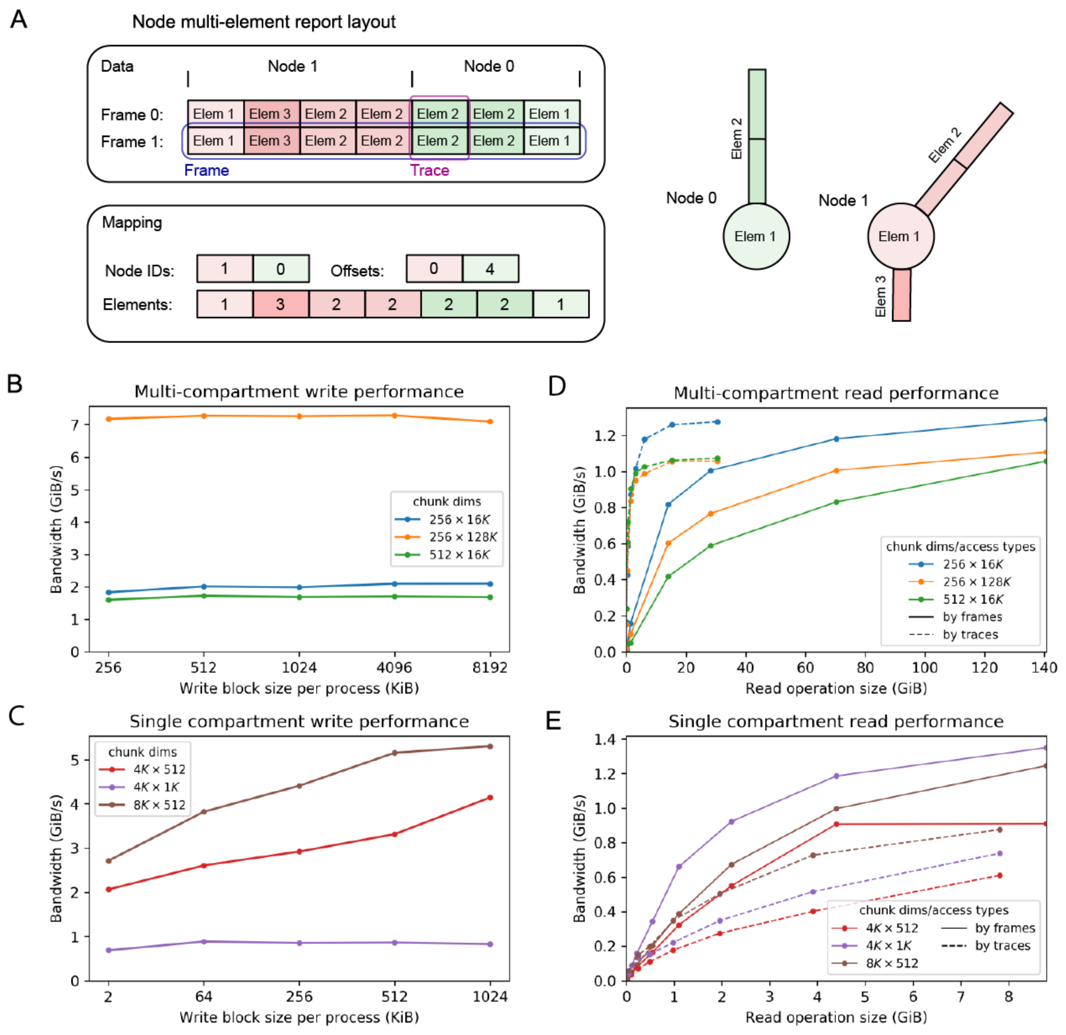


Figure 3. Simulation output in SONATA format. (A) Layout of a multi-compartment report. The dataset is a matrix where each frame (set of values at one point in time) is a row and columns represent traces (the time series of all values associated with one element). All the elements of a node are contiguous within a frame, but nodes may not appear sorted by GID. The position of the first element of each node is indicated by the offset array. Node elements can appear multiple times (e.g. morphological sections with multiple electrical compartments). (B-E) Examples of read/write performance (see **Methods**). Write performance (B, C) and read performance (D, E) of multi-compartment reports (B and D) and single compartment reports (C and E) is measured as bandwidth (amount of data written/read per time unit). Three different HDF5 chunk dimensions (specified in the legend, note that the K suffix indicates multiplication by 1024) were evaluated to demonstrate that high effective bandwidth can be obtained. In the reading evaluation, data was read by frames (continuous lines) and by traces (dotted lines) in single operations of different sizes to demonstrate the flexibility and high performance of the SONATA format; in the writing evaluation, data was only written by frames (continuous lines), which imitates the way most simulators generate data.

Performance benchmarks

In order to demonstrate the viability of the proposed format for simulation output, we measured the performance of writing and reading sample reports by computing the effective I/O bandwidth (amount of useful data read/written per time unit) obtained in both operations on a supercomputer with a distributed file system (IBM's GPFS^{TM}; see **Methods**). We considered different options for our experiments: (i) the amount of data read/written, (ii) HDF5 dataset chunk dimensions for the dataset, (iii) only for write benchmarks — the amount of data written at each write operation (block size per process), and (iv) only for read benchmarks — the direction in which data is accessed (by frames or by traces). We did not consider the latter option in the write benchmark because simulators typically generate data which is ordered temporally, i.e. in frames.

For the multi-compartment report benchmark we used 26,576 neurons (with a total of 41,389,269 reported cell elements), 1,000 time steps, and HDF5 chunk dimensions of 256 $\times$ 16K, 256 $\times$ 128K, or 512 $\times$ 16K. For single compartment report benchmark we used 217,000 neurons, 130,000 time steps, and HDF5 chunk dimensions of 4K $\times$ 512, 4K $\times$ 1K, and 8K $\times$ 512. (The K suffix here means multiplication by 1024.)

Fig. 3B-E shows the effective bandwidth of several write/read operations of different sizes in a collection of single- and multi-compartment reports with different HDF5 chunk dimensions (see **Methods**). In the case of multi-compartment reports (**Fig. 3B**), the HDF5 chunk size determines the effective write performance, almost regardless of the amount of data each process writes at each write operation. We attribute this observation to the overhead caused when using smaller HDF5 chunk dimensions, as the absolute number of HDF5 chunks increases, making the support data structures in the file larger. On the contrary, in single-compartment reports (**Fig. 3C**), the amount of data written by each process at each write operation affects performance. This is because the amount of data written is very small and not efficient for small block sizes. In this case, the performance is also affected by the fact that, in some cases, multiple processes write to the same HDF5 chunk, which leads to lower effective bandwidth (compare 4K $\times$ 512 vs 4K $\times$ 1K).

The read performance tests (**Fig. 3D, E**) were run on a single-node, single-thread configuration, because we consider that this is the typical scenario of analysis and visualization use cases. In all cases, read bandwidth improves as the number of contiguous cells per operation increases. Interestingly, for multi-compartment reports, reading speed improvements plateau quickly when reading by traces as the number of contiguous cells per operation increases.

These results indicate that SONATA is suitable for high-performance computing applications, enabling in these examples both reading and writing of hundreds of GiB in matter of minutes.

An example of a large-scale model: a network model of the layer 4 of mouse cortical area V1

To provide a realistic example of handling large-scale biologically detailed networks with SONATA, we consider the recently published network model of the layer 4 of the mouse primary visual cortex (area V1) (Arkhipov et al., 2018). The model consists of 45,000 neurons (representing more than half of layer 4 neurons in V1) and employs realistic patterns of highly recurrent connectivity. The central portion of the model (**Fig. 4A**) consists of 10,000 neurons modeled using a biophysically detailed, compartmental approach, whereas the remaining 35,000 neurons are modeled using a much simpler point-neuron, leaky integrate-and-fire (LIF) formulation and serve mainly to prevent boundary artifacts. This hybrid model contains ~40 million edges for connections between explicitly modeled nodes and another ~8 million edges from ~10,000 external virtual nodes providing external spiking inputs. One set of virtual nodes provides a background state (representing the rest of the brain) and the other represents the visual pathway from the retina to thalamus to cortex. Consequently, the model could be subjected to a battery of visual stimuli (movies), and results were compared to published work and new in vivo experiments (Arkhipov et al., 2018) (see an example of spiking output in **Fig. 4B**).

Fig. 4C shows benchmarks for loading the layer 4 model in SONATA format (converted to this format from its original pre-SONATA description) for simulation in NEURON (Carnevale and Hines, 2006) using the BioNet module (Gratiy et al., 2018) of the Allen Institute's Brain Modeling ToolKit. We performed these benchmarks on cluster partitions containing from 5 to 390 CPU cores. **Fig. 4C** shows the time required to build the nodes, establish edges from the external virtual nodes, and establish edges among the explicitly simulated, recurrently connected nodes. Two views of the same data are presented: (i) scaling with the number of cores and (ii) scaling with the number of edges or nodes per core. The scaling is approximately linear (with a slope close to 1) starting at about 32 cores. The overall simulation setup time is dominated by the recurrent connections, which are about 5 times more numerous than the virtual input connections and take about 5 times longer to set up (~10 times longer than instantiating nodes).

With a typical partition of hundreds of CPU cores used for simulation, the considerably large layer 4 network model requires <10 s for instantiating nodes, <50 s for external edges, and ~4 minutes for recurrent edges, resulting in ~5-minute setup time total. Using uncompressed HDF5 files, the total size of network files, including recurrent and feedforward network connections, is just under 2.5 GB. As noted above, the setup time scales very well with number of CPU cores, whereas our previous work also demonstrated good scaling of time required for actual simulations of such a model (see details in (Gratiy et al., 2018)).

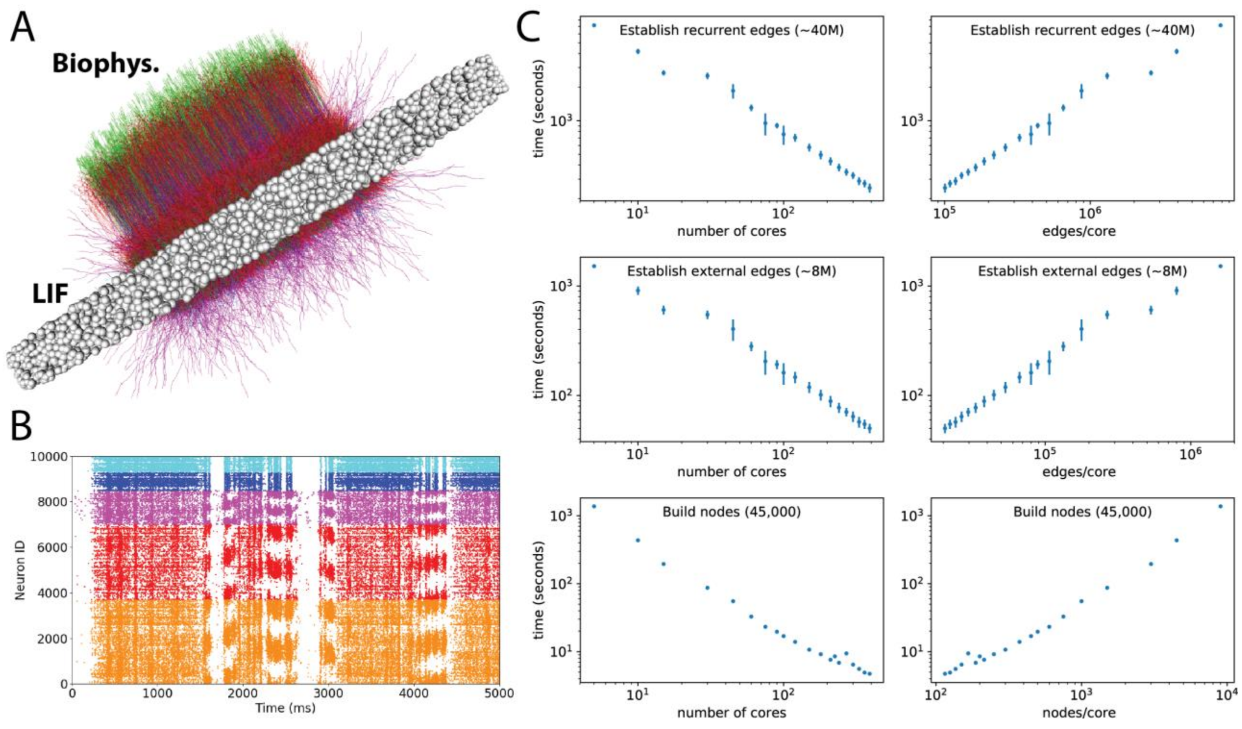


Figure 4. A 45,000-neuron hybrid network model of the layer 4 of mouse cortical area V1. (A) Visualization of the network model, which consists of 10,000 biophysically detailed neurons (colored morphologies) in the center and 35,000 point neurons (white balls) forming an annulus around the biophysical neurons to prevent boundary artifacts. (B) An example raster plot output from a simulation of the layer 4 model. Shown are the spikes of 10,000 biophysical neurons in response to a clip from a natural movie. Colors indicate the five types of neurons: excitatory Scnn1a (orange), Rorb (red), Nr5a1 (magenta) and inhibitory PV1 (blue) and PV2 (cyan). See details in (Arkhipov et al., 2018). (C) Benchmarks for instantiating different parts of the layer 4 model. The left and right column show the same data: against the number of CPU cores used for simulation on the left and against the number of edges or nodes per core on the right.

Ecosystem support

SONATA is intended for broad use in the field of computational neuroscience and is provided as a free format, open for community development and evolution. Anyone desiring to implement SONATA may utilize the PySONATA Python API hosted at GitHub and developed jointly by the Allen Institute and Blue Brain Project (BBP). It is open for community collaboration. The Allen Institute's and BBP's tools that already provide SONATA support are the modeling suites BMTK and Brion/Brain, as well as the library libSONATA and visualization tool RTNeuron. Additional tools from other modelling and standardization initiatives — NeuroML, PyNN, and NetPyNE — also implement SONATA support (**Fig. 5**).

Below we briefly describe examples of using these tools to construct, read, write, visualize, and simulate network models in SONATA format. Note that, in general, when different simulators load one SONATA model for simulation, bitwise agreement between their outputs cannot be guaranteed. The reasons for that are non-standardized processing of certain data in different simulators, or even within the same simulator, different approaches for instantiating initial conditions, etc. For a real-life example, consider that loading SWC morphologies can be done using different functions (e.g., hoc or Python), which employ different numerical precisions; as a result, simulation outputs will not be bitwise identical, but will be only statistically the same to the level permitted by the precision discrepancy in morphologies. Nevertheless, SONATA constrains a vast variety of important degrees of freedom in network simulations, enabling statistically similar results between simulators and bitwise reproducibility within a simulator with fixed software code.

PySONATA

PySONATA is a Python based API for reading node and network files that have been saved in the SONATA format. It has been open-sourced under a BSD license and maintained as an official tool of the SONATA working group, hosted on GitHub (https://github.com/AllenInstitute/sonata). Users wishing to begin integrating the SONATA format into their own software are encouraged to use the PySONATA Python modules. Examples of how to use the module can be found at https://github.com/AllenInstitute/sonata/blob/master/src/pysonata/docs/Tutorial%20-%20pySONATA.ipynb.

The Brain Modeling Toolkit

The Brain Modeling Toolkit (BMTK; https://github.com/AllenInstitute/bmtk) is a Python based package for building, simulating and analyzing large-scale neural networks across different levels of resolution. The BMTK is open-sourced under a BSD-3 license and has full support for generating and reading the SONATA data format (**Fig. 5**). Modelers can use the BMTK Builder submodule to create their own SONATA based networks from scratch. It supports cell template files, electrophysiological parameters, and morphology from the Allen Cell Types Database (http://celltypes.brain-map.org/) (Gouwens et al., 2018b; Teeter et al., 2018) as well as other cell model formats, including NeuroML2 (Cannon et al., 2014; Gleeson et al., 2010), NEURON hoc files (Carnevale and Hines, 2006), or even user defined Python functions. For simulations, BMTK relies on an increasing array of simulation engines (NEURON (Carnevale and Hines, 2006), NEST (Gewaltig and Diesmann, 2007), Dipde (Cain et al., 2016), etc.), which allow users to run simulations of SONATA networks using either multi-compartment, point, or population based representations. The results of these simulations, regardless of the underlying simulator used to run them, are transformed into SONATA output format, allowing networks built and run with BMTK to be analyzed and visualized by any third-party software that supports SONATA. **Fig. 5B** and **5C** show a network with 300 biophysically detailed cells, in SONATA format, generated using BMTK and visualized with RTNeuron and NetPyNE, respectively. The results of simulations of this network using BMTK and NetPyNE are shown in **Fig. 5D**. **Fig 5E** shows simulations of a network of 300 integrate and fire neurons created with BMTK, PyNN and pyNeuroML.

Brion/Brain

The Blue Brain's C++ libraries for handling large scale data and simulation setup, Brion/Brain (https://github.com/BlueBrain/Brion), provide partial support for SONATA. Currently Brion provides a low level API to read circuit and simulation JSON configurations, spike and multi-compartment simulation outputs, SWC morphologies and query nodes in HDF5 files. It also provides a single threaded writer for multi-compartment simulation output reports. Brain provides a higher level API that makes it easier to work with full networks. All this functionality is also available in Python through the associated Python wrapping module.

libSONATA

Blue Brain's libSONATA (https://github.com/BlueBrain/libsonata) is a library that provides support to read SONATA files. The library offers an API for both Python and C++ applications. Currently libSONATA supports reading circuit files, including nodes and edges populations. The development of this library is in progress and support for the rest of the SONATA specification will be added in the future.

RTNeuron

Blue Brain's RTNeuron (Hernando et al., 2013) is a framework for visualizing detailed neuronal network models and simulations. As it relies on Brion/Brain for data access, it currently provides basic support to visualize SONATA circuits and simulations. For instance, **Fig. 5B** illustrates the RTNeuron visualization of a model of 300 biophysically detailed neurons, provided as an example in the SONATA specification GitHub repository (https://github.com/AllenInstitute/sonata/tree/master/examples/300_cells). Here, one can see neuronal morphologies and the distribution of membrane voltage across the electrical compartments comprising these morphologies as the simulation evolves over time.

pyNeuroML

NeuroML (Cannon et al., 2014; Gleeson et al., 2010) is a standardized format based on XML for declaratively describing models of neurons and networks in computational neuroscience. Cellular models which can be described range from simple point neurons (e.g. leaky integrate and fire) to multicompartmental neuron models with multiple active conductances. Networks of these cells can be specified, detailing the 3D positions or populations, connectivity between them and stimulus applied to drive the activity.

Multiple libraries have been created to support user adoption of the NeuroML language, including jNeuroML (https://github.com/NeuroML/jNeuroML) in the Java language and pyNeuroML (https://github.com/NeuroML/pyNeuroML) in Python. The latter package also gives access to all of the functionality of jNeuroML (including the ability to convert NeuroML to simulator specific code, e.g. for NEURON) through Python scripts, by bundling a binary copy of the library. PyNeuroML has recently added support for importing networks and simulations specified in the SONATA format and converting them to NeuroML. **Fig. 5E** shows a simulation of 300 integrate and fire cells in SONATA which has been imported by pyNeuroML, converted to NeuroML and executed in the NEURON simulator.

PyNN

PyNN is a simulator-agnostic Python API for describing network models of point neurons, and simulation experiments with such models (Davison et al. 2009). A reference implementation of the API for the NEURON, NEST and Brian simulators is available (http://neuralensemble.org/PyNN), and a number of other simulation tools, including neuromorphic hardware systems, have implemented the API (Brüderle et al., 2011; Rhodes et al., 2018). PyNN models can be converted to and from the NeuroML and SONATA formats with a single function call. **Fig. 5E** illustrates an example where a model in SONATA format was loaded using the PyNN “serialization” module, a simulation was carried out using the PyNN NEST backend, and simulation output was saved in the SONATA format.

NetPyNE

NetPyNE (www.netpyne.org; (Dura-Bernal et al., 2019)) is a package developed in Python and building on the NEURON simulator (Carnevale and Hines, 2006). It provides both a programmatic and graphical interfaces that facilitate the definition, parallel simulation, and analysis of data-driven multiscale models. Users can provide specifications at a high level via its standardized declarative language, e.g. a probability of connection, instead of millions of explicit cell-to-cell connections. NetPyNE supports both point neurons and biophysically-detailed multi-compartment neurons, as well as NEURON's Reaction-Diffusion (RxD) molecular-level descriptions. The tool includes built-in functions to visualize and analyze the model, including connectivity matrices, voltage traces, raster plot, local field potential (LFP) plots and information transfer measures. Additionally, it facilitates parameter exploration and optimization by automating the submission of batch parallel simulation on multicore machines and supercomputers.

NetPyNE models can be converted to and from the NeuroML and SONATA formats. As an example, we imported the 300-cell SONATA example with multicompartment cells into NetPyNE, visualized it using the NetPyNE GUI (**Fig. 5C),** and carried out a NetPyNE simulation **(Fig. 5E).**

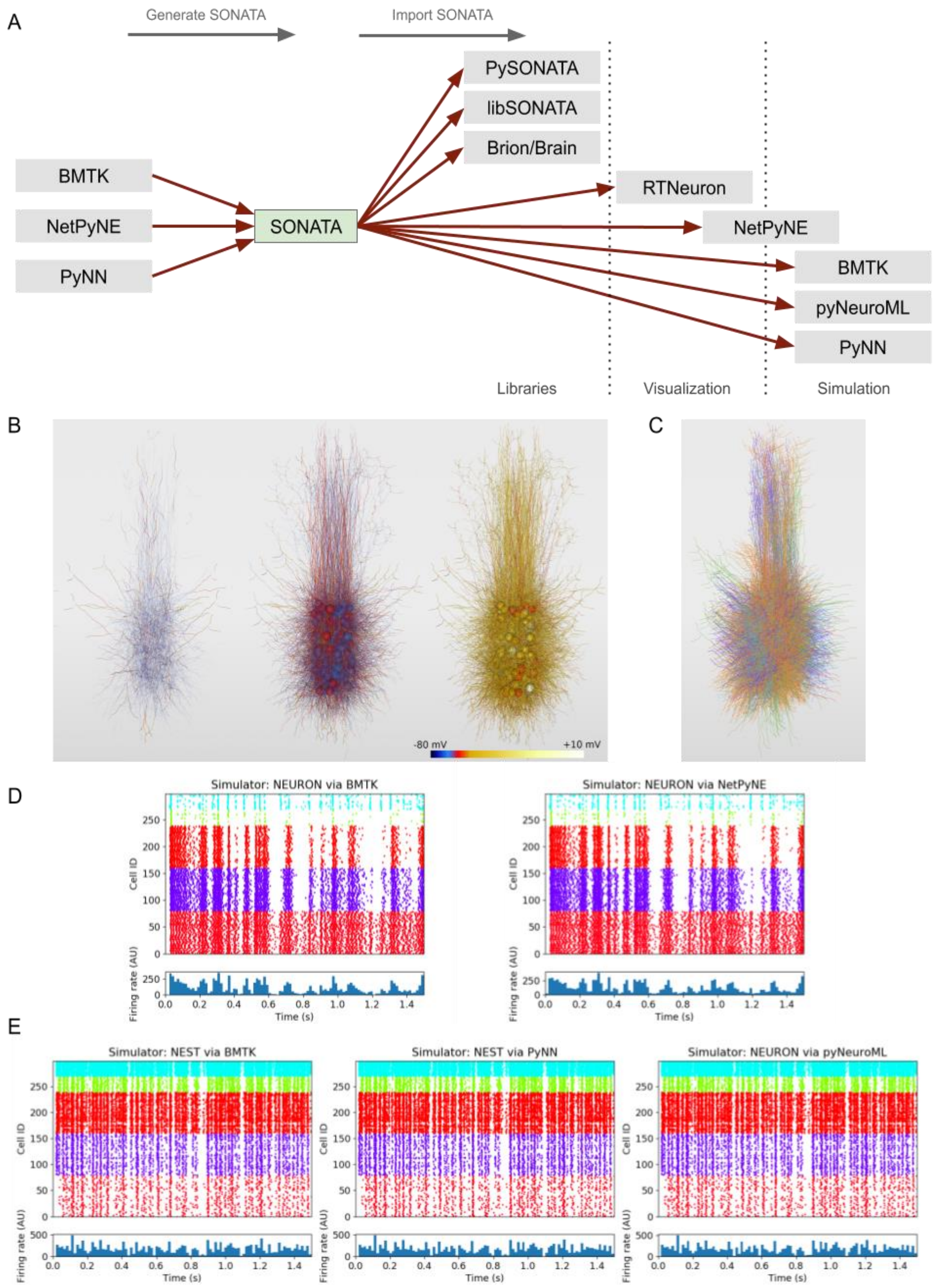


Figure 5. **Support for SONATA in simulators and libraries.** (A) Overview of applications which can generate SONATA files and the various categories of applications which can read the

generated files in, including general purpose libraries, tools allowing visualization of the networks, and packages supporting simulation. (B) RTNeuron visualization. Sample renderings at 3 simulation timesteps of an example network (generated by BMTK) with 300 biophysically detailed cells, with somatic and dendritic compartments colored according to the simulated membrane potential. (C) Rendering of the same model using the NetPyNE GUI. Each cell is colored according to which of the 5 node types it belongs. (D) The 300-cell biophysically detailed example from (B) and (C) simulated in NEURON using BMTK (left) and NetPyNE (right). (E) A network with 300 integrate and fire neurons generated by BMTK, and simulated in NEST via BMTK (left), NEST after importing the SONATA files into PyNN (middle) and NEURON after conversion of the network to NeuroML by pyNeuroML. Each raster plot in (D) and (E) is accompanied by a panel underneath showing population firing rate (arbitrary units).

Discussion

We have described SONATA, which is an open-source data format for representing neuronal circuits and simulation outputs. The development of SONATA was intended to answer the challenges of modern computational neuroscience and especially those inherent in large-scale data-driven modeling of brain networks. While very successful simulation engines such as NEURON (Carnevale and Hines, 2006), NEST (Gewaltig and Diesmann, 2007), Brian (Goodman and Brette, 2008), and others (Bower and Beeman, 1997; Ray and Bhalla, 2008) are available and are employed extensively in the field, standardization of models and simulations via data-centered file format frameworks is not widely practiced. This hinders progress of computational modeling in neuroscience and provides high barriers to non-sophisticated potential users, including many experimentalists, for entering the field, as well as contributes to the scientific reproducibility crisis (Amunts et al., 2016; Bouchard et al., 2016; Hawrylycz et al., 2016; Koch and Jones, 2016; Martin and Chun, 2016; Vogelstein et al., 2016), (Baker, 2016; Goodman et al., 2016). Several initiatives are tackling this problem, including NeuroML, PyNN, and others (Cannon et al., 2014; Gleeson et al., 2010); (Davison et al., 2009); (Ray et al., 2016). However, highly efficient and sophisticated data framework solutions are still required, given that recent increases in the amount and complexity of experimental data as well as increasing availability of high-performance computing resources are causing many research teams to develop very large-scale, highly complex and biologically realistic simulations (e.g., (Arkhipov et al., 2018; Bezaire et al., 2016; Markram et al., 2015; Schmidt et al., 2018)).

SONATA addresses these challenges by providing a data format designed for memory and computational efficiency, as well as for working across multiple platforms, and at the same time enabling as much flexibility as possible for diverse applications. To achieve this, SONATA relies on commonly used data formats such as CSV, HDF5, and JSON, which can be used across platforms, can be read and written by many existing libraries in various programming languages, and (especially in the case of HDF5) have been proven to work efficiently in parallel computations with very large datasets. Relying on these formats, we have created a specification for a wide variety of aspects of model and simulation representations, including network descriptions, simulation configuration, and simulation output.

The flexibility of these specifications is ensured by several design criteria. First, the design leaves it up to users to decide which attributes are shared within node or edge type vs. which are unique to specific nodes or edges. Second, it allows limitless creation of user-defined attributes and maintains only a small number of reserved fields. And third, via a hierarchy of types, populations, and groups of nodes/edges, it permits specification of hybrid models that may include biophysically detailed neurons, point neurons, and many other model types, all in one network model.

The SONATA community and ecosystem include multiple groups with diverse interests and are growing due to the open-source design. Initially developed jointly by the Allen Institute and the Blue Brain Project, SONATA is now supported by tools from additional teams. As described above, tools such as BMTK (Gratiy et al., 2018), RTNeuron (Hernando et al., 2013), PyNN (Davison et al., 2009), NeuroML (Cannon et al., 2014; Gleeson et al., 2010), and NetPyNE (Dura-Bernal et al., 2019) include SONATA support. The SONATA data format and framework are reflected in the free and open-source PySONATA project hosted on the GitHub (https://github.com/AllenInstitute/sonata), which is intended as a key resource for those wishing to add support for SONATA to their applications and includes specification documentation, open-source reference application programming interfaces, and model and simulation output examples. Suggestions and specification or code extensions from the community are welcome, as SONATA is intended as a living format that can evolve together with the needs of the users and trends in the field.

As such, SONATA may be extended or modified in the future, in particular to reflect and/or include specifications that are being developed for experimental neuroscience, such as NWB:N (Ruebel et al., 2019). In turn, we invite experimentalist colleagues to explore SONATA's applicability to their circumstances, as SONATA framework provides an efficient description for a variety of network aspects, including cell properties (e.g., morphologies, positions), connectivity (e.g., synaptic connections and their weights), and neural activity (e.g., spikes and voltage traces). Mutual cross-pollination between experimental and computational formats will be beneficial for everyone and perhaps will lead to convergence in data formats that will help resolve reproducibility issues and facilitate data/model exchange and collaboration.

Methods

JSON, CSV, and HDF5

JSON

JSON (JavaScript Object Notation) is a data exchange format that is easy for both humans and machines to read and write. Being text based, JSON is platform and language independent. Data organization is based on two common structures: key-value pairs and ordered lists, which are have equivalents in almost all programming languages.

CSV

CSV stands for "*comma-separated values"* and it is a very common way of laying out tabular data in text files. CSV is not a standard *per se*; the choices that have been made for SONATA are described in the official specification. It should be noted that, although the CSV abbreviation suggests comma as a separator, CSV files can use many types of separator, and, in fact, SONATA format specifies spaces as preferred separators for CSV.

HDF5

HDF5 (Hierarchical Data Format version 5) is a technology designed for storing very large heterogeneous data collections and their metadata in a single container file. HDF5 defines a binary container file format for which the HDF Group provides an implementation in C. Bindings for several other languages exist as well. Basic concepts of HDF5 include groups, datasets and attributes. Making an analogy to filesystems, groups are similar to directories and datasets to files. The main differences between HDF5 and a general purpose filesystem are that a) a dataset is not a stream of bytes like a file, but consists of a multidimensional array with a single data type for all values and that b) groups and datasets can be annotated by means of attributes. HDF5 defines some basic data types common to most programming languages: integers, floats, strings. Data can be stored linearly (the elements of a dataset are stored in increasing order, according to their index and dimension) or in "chunks" for computational efficiency (the order in how dataset elements is interleaved according to their index and dimension. For details, see https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/).

Benchmarking

Edge file benchmarks

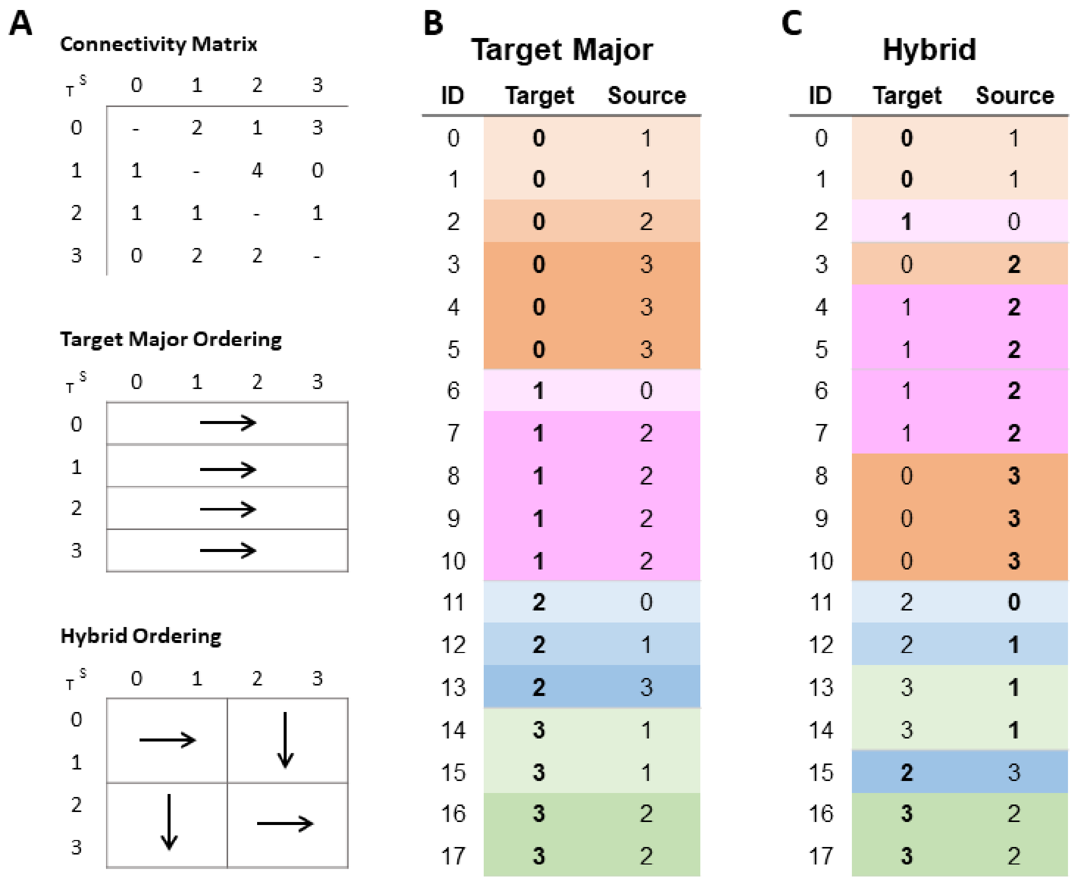
The edge file benchmarks (**Fig. 2E**) were performed on SONATA edge files representing the recurrent connections of the 45,000-neuron mouse V1 Layer 4 network model (**Fig. 4**) of mixed multicompartment and point neurons (Arkhipov et al., 2018). On average each cell receives input from 438.8 neighbors with the number and strength of synapses between any two cells being determined by source and target cell types. The network file contains over 39.2 million unique synapses partitioned into two groups, those synapses that target multicompartment neurons and those that target point points. Connections that target point neurons only require synaptic strength variable, while those that target multi-compartment neurons also require information about section number and segment distance for each synapse. The HDF5 edge file is 1.9 GB in size.

The benchmarks were conducted on an HPE SGI 8600 supercomputer. Each compute node had two Intel Xeon Gold 6140 CPUs (each with 18 cores at 2.30 GHz) and 768 GB of DRAM. Nodes were connected through a Mellanox Infiniband (IB) EDR fabric to two GS14K storage racks with a total storage capacity of 4 PB. The computing system was running Linux 3.10.0 and the filesystem was GPFS 4.2.3-6, configured with 4 MiB block size. The storage system did not have dedicated metadata drivers. The software components used and their versions are the following: glibc 2.25-49, gcc 6.4, boost 1.58, HDF5 1.10.1, Python 2.7, numpy 1.13.3 and MPI 2.16 provided by HPE.

For reference, the maximum average read bandwidth obtained in pure I/O benchmark experiments with IOR (https://ior.readthedocs.io/en/latest/) in this machine is 5.6 GiB/s using 1 single core accessing a 1 GiB file in 4 MiB blocks. The maximum average write bandwidth measured is 9.5 GiB/s using 8 cores from 1 node writing 1 GiB per core in 4 MiB operations to a shared file. POSIX I/O was used to obtain both measurements.

Two orderings of edge files were used, as illustrated in **Fig. 6A**. In target major ordering (**Fig. 6B**), data is sorted according to the ID of the target neuron, in increasing order. In hybrid ordering (**Fig. 6C**), the connectivity matrix is divided in blocks, and edges inside each block are enumerated, alternating (from block to block) between source-major and target-major orderings.

Note that SONATA supports arbitrary ordering of edges, and the two variants tested in the benchmarks are only for demonstration purposes. It is up to users to select their preferred ordering for write and read operations. As mentioned in the main text, high-performance applications may require that one creates several versions of edge files with different orderings, each for a specific type of operations that may benefit from that ordering.


A
Connectivity Matrix
Target Major Ordering
Hybrid Ordering
B
Target Major
C
Hybrid

Connectivity Matrix

T \ S	0	1	2	3
0	-	2	1	3
1	1	-	4	0
2	1	1	-	1
3	0	2	2	-

Target Major

ID	Target	Source
0	0	1
1	0	1
2	0	2
3	0	3
4	0	3
5	0	3
6	1	0
7	1	2
8	1	2
9	1	2
10	1	2
11	2	0
12	2	1
13	2	3
14	3	1
15	3	1
16	3	2
17	3	2

Hybrid

ID	Target	Source
0	0	1
1	0	1
2	1	0
3	0	2
4	1	2
5	1	2
6	1	2
7	1	2
8	0	3
9	0	3
10	0	3
11	2	0
12	2	1
13	3	1
14	3	1
15	2	3
16	3	2
17	3	2

Figure 6. Target major and hybrid ordering of edges. (A) A simple example of connectivity matrix (the number within each matrix element indicates the number of edges -- i.e., synapses -- between the two nodes) and schematics of target major and hybrid orderings. (B) and (C) Edge lists representing edges from the connectivity matrix in (A), sorted according to target major (B) or hybrid (C) ordering.

Simulation output benchmarks

The simulation output benchmarks (**Fig. 3**) were run on the aforementioned HPE SGI 8600 system. Since most simulators can run in parallel (multi-thread and/or multi-process), the benchmarking of the report generation was also done in parallel, on 16 nodes and 36 processes per node (using 1 core per process). All processes were periodically dumping data to a single, shared HDF5 file in the SONATA format. At each write operation, each process was writing several columns at its designated frame/trace region. The amount of data written at each operation is presented as the “Write block size per process” illustrated in the performance plots (the write block size applies for each process and for each write operation).

Write benchmarks made use of the Neuromapp library (https://github.com/BlueBrain/neuromapp, revision f03d3ea) (Ewart et al., 2017), which uses parallel HDF5 and MPI underneath. Read benchmarks were implemented using the Python binding of Brion/Brain (revision c16a694), the testing and plotting code can be found in the SONATA github repository in the benchmarks branch.

Loading of simulation data

Benchmarks for loading simulation data (**Fig. 4C**) were obtained for the full simulation of the 45,000-neuron recurrently connected model of Layer 4 of mouse V1 (Arkhipov et al., 2018). Figure **Fig. 4C** shows the amount of time required to parse through the SONATA network files and instantiate the in-memory cell and synaptic objects to run a full NEURON (Carnevale and Hines, 2006) simulation. Each simulation was instantiated with a computing cluster of Intel Xeon E5 processors (each core either 2.1 or 2.2 GHz), using a minimum of 5 cores and a maximum of 390 cores. The network was built using the Brain Modeling Toolkit with Python 3.6 and NEURON 7.5 with Python bindings.

Acknowledgements

This project/research has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 785907 (Human Brain Project SGA2) and from the EPFL Blue Brain Project (funded by the Swiss government's ETH Board of the Swiss Federal Institutes of Technology). SD-B was funded by NIH grants U01EB017695, R01EB022903 and 2R01DC012947-06A1, and New York State grant DOH01-C32250GG-3450000. PG was supported by the Wellcome Trust (212941, 101445). This work benefited from interactions between the authors as part of the International Neuroinformatics Coordinating Facility (INCF) Special Interest Group on Standardised Representations of Network Structures. We are grateful to Christof Koch for inspiring and supporting this work. We wish to thank the Allen Institute founder, Paul G. Allen, for his vision, encouragement, and support.

References

Amunts, K., Ebell, C., Muller, J., Telefont, M., Knoll, A., and Lippert, T. (2016). The Human Brain Project: Creating a European Research Infrastructure to Decode the Human Brain. Neuron *92*, 574–581.

Arkhipov, A., Gouwens, N.W., Billeh, Y.N., Gratiy, S., Iyer, R., Wei, Z., Xu, Z., Abbasi-Asl, R., Berg, J., Buice, M., et al. (2018). Visual physiology of the layer 4 cortical circuit in silico. PLoS Comput. Biol. *14*, e1006535.

Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. Nature *533*, 452–454.

Bezaire, M.J., Raikov, I., Burk, K., Vyas, D., and Soltesz, I. (2016). Interneuronal mechanisms of hippocampal theta oscillation in a full-scale model of the rodent CA1 circuit. Elife *5*, e18566.

Bouchard, K.E., Aimone, J.B., Chun, M., Dean, T., Denker, M., Diesmann, M., Donofrio, D.D., Frank, L.M., Kasthuri, N., Koch, C., et al. (2016). High-Performance Computing in Neuroscience for Data-Driven Discovery, Integration, and Dissemination. Neuron *92*, 628–631.

Bower, J.M., and Beeman, D. (1997). The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System (Springer, New York).

Brüderle, D., Petrovici, M.A., Vogginger, B., Ehrlich, M., Pfeil, T., Millner, S., Grübl, A., Wendt, K., Müller, E., Schwartz, M.-O., et al. (2011). A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. Biol. Cybern. *104*, 263–296.

Cain, N., Iyer, R., Koch, C., and Mihalas, S. (2016). The Computational Properties of a Simplified Cortical Column Model. PLoS Comput. Biol. *12*, e1005045.

Cannon, R.C., Turner, D.A., Pyapali, G.K., and Wheal, H.V. (1998). An on-line archive of reconstructed hippocampal neurons. J. Neurosci. Methods *84*, 49–54.

Cannon, R.C., Gleeson, P., Crook, S., Ganapathy, G., Marin, B., Piasini, E., and Silver, R.A. (2014). LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. Front. Neuroinform. *8*, 79.

Carnevale, N.T., and Hines, M.L. (2006). The NEURON Book (Cambridge University Press).

Davison, A.P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). PyNN: A Common Interface for Neuronal Network Simulators. Front. Neuroinform. *2*, 11.

Dura-Bernal, S., Suter, B.A., Gleeson, P., Cantarelli, M., Quintana, A., Rodriguez, F., Kedziora, D.J., Chadderdon, G.L., Kerr, C.C., Neymotin, S.A., et al. (2019). NetPyNE, a tool for data-driven multiscale modeling of brain circuits. eLife *8*, e44494.

Ewart, T., Planas, J., Cremonesi, F., Langen, K., Schürmann, F., and Delalondre, F. (2017). Neuromapp: A Mini-application Framework to Improve Neural Simulators. Lecture Notes in Computer Science 181–198.

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). Scholarpedia J. *2*, 1430.

Gleeson, P., Steuber, V., and Silver, R.A. (2007). neuroConstruct: a tool for modeling networks of neurons in 3D space. Neuron *54*, 219–235.

Gleeson, P., Crook, S., Cannon, R.C., Hines, M.L., Billings, G.O., Farinella, M., Morse, T.M., Davison, A.P., Ray, S., Bhalla, U.S., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. PLoS Comput. Biol. *6*, e1000815.

Gleeson, P., Cantarelli, M., Marin, B., Quintana, A., Earnshaw, M., Piasini, E., Birgiolas, J., Cannon, R.C., Alex Cayco-Gajic, N., Crook, S., et al. (2018). Open Source Brain: a collaborative resource for visualizing, analyzing, simulating and developing standardized models of neurons and circuits. bioRxiv 229484.

Goodman, D., and Brette, R. (2008). Brian: A simulator for spiking neural networks in Python. Front. Neuroinform. *2*, 5.

Goodman, S.N., Fanelli, D., and Ioannidis, J.P.A. (2016). What does research reproducibility mean? Sci. Transl. Med. *8*, 341ps12.

Gorgolewski, K.J., Auer, T., Calhoun, V.D., Craddock, R.C., Das, S., Duff, E.P., Flandin, G., Ghosh, S.S., Glatard, T., Halchenko, Y.O., et al. (2016). The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. Sci Data *3*, 160044.

Gouwens, N.W., Sorensen, S.A., Berg, J., Lee, C., and Jarsky, T. (2018a). Classification of electrophysiological and morphological types in mouse visual cortex. bioRxiv.

Gouwens, N.W., Berg, J., Feng, D., Sorensen, S.A., Zeng, H., Hawrylycz, M.J., Koch, C., and Arkhipov, A. (2018b). Systematic generation of biophysically detailed models for diverse cortical neuron types. Nat. Commun. *9*, 710.

Gratiy, S.L., Billeh, Y.N., Dai, K., Mitelut, C., Feng, D., Gouwens, N.W., Cain, N., Koch, C., Anastassiou, C.A., and Arkhipov, A. (2018). BioNet: A Python interface to NEURON for modeling large-scale networks. PLoS One *13*, e0201630.

Hawrylycz, M., Anastassiou, C., Arkhipov, A., Berg, J., Buice, M., Cain, N., Gouwens, N.W., Gratiy, S., Iyer, R., Lee, J.H., et al. (2016). Inferring cortical function in the mouse visual system through large-scale systems neuroscience. Proceedings of the National Academy of Sciences *113*, 7337–7344.

Hernando, J., Biddiscombe, J., Bohara, B., Eilemann, S., and Schürmann, F. (2013). Practical Parallel Rendering of Detailed Neuron Simulations. In EGPGV, pp. 49–56.

Jiang, X., Shen, S., Cadwell, C.R., Berens, P., Sinz, F., Ecker, A.S., Patel, S., and Tolias, A.S. (2015). Principles of connectivity among morphologically defined cell types in adult neocortex. Science *350*, aac9462.

Jun, J.J., Steinmetz, N.A., Siegle, J.H., Denman, D.J., Bauza, M., Barbarits, B., Lee, A.K., Anastassiou, C.A., Andrei, A., Aydın, Ç., et al. (2017). Fully integrated silicon probes for high-density recording of neural activity. Nature *551*, 232–236.

Kasthuri, N., Hayworth, K.J., Berger, D.R., Schalek, R.L., Conchello, J.A., Knowles-Barley, S., Lee, D., Vázquez-Reina, A., Kaynig, V., Jones, T.R., et al. (2015). Saturated Reconstruction of a Volume of Neocortex. Cell *162*, 648–661.

Koch, C., and Jones, A. (2016). Big Science, Team Science, and Open Science for Neuroscience. Neuron *92*, 612–616.

Koch, C., and Reid, R.C. (2012). Neuroscience: Observatories of the mind. Nature *483*, 397–398.

Lee, W.-C.A., Bonin, V., Reed, M., Graham, B.J., Hood, G., Glattfelder, K., and Reid, R.C. (2016). Anatomy and function of an excitatory network in the visual cortex. Nature *532*, 370–374.

Markov, N.T., Ercsey-Ravasz, M.M., Ribeiro Gomes, A.R., Lamy, C., Magrou, L., Vezoli, J., Misery, P., Falchier, A., Quilodran, R., Gariel, M.A., et al. (2012). A weighted and directed interareal connectivity matrix for macaque cerebral cortex. Cereb. Cortex *24*, 17–36.

Markram, H., Muller, E., Ramaswamy, S., Reimann, M.W., Abdellah, M., Sanchez, C.A., Ailamaki, A., Alonso-Nanclares, L., Antille, N., Arsever, S., et al. (2015). Reconstruction and Simulation of Neocortical Microcircuitry. Cell *163*, 456–492.

Martin, C.L., and Chun, M. (2016). The BRAIN Initiative: Building, Strengthening, and Sustaining. Neuron *92*, 570–573.

Oh, S.W., Harris, J.A., Ng, L., Winslow, B., Cain, N., Mihalas, S., Wang, Q., Lau, C., Kuan, L., Henry, A.M., et al. (2014). A mesoscale connectome of the mouse brain. Nature *508*, 207–214.

Ray, S., and Bhalla, U.S. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. Front. Neuroinform. *2:6*.

Ray, S., Chintaluri, C., Bhalla, U.S., and Wójcik, D.K. (2016). NSDF: Neuroscience Simulation Data Format. Neuroinformatics *14*, 147–167.

Rhodes, O., Bogdan, P.A., Brenninkmeijer, C., Davidson, S., Fellows, D., Gait, A., Lester, D.R., Mikaitis, M., Plana, L.A., Rowley, A.G.D., et al. (2018). sPyNNaker: A Software Package for Running PyNN Simulations on SpiNNaker. Front. Neurosci. *12*, 816.

Ruebel, O., Tritt, A., Dichter, B., Braun, T., Cain, N., Clack, N., Davidson, T.J., Dougherty, M., Fillion-Robin, J.-C., Graddis, N., et al. (2019). NWB:N 2.0: An Accessible Data Standard for Neurophysiology. bioRxiv.

Schmidt, M., Bakker, R., Hilgetag, C.C., Diesmann, M., and van Albada, S.J. (2018). Multi-scale account of the network structure of macaque visual cortex. Brain Struct. Funct. *223*, 1409–1435.

Tasic, B., Yao, Z., Graybuck, L.T., Smith, K.A., Nguyen, T.N., Bertagnolli, D., Goldy, J., Garren, E., Economo, M.N., Viswanathan, S., et al. (2018). Shared and distinct transcriptomic cell types across neocortical areas. Nature *563*, 72–78.

Teeter, C., Iyer, R., Menon, V., Gouwens, N., Feng, D., Berg, J., Szafer, A., Cain, N., Zeng, H., Hawrylycz, M., et al. (2018). Generalized leaky integrate-and-fire models classify multiple neuron types. Nat. Commun. *9*, 709.

Vogelstein, J.T., Mensh, B., Häusser, M., Spruston, N., Evans, A.C., Kording, K., Amunts, K., Ebell, C., Muller, J., Telefont, M., et al. (2016). To the Cloud! A Grassroots Proposal to Accelerate Brain Science Discovery. Neuron *92*, 622–627.

de Vries, S.E.J., Lecoq, J., Buice, M.A., and Groblewski, P.A. (2018). A large-scale, standardized physiological survey reveals higher order coding throughout the mouse visual cortex. bioRxiv.