CxSystem2

Jan 03, 2023

Contents

CxSystem2 is a simulation framework for cortical networks, which operates on personal computers. It is implemented in Python on top of the popular Brian2 simulator, and runs on Linux, Windows and MacOS. There is also a *web-based version* available via the Human Brain Project Brain Simulation Platform (BSP).

CxSystem2 embraces the main goal of Brian – minimizing development time – by providing the user with a simplified interface. While many simple models can be written in pure Brian code, more complex models can get hard to manage due to the large number of biological details.

We currently provide two interfaces for constructing networks: a browser-based interface (locally or via the BSP), and a file-based interface (json or csv). Before incorporating neuron models into a network, the user can explore their behavior using the *Neurodynlib* submodule. Spike output and 3D structure of network simulations can be visualized using ViSimpl, a visualization tool developed by the GMRV Lab.

More information on the technical details and our motivation is available here. We have used the simulator to construct a simplified version of a comprehensive cortical microcircuit.

The preliminary version of CxSystem2 was developed at the Aalto University in 2012-2015, and the current version was developed at the HUS Helsinki University Hospital and University of Helsinki in 2013-2019. The software is distributed under the terms of the GNU GPL v3.

Contents:

CHAPTER 1

Introduction

1.1 Using the BSP interface

CxSystem2 can be used on the Human Brain Project Brain Simulation Platform (BSP), which is a collaborative web platform designed for reconstruction and simulation of brain models. To use CxSystem2 on the BSP, you only need an account for the BSP and a compatible web browser (Firefox or Chrome). This link provides information on acquiring an account to the BSP.

After you have used CxSystem2 on the BSP and want to use it for more complex simulations, we recommend that you *install it locally*.

Access the BSP collaboratory.

1.2 Installing locally

CxSystem2 can be installed locally using *pip*. The local version allows running simulations via a browser user interface (BUI) or from command line. Simulations can also be sent via ssh to clusters that use the SLURM workload manager.

To install CxSystem2, please follow the instructions on our GitHub page.

1.3 Neurodynlib

Neurodynlib is a submodule that contains all the point neuron models included in CxSystem2. It can be used independently of CxSystem2, for example, inside a Jupyter notebook to explore the behaviour of single neuron models or small networks. Neurodynlib grew out of the need to understand the behaviour of neuron models before incorporating them into a larger network.

Neurodynlib is based on the exercise code repository accompanying the book Neuronal Dynamics.

The book is also published in print: Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

We provide a *short tutorial* on Neurodynlib as Jupyter Notebooks.

1.3.1 Available neuron models

The currently supported point neuron models are:

`LIF`: Leaky integrate-and-fire.

`EIF`: Exponential integrate-and-fire.

`ADEX`: Adaptive exponential integrate-and-fire.

`IZHIKEVICH`: Izhikevich model.

`LIFASC`: Leaky integrate-and-fire with after-spike currents.

In addition, there is a multicompartmental neuron type (PC, pyramidal cell) that follows exponential integrate-and-fire dynamics.

1.3.2 Available receptor models

The currently supported receptor models are:

`SIMPLE_E`: Excitatory conductance with exponential decay.

`SIMPLE_I`: Inhibitory conductance with exponential decay.

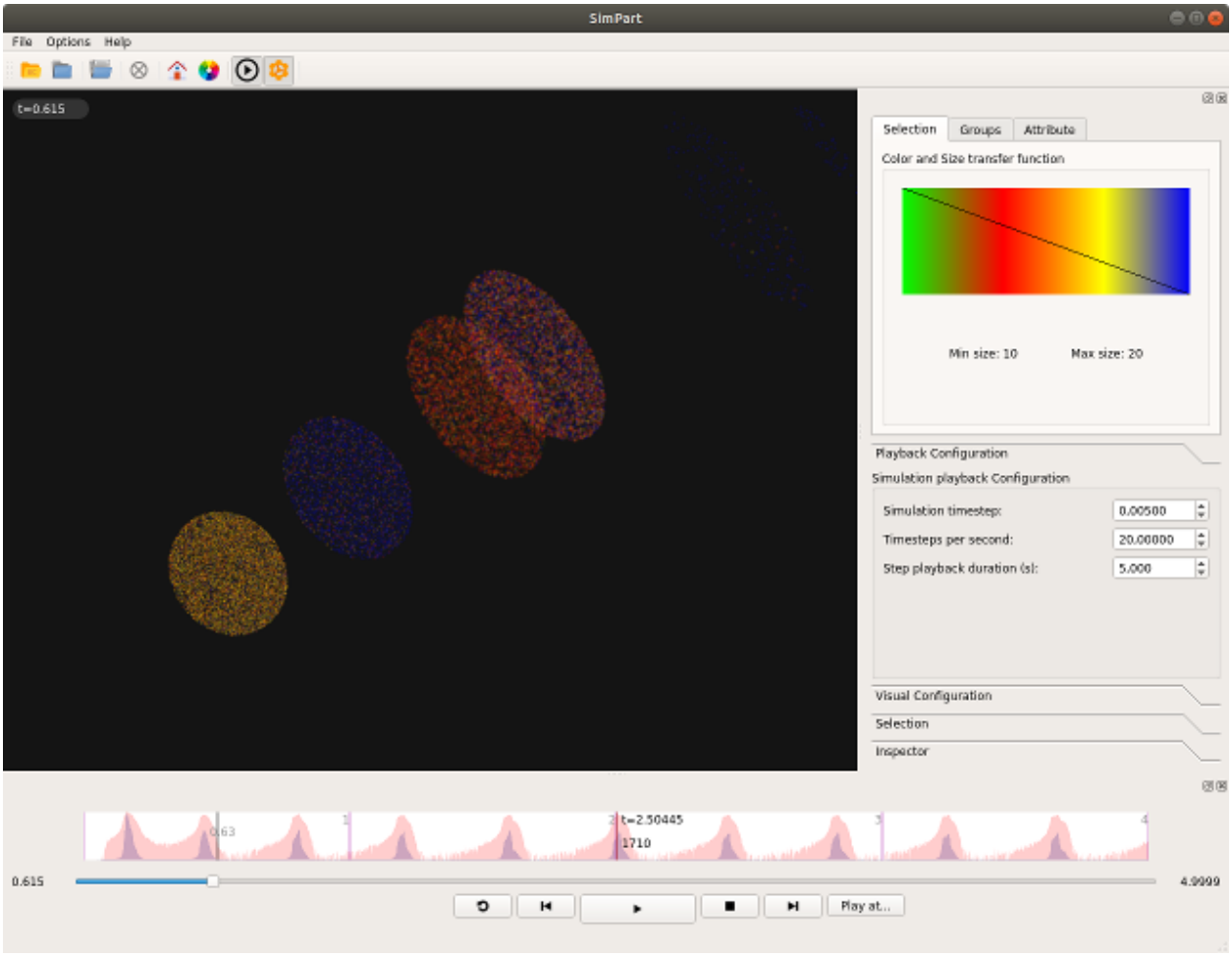
`SIMPLE_E_NMDA`: AMPA and NMDA receptors.

`SIMPLE_I_GABAB`: GABA-A and GABA-B receptors.

`E_ALPHA`: Excitatory alpha synapse.

`I_ALPHA`: Inhibitory alpha synapse.

1.4 Visualization in ViSimpl



Simulation results from CxSystem2 and the 3D structure of the model can be visualized using ViSimpl, a 3D particle-based rendering tool developed by the GMRV Lab. Currently only spike data can be visualized using this tool.

More information on ViSimpl and binaries for download (currently only for Linux and MacOS) are available here. Note that the .AppImage binary is for Linux and the .dmg binary is for macOS.

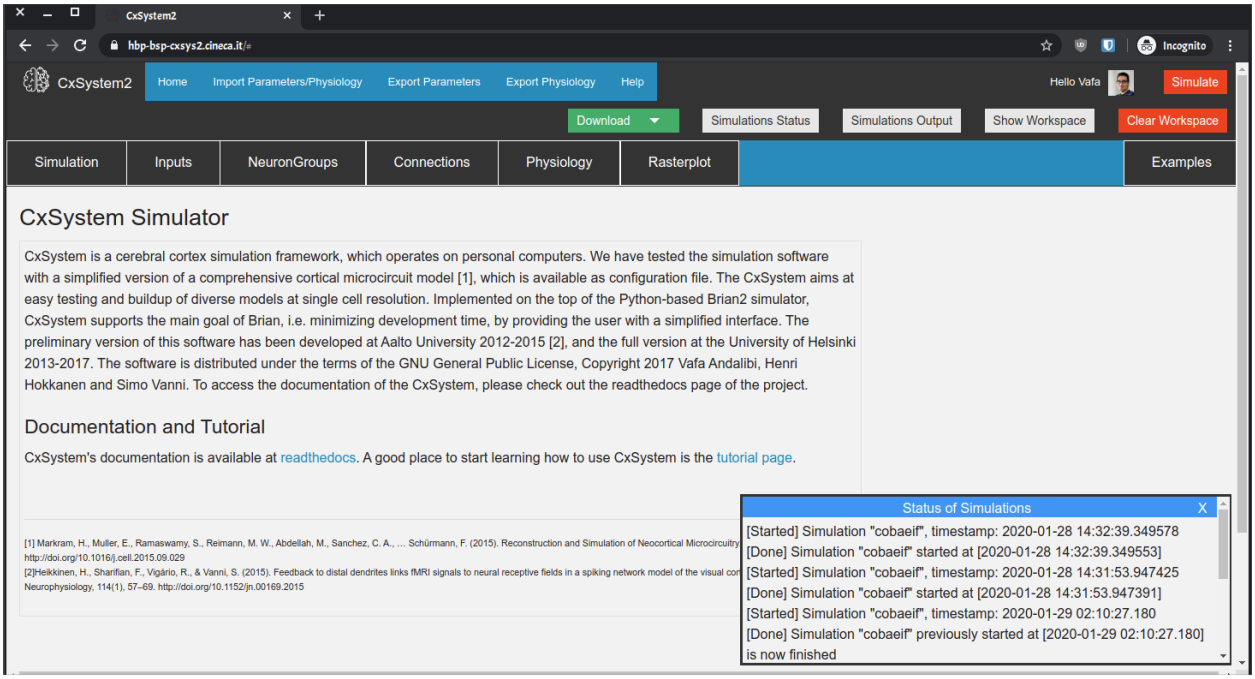
CHAPTER 2

Tutorials

TODO: Running an example, Visualizing results, Constructing a new model, Using neurodynlib

2.1 1 - Running an example simulation

This is the main window of CxSystem2 you should see after opening the application in *the Brain Simulation Platform* (the window for the local browser user interface is similar):

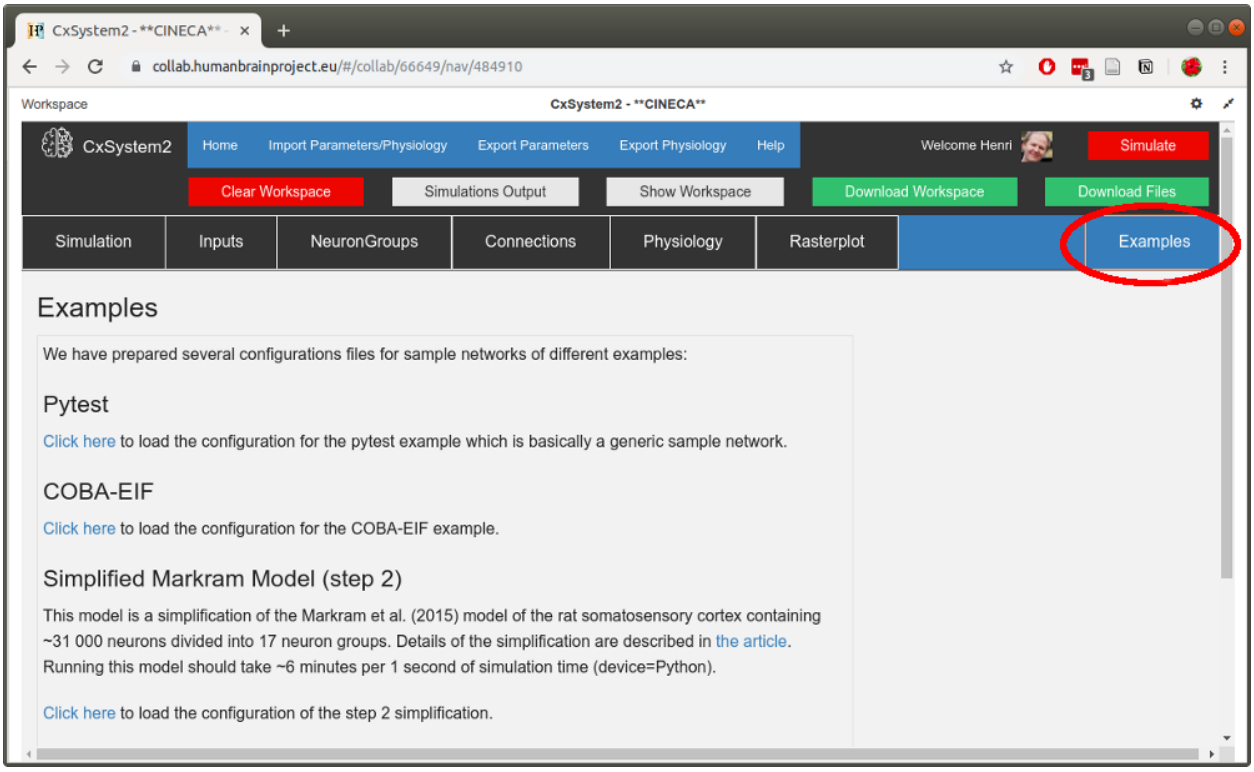


The top panel consists of three rows of buttons. The topmost one has the *Home* and *Help* buttons as well as buttons for exporting and importing configuration files. The row in the middle has buttons for viewing console output and for

managing files. The last row is mainly for building the network and configuring the simulation. The *Simulate* in the top-right corner launches the simulation(s).

On lower right part of the browser, you can see a small windows entitled "Status of Simulations". As the name implies, this window will print out information about the simulation you've submitted the request for. You can temporarily close this window and open it later on by clicking on the "Simulation Status" button on the top menu.

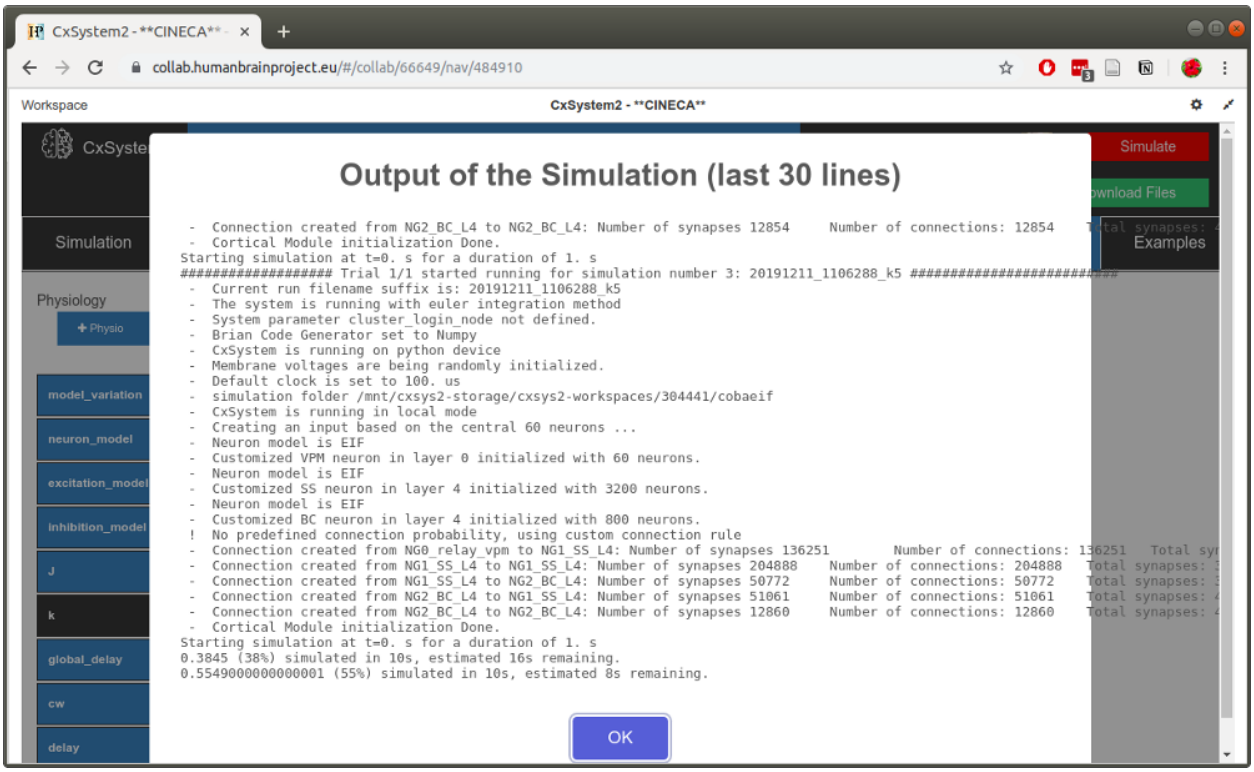
Now, to run a simple example, let's open up the *Examples* tab:



Under the *Examples* tab you can find networks of different complexity, the simplest ones being on the top. We plan to provide more examples and our scientific contributions here in the future (you're also free to contribute).

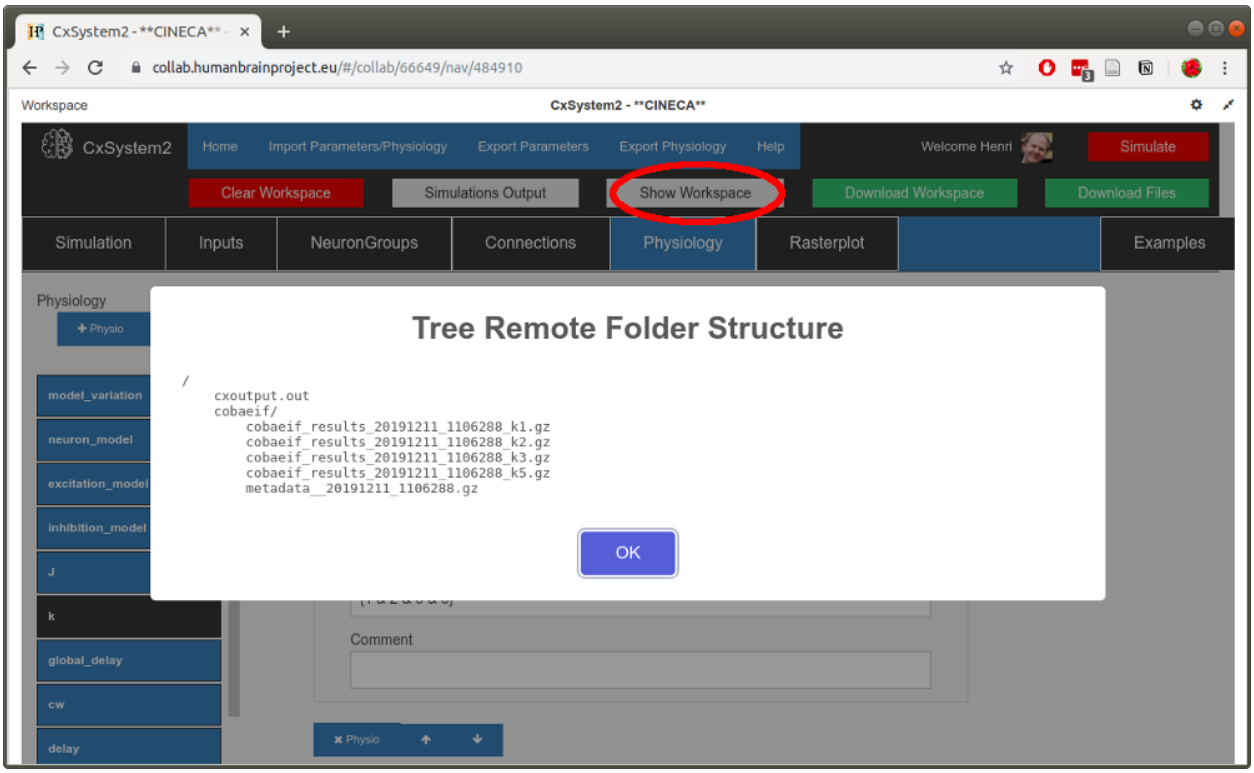
Now, pick the "COBA-EIF" example. After you have loaded the example, hit the Simulate button without changing anything in the configurations. After a few seconds you should get a window saying that the simulation has been sent to the server.

To see how your simulation is progressing, click on the *Simulations Output* button:



This shows you the console output and it allows you to follow the progress of your simulations. (If you are running CxSystem2 locally, you can see the output in the terminal window where the server is running.)

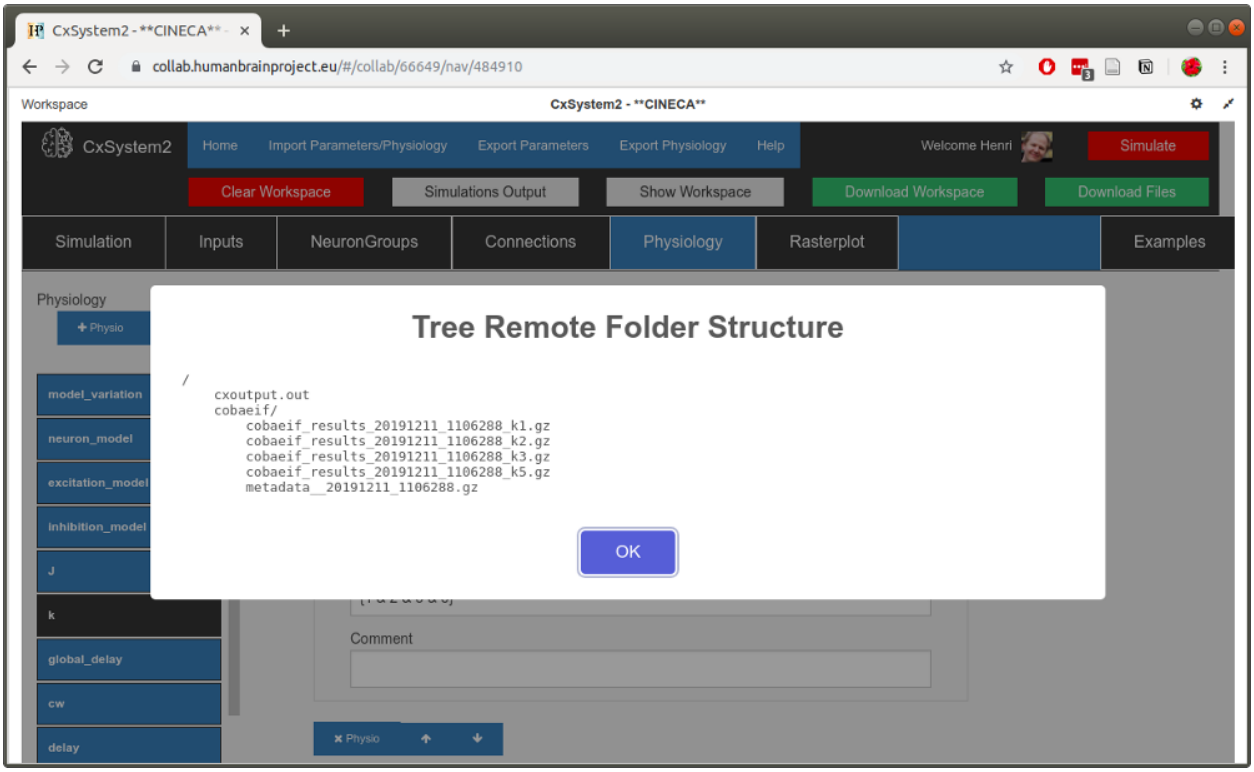
After a few minutes, all the simulations should be ready. You should be able to see the resulting data files by clicking the *Show Workspace* button:



Hooray! You have run your first simulations! See the next section for visualizing the results.

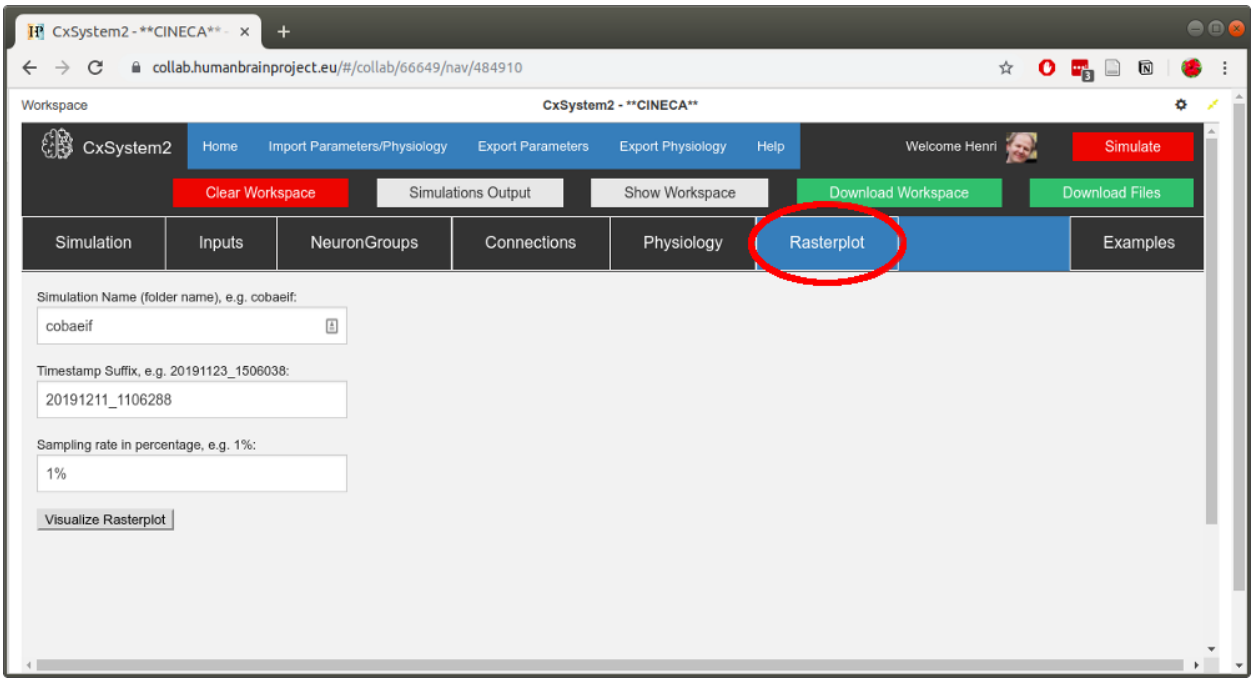
2.2 2 - Basic visualization

Let's continue by visualizing the spike data we created in Tutorial 1:



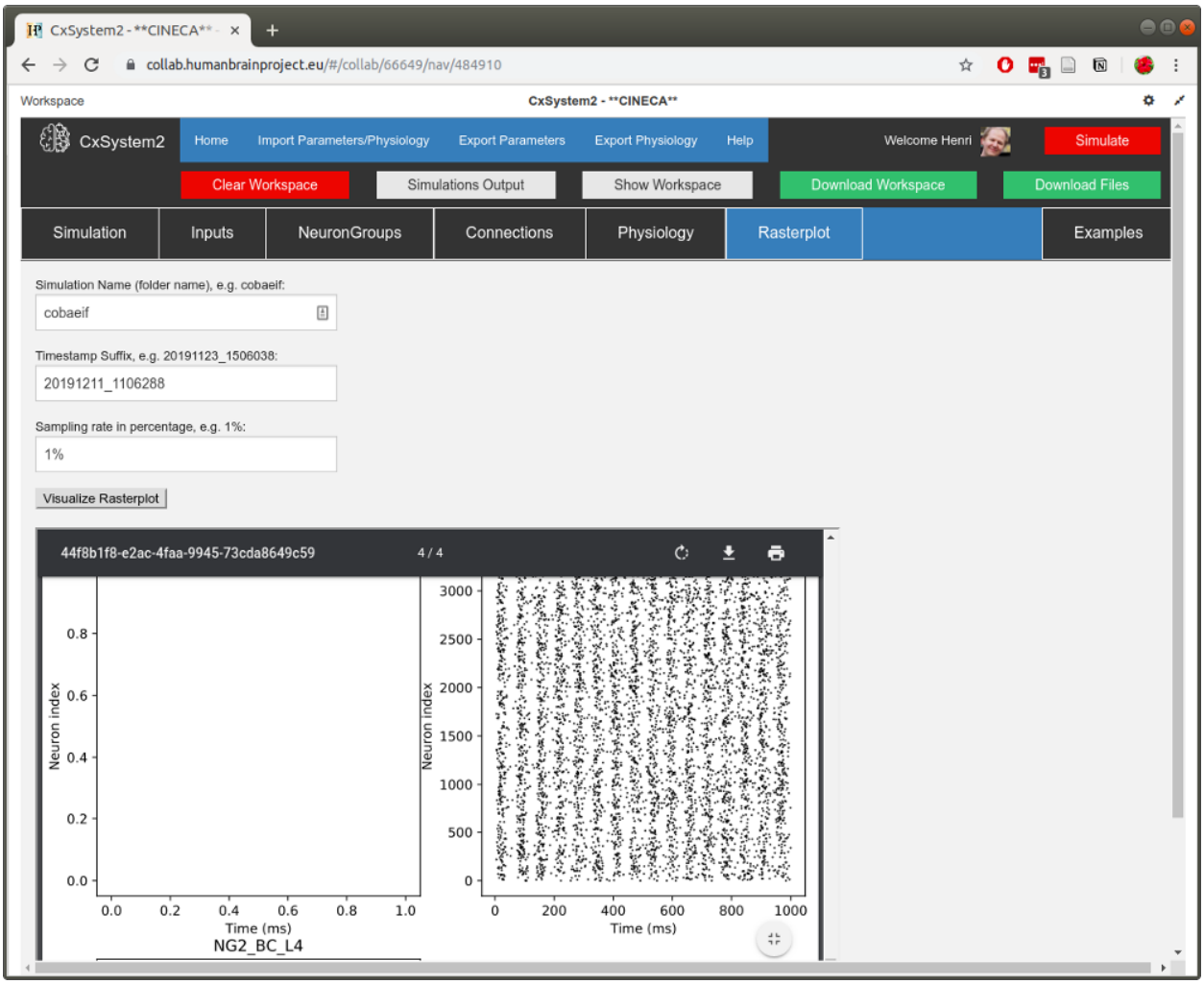
We actually did a small parameter search for the parameter *k*, which in this configuration means the relative weight of inhibitory synapses. Thus we have 4 data files containing spikes (plus the metadata file).

Now, from the workspace view (above), copy the timestamp of one of the simulation files (here 20191211_1106288). Then open up the *Rasterplot* tab:



Fill in the *simulation name* (cobaeif) to the first text field and paste the *timestamp* to the second field. Leave the *sampling rate* as 1%: the current plotting method will create pdfs that are heavy to render if some of your simulations have a lot of spikes. (This visualization tool is intended only as a simple screening tool.)

When you have filled in the fields, hit the *Visualize* button. After a few seconds, a PDF with the spike data should open up:

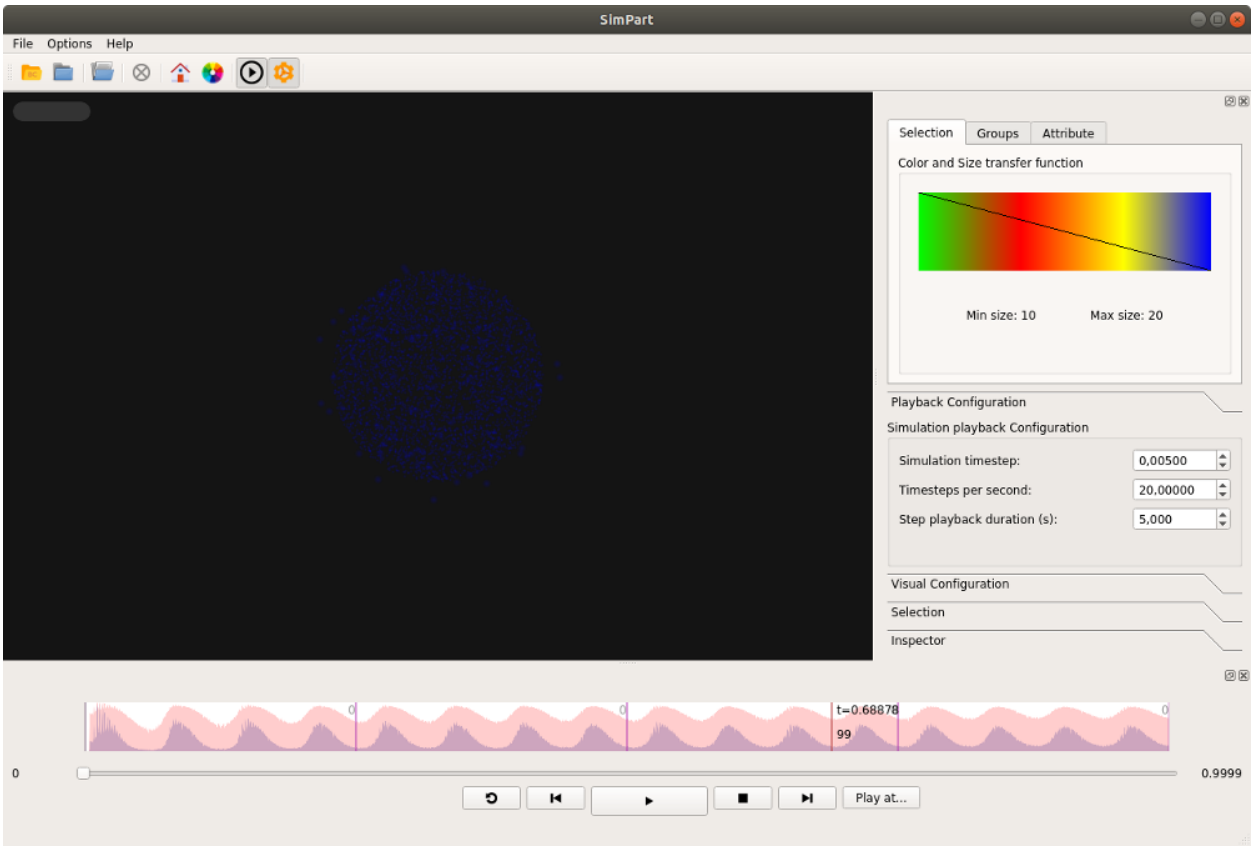


When you scroll down the pdf, you can see that there are four pages corresponding to the four simulation files. (In networks with more neuron groups to visualize, a single simulation will occupy multiple pages.) You can see that with *k=1,2,3* (not that much inhibition) there is little variation in spike dynamics, but with *k=5* the activity seems more rhythmic.

If you have CxSystem2 *installed locally*, you can download the simulation data to visualize it in *ViSimpl*. Just click on the *Download workspace* button. After the download has finished, open up a terminal window, unpack the file and use the `cxvisualize` command:

```
(CX2) henhok@taz:~: cd Downloads
(CX2) henhok@taz:~/Downloads$ tar xvfz workspace.tar.gz
304441/
304441/cxoutput.out
304441/cobaeif/
304441/cobaeif/cobaeif_results_20191211_1106288_k1.gz
304441/cobaeif/cobaeif_results_20191211_1106288_k2.gz
304441/cobaeif/cobaeif_results_20191211_1106288_k3.gz
304441/cobaeif/cobaeif_results_20191211_1106288_k5.gz
(CX2) henhok@taz:~/Downloads$ cxvisualize 304441/cobaeif/cobaeif_results_20191211_
↪1106288_k5.gz
Setting OpenGL context to 4.4
...
```

If the ViSimpl binary is available in the system path, the `cxvisualize` command should lauch ViSimpl with the simulation data (for *k=5* in this case):



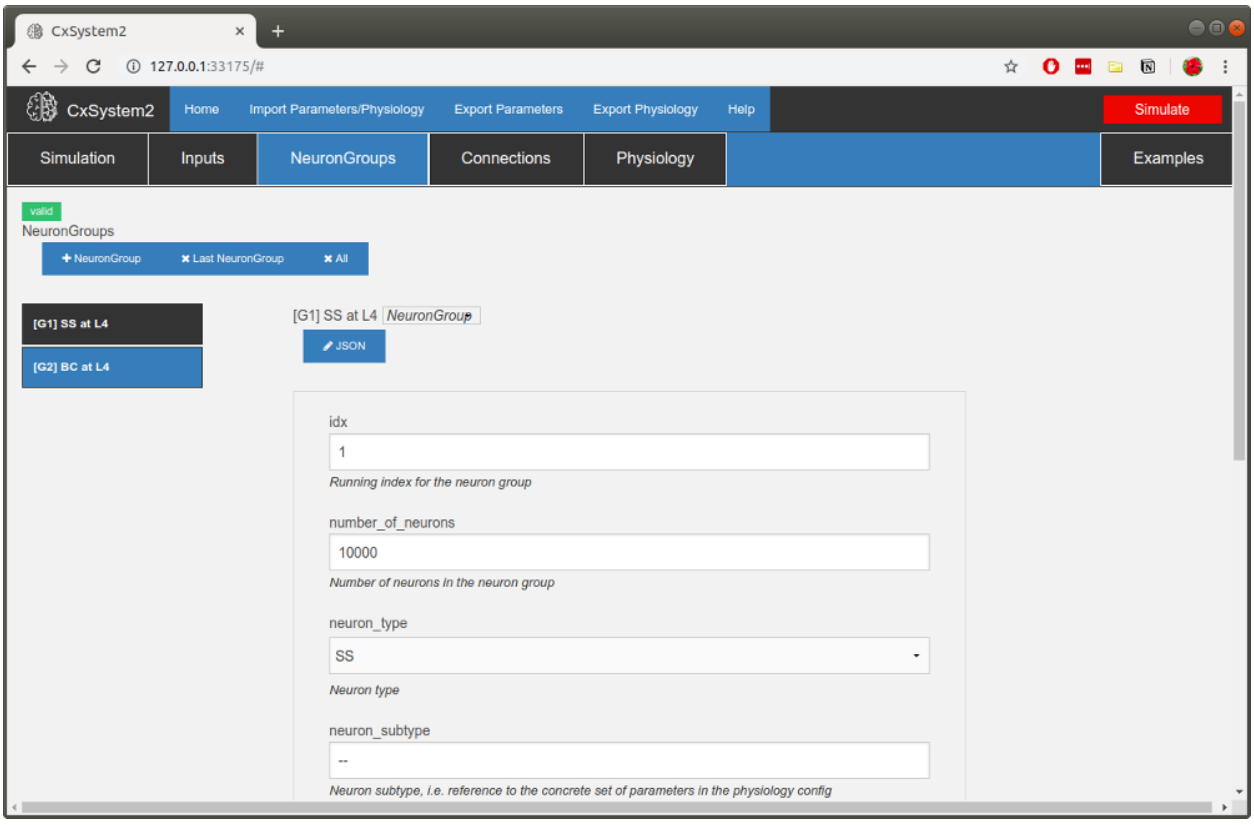
Congratulations! If you want to do your own visualizations on the spike data using Python, please see the *user's guide*.

2.3 3 - Building a new model

Let's then build a new network composed of adaptive exponential integrate-and-fire neurons. First, open the *Template network* under the *Examples* tab.

Let's first set the general simulation parameters. You don't need to worry about many of the things here for now. Just change the `simulation_title` from the template title. Also, check that `init_vms` (we want to randomize initial membrane voltages) and `multidimension_array_run` are checked (we want to do a parameter search). Run_in_cluster should be left unchecked. Finally set the `runtime` parameter (topmost) to 3*second and `device` to Cython.

Now, click the *NeuronGroups* tab. In the template config, there is one excitatory and one inhibitory population, and this is the same setup we will be using here. Let's however increase the neuron counts a bit. Pick the spiny stellate (SS) group and change the number of neurons to 10000:

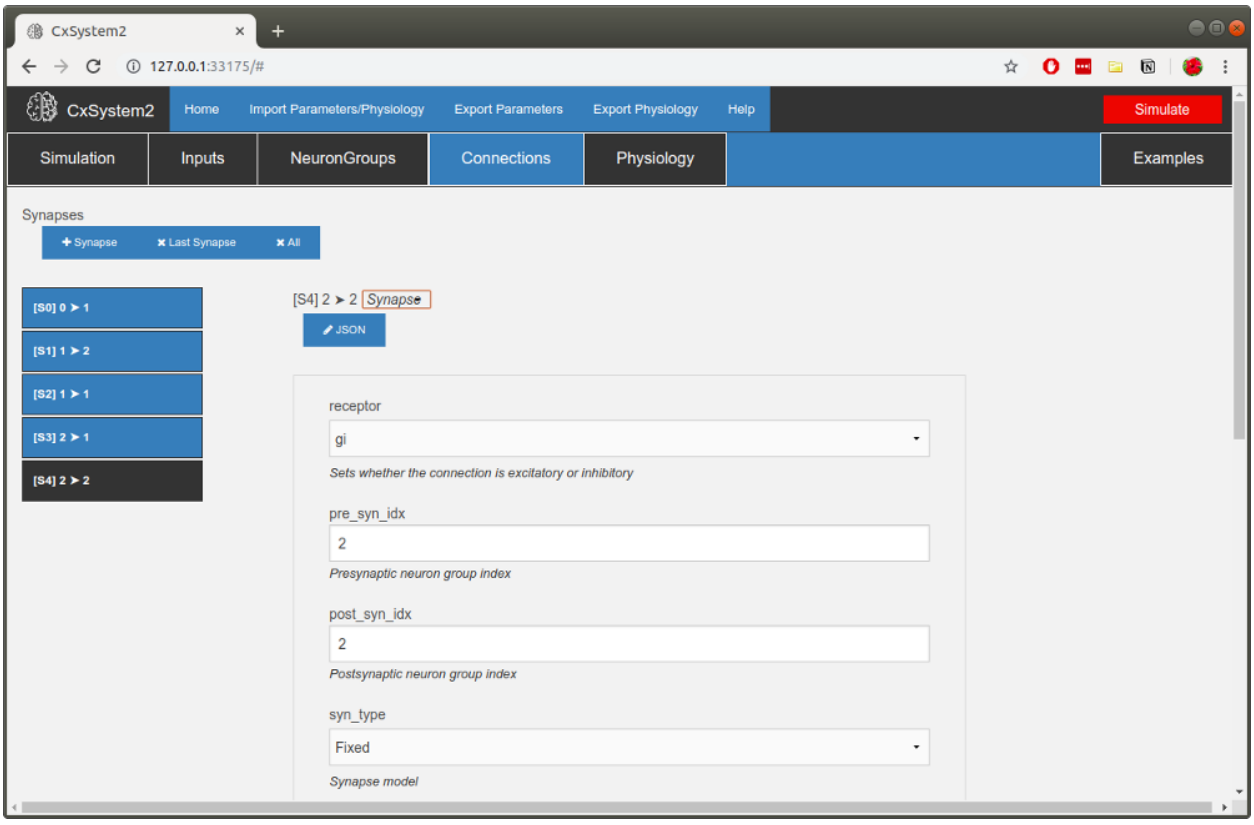


Also change the number of background inputs to 1000. For the basket cell (BC) group, change the number of neurons to 2500 and the number of background inputs to 1000.

Now, let's add some connections. We need to remember here that neuron index 1 corresponds to the excitatory group and neuron index 2 correspond to the inhibitory group. Let's leave the first connection `S0` as it is (this is a connection from the input group). Now open up `S1` and from there change the number of synapses per connection (parameter `n`) to 1. Let's leave the connection probability (parameter `p`) at 10% (0.1).

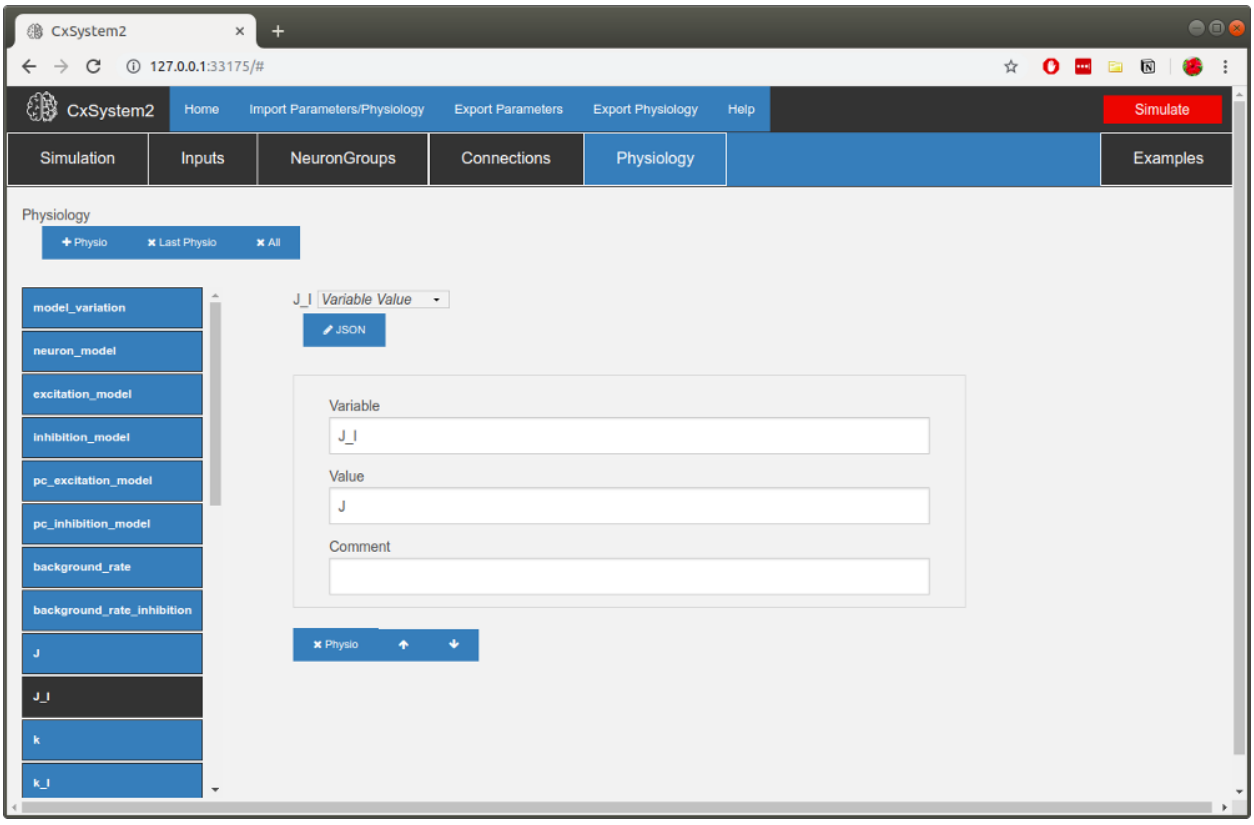
Let's add the rest of the connections. Click the *+Synapse* button. Then set the parameters for `S2`: we want this to be an excitatory connection (set `receptor` to `ge`) from the excitatory group to itself (set `pre_syn_idx` to 1 and `post_syn_idx` to 1). Set `p` to 0.1 and `n` to 1 again. The rest of the parameters can be left as they are by default.

Now add the inhibitory connections. Click the *+Synapse* button again. Set the parameters for `S3`: we want this to be an inhibitory connection (set `receptor` to `gi`) from the inhibitory group to the excitatory group (set `pre_syn_idx` to 2 and `post_syn_idx` to 1). Set `p` to 0.1 and `n` to 1 again. Finally, add an inhibitory self-connection with the same `p` and `n`. You should have a collection of synapses like this:



We're almost there... but we still need to set parameters under the *Physiology* tab. Because we want to simulate AdEx neurons you need to change the `neuron_model` parameter to `'ADEX'`. (Note that you need to have single- or double-quotes around the model name – otherwise CxSystem will think you are referring to a constant in the configuration file.) Check that `excitation_model` is `'SIMPLE_E'` and `inhibition_model` is `'SIMPLE_I'` (exponentially decaying conductance for excitatory and inhibitory receptors). There are some parameters that are not used in this simulation (like `pc_excitation_model` which is for pyramidal cells), but you don't need to delete these parameters.

A common scheme to set connection weights is to set weights to excitatory connections and then specify the inhibitory connections with respect to them. Thus, in this template, there are global variables for each of these: `J` (E-to-E connections), `J_I` (E-to-I connections), `k` (ratio of I-E connection weight with respect to E-to-E connection weight) and `k_I` (ratio of I-to-I connection weight with respect to E-to-I connection weight). We will only be needing one excitatory weight and one inhibitory weight. Set `J` to 0.5*nS. Then set `J_I` to J and `k_I` to k:



Can you find where the time constants for the receptors are? They are under `receptor_taus` (the letter tau often signifies a time constant). Change `tau_e` to 5*ms and `tau_i` to 10*ms.

Now we still need to change the neuron model parameters. Go to the bottom of the list of parameters and open up the *BC* dictionary. Inside you'll find parameters for the BC neuron, which in the template are EIF neurons. We need to add some parameters to parametrize AdEx neurons. First, however, change the existing parameters: `C` to 59*pF, `gL` to 2.9*nS, `EL` to -62*mV, `VT` to -42*mV, `DeltaT` to 3*mV and `V_res` to -54*mV. Then scroll down to the bottom and click the *+row* button three times. Then in the new empty fields write down the new parameter values: `a` = 1.8*nS, `b` = 61*pA and `tau_w` = 16*ms. Finally, change the `V_init_min` parameter to EL-5*mV and `V_init_max` to EL+5*mV.

Once you have done all the changes to the BC neuron group, do the same changes to the SS group.

Now we should have a working simulation config. But what's the parameter search we wanted to do? We want to see how changing background input rate and changing the ratio of inhibition-to-excitation changes changes network behavior. There are two notations for parameter search in CxSystem. The first notation is `{start|end|step}` which has a similar behavior as `numpy.arange(start,end,step)`. For example, setting a parameter to `{0|1|0.2}` would create an array of following values: `0.0, 0.2, 0.4, 0.6, 0.8`. The next notation is `{value1 & value2 & value3 & ... }` where the user can add multiple desired values manually. For example, setting a parameter to `{ 0 & 1 & 100 & 9 }` will create an array of following values: `0, 1 , 100, 9`. Note that these values can be used at the same time. As an example, consider we want to add search ranges to the parameters `background_rate` and `k`. Here we use the curly brace syntax: set `background_rate` as {0|5|2}*Hz (corresponding to 0, 2, 4 Hz) and `k` as {1 & 3 & 5 & 7}. Now you have defined a 3x4 parameter search of following values:

background_rate	0	0	0	0	2	2	2	2	4	4	4	4
k	1	3	5	7	1	3	5	7	1	3	5	7

Finally you can hit the *Simulate* button. When you simulate this on a single CPU core (`number_of_process` is

1) it should take 30 min - 1 hour to finish. Once the simulation is finished you should be able to see 12 files in your workspace:

```
my_first_model_results_20191216_0901136_background_rate0H_k1.gz
my_first_model_results_20191216_0901136_background_rate0H_k3.gz
my_first_model_results_20191216_0901136_background_rate0H_k5.gz
my_first_model_results_20191216_0901136_background_rate0H_k7.gz
my_first_model_results_20191216_0901136_background_rate2H_k1.gz
my_first_model_results_20191216_0901136_background_rate2H_k3.gz
my_first_model_results_20191216_0901136_background_rate2H_k5.gz
my_first_model_results_20191216_0901136_background_rate2H_k7.gz
my_first_model_results_20191216_0901136_background_rate4H_k1.gz
my_first_model_results_20191216_0901136_background_rate4H_k3.gz
my_first_model_results_20191216_0901136_background_rate4H_k5.gz
my_first_model_results_20191216_0901136_background_rate4H_k7.gz
```

Now you can use the techniques described in Tutorial 3 to visualize the results!

2.4 4 - Using neurodynlib

Neurodynlib tutorials are composed as Jupyter Notebooks. You can find the notebooks in the Brain Simulation Platform. If you want to play around with the models, copy the notebooks to your own collab.

If you have installed CxSystem2 locally, the tutorial notebooks are also available in the `cxsystem2.neurodynlib` submodule under the folder *tutorial*. In the terminal, write

```
cd /path_to_CxSystem2/cxsystem2/neurodynlib/tutorial/
jupyter notebook
```

If you don't have jupyter, you can install it by typing `pip install jupyter`.

CHAPTER 3

User's guide

3.1 Anatomy & simulation configuration

This configuration consists of

- *Simulation run parameters*
- *External inputs*
- *Neuron groups (cell types)*
- *Connections (pathways)*

In the browser user interface these are divided into separate tabs, while in the csv file interface, they are all located in the same "anatomy configuration file". This configuration includes also *recording monitors* for state variables.

If you are using the csv file interface to construct your model, please start from the anatomy and physiology template csv files provided. If you still have trouble defining your model, please see CxSystem1 documentation or contact us.

3.1.1 Simulation run parameters

The necessary parameters for running simulations locally are:

Simulation `runtime{int*unit}`: Sets the duration of the simulation, e.g. 200*ms.

`device{Python,Cython,Cpp,GeNN}`: Sets the device for Brian2 code generation.

`sys_mode{local, expanded}`: The system can be run in two modes: **local** and **expanded** mode. Expanded mode applies distance dependence of connection probabilities.

`scale{int}`: Relative area of cortical surface.

`grid_radius{float*unit}`: Sets the radius of the 2D circle from which the x-y coordinates are randomized, e.g. 210*um.

`min_distance{float*unit}`: Sets the minimum distance between neurons inside single layer, e.g. 1*um.

`workspace_path{string}`: The main working directory of CxSystem; other paths are given relative to this path.

`simulation_title{string}`: Title for the current batch of simulations.

`compression_method{gzip,bzip2,pickle}`: Data compression method.

`import_connections_from`: Path and filename from where connections with synaptic weights are imported (relative to workspace path).

`number_of_process`: Number of CPU cores to use in parallel. For array runs only. A single simulation cannot be run in parallel.

`default_clock`: Simulation time step (e.g. 0.1*ms).

`trials_per_config{int}`: Number of trials for each set of parameters.

`init_vms{True, False}`: If True, randomize initial membrane voltages between V_init_min and V_init_max. If False, the initial membrane voltage is set as V_init.

`load_positions_only{True, False}`: Import neuron positions from connectivity file but randomize connections.

`benchmark{True, False}`: Only for development. Needs modified copy of brian2 library.

`save_input_video{True, False}`: Defines whether the generated video input is to be saved or not. This is essential in case the users wants to use the identical input on different runs, so the input can be saved by setting this to 1 and used later.

`multidimension_array_run{True, False}`: Defines whether the array run is multi-dimensional or one-dimensional. In one-dimensional array run, each set of parameters is run separately, while the other set is fixed. In multidimensional run, the full matrix of parameter combinations are run.

`profiling{True, False}` Defines whether CxSystem should report the benchmark using the built-in Brian profiler.

`run_in_cluster{True, False}` Run the simulation on a cluster.

If you want to run simulations on a cluster, you will also need to define:

Simulation `cluster_job_file_path{string}`: Absolute path and filename of the local batch file. Slurm example provided.

`cluster_number_of_nodes{int}`: Number of nodes to be employed on the HPC server.

`cluster_address{string}`: Address of the HPC server (e.g. daint.cscs.ch).

`cluster_login_node{string}`: Address of the login node in case there is one (e.g. ela.cscs.ch).

`cluster_username{string}`: Username on the HPC system.

`cluster_workspace{string}`: Workspace path on the HPC server.

3.1.2 External input

Currently, three types of inputs can be used, namely `VPM` (referring to nucleus ventralis posteromedialis; produces synchronous spikes), `Video`, and `spikes`. Note that external inputs use common indexing with the neuron groups. We recommend using the index 0 for the input group and indexing neuron groups from 1.

VPM `idx{int}`: Index of the neuron group.

`type{VPM}`:

`number_of_neurons{int}`: number of thalamocortical fibers.

`radius{float*unit}`: Total radius of all thalamocortical fibers, e.g. 60*um.

`spike_times{float*unit}`: stimulation spike times, e.g. [1.0 3.0]*ms means input spikes at 1.0 and 3.0 ms.

`net_center`: defines the center of the thalamocortical fibers in complex coordinates float+floatj.

`monitors`: Monitors for recording spikes or state variables. *More information on monitors*

Video `idx{int}`: Index of the `NeuronGroup()`.

`type`: videos

`path`: relative path to the input .mat file.

`[freq]`

`[monitors]`

spikes `idx{int}`: Index of the `NeuronGroup()`.

`type`: spikes

`input_spikes_filename`: path to the spike file.

`[monitors]`

This is an example of defining a video input for the system:

row_type	idx	type	path	freq	monitors
IN	0	video	./V1_input_layer.mat	190*Hz	[Sp]

In this example an input `NeuronGroup()` with index 0 is created based on the `V1_input_layer_2015_10_30_11_7_31.mat` file with a frequency of `190*Hz` and a `SpikeMonitor()` is set on it. Here's another example for `VPM` input for the system:

row_type	idx	type	number_of_neurons	radius	spike_times	net_center	monitors
IN	0	VPM	60	92*um	[0.5]*second	–	[Sp]

3.1.3 Neuron groups

Neuron groups (cell types) are defined using the following parameters. Note that biophysical parameters of the corresponding neuron groups are defined in the *Physiology configuration*. If you add a subtype, you need to add a corresponding entry to the physiology configuration file.

There are five hard-coded (neocortical) cell types in CxSystem2. The two excitatory cell types are spiny stellate (SS) and PC (pyramidal cell). The three inhibitory cell types are basket cell (BC), Martinotti cell (MC) and L1 inhibitory cell (L1i). The user can easily define subtypes, e.g. L4_MC. Subtypes can have arbitrary names (e.g. MyFavouriteBasketCellType, L5_LBC).

NeuronGroups `idx{int}`: Running index of the neuron group.

`number_of_neurons{int}`: Number of neurons.

`neuron_type{L1i,PC,BC,MC,SS}`: Neuron type (one of the hard-coded types).

`neuron_subtype{string}`: Neuron subtype (can be an arbitrary string, or – if no subtype is needed).

`layer_idx`: Layer where the neuron population is located (layer 2/3 = 2). For PCs, please use the [X->Y] syntax, where X is soma layer and Y is the most distal apical compartment. Note that PC the [X->Y] syntax creates X minus Y compartments above soma layer. E.g. for [4->1], the PC will have 3 compartments at soma layer [basal (b), soma (s) and apical0 (a)] and 3 apical dendrite compartments above soma layer [a1 at layer idx 3, a2 at layer idx 2 and a3 at layer idx 1]

`net_center{float+floatj}`: Center point of the neuron population in complex coordinates (e.g. 0+0j).

`monitors`: Monitors for recording spikes and neuron state variables, e.g. [Sp]. *More information on monitors*

`n_background_inputs{int}`: Number of excitatory background synapses.

`n_background_inhibition{int}`: Number of inhibitory background synapses.

3.1.4 Connections

Connections between neuron groups are defined using the following parameters. We currently have the following synapse types: *Fixed*, *Depressing* and *Facilitating*. Short-term plasticity (STP) parameters of the depressing and facilitating synapses are defined in the *Physiology configuration*.

Connections `receptor{ge,gi}`: Sets whether the connection is excitatory or inhibitory.

`pre_syn_idx{int}`: Presynaptic neuron group index.

`post_syn_idx`: Postsynaptic neuron group index. When targeting a PC, please use the X[C]Y syntax, where X is the neuron group index and Y is the compartment index. See below for an example.

`syn_type`: Synapse model.

`p{float<=1}`: Connection probability.

`n{int}`: Number of synapses per connection.

`monitors`: Monitors for synaptic state variables. *More information on monitors*

`load_connection{0,1}`: Flag for loading the connection and its parameters.

`save_connection{0,1}`: Flag for saving the connection and its parameters.

`custom_weight{float*unit}`: Synaptic weight for this specific connection, e.g. 1.5*nS. Overrides *more general weight definitions*.

`spatial_decay{float,[ij],float[ij]}`: When sys_mode is expanded, provides lambda (spatial decay parameter) for weight * p * exp(-lambda * d); d is the distance between neurons.

If the postsynaptic neuron is a multicompartmental neuron, the target compartment must be defined using the `[C]` tag. Compartmental indexing starts from zero at the soma layer and increases towards the distal apical dendrite. The soma, and the basal dendrites and the first apical dendrite compartment are located in the soma layer are distinguished with s, b and a tags, respectively.

For example, if you have PC neuron with a `layer_idx` of [6->1] (soma in layer 6 and apical dendrite extending up to layer 1), the compartmental indexing is:

Comp. Index	Compartment type	Layer
4	Apical dendrite (distal)	1
3	Apical dendrite	2/3
2	Apical dendrite	4
1	Apical dendrite	5
0a	Apical dendrite (proximal)	6
0s	Soma	6
0b	Basal dendrites	6

Thus, if you want target the most distal apical compartment of this group, the `post_syn_idx` should be neuron_group_index[C]4.

3.1.5 Monitors

Both neuron groups and synapses can be monitored, i.e. their state variables can be recorded and stored for analysis. Most commonly users only need the spikes. Note that continuous state variables (like the membrane voltage) are recorded with the same resolution as the time step, and thus large networks can quickly create gigabytes of data.

The following tags can be used to define a specific monitor:

`[Sp]`: This tag defines a `[Sp]` ike monitor.

`[St]`: This tag defines a `[St]` ate monitor.

You can combine a spike monitor with multiple state monitors like this (note the space between [Sp] and [St]):

`[Sp] [St]ge_soma+gi_soma+vm`.

By default all neurons/synapses are being monitored. If you want to monitor specific neurons (or synapses), you should use the `[rec]` tag followed by indices of interest. For example, to monitor the membrane voltage (vm) of the first 20 neurons (in the group) and the excitatory conductance (ge_soma) of every evenly indexed neuron between 0 and 100, you would write:

```
[St]vm[rec](0-20)+ge_soma[rec](0-100-2)
```

Often you want to assign a specific type of monitor to several consecutive neuron groups (or connections). In this case, the monitor can be defined for the first neuron group and a `-->` tag should be written at the end of the line. `-->` indicates that all the consecutive neuron groups should be assigned with the same monitor. For finishing this assignment, a `<--` symbol should be put at the last target line of interest. Note that it is possible to overwrite the defined monitors of some lines between the `-->` and `<--` symbols simply by adding the monitor of the interest.

G1	[St]ge_soma –>
G2	–
G3	
G4	[Sp]
G5	<–

In this example, a state monitor over *ge_soma* is assigned to neuron groups 1, 3 and 5 by using the `-->` and `<--` tags. For the second group, the usage of default state monitor is over-written by using the `--` keyword, indicating that the second line is not monitored. For the fourth group, however, the default monitor is overwritten by a spike monitor.

3.2 Physiology configuration

The physiological configuration consists of

- *General model parameters*
- *Neuron-type specific parameters*
- *Connection- and synapse-type specific parameters*
- *Other simulation parameters*

The physiology configuration consists of constants (*variable-values*) and dictionaries (*variable-key-values*).

There are five hard-coded neuron types in CxSystem2 (two excitatory, SS and PC; three inhibitory, BC, MC and L1i), but the user can easily define subtypes, e.g. L4_MC. Subtypes can have arbitrary names (e.g. MyFavouriteBasketCellType, L5_LBC). Each 'model' refers to Brian equations, which are explained in the *Neurodynlib* section. Advanced users are able to add new neuron types, including new membrane equation models. See the *Developer's Guide*.

3.2.1 General model parameters

General model parameters affect all neurons and synapses. They include the point neuron model, compartmental PC neuron model, receptor models, receptor weights and background noise rates. Note that the model names must be surrounded by single-quotes.

These parameters are included in the example configuration files, including some short explanations in comments.

The parameters are:

Physiology `model_variation{0,1}`: This should be 1. Value 0 is only for backwards compatibility, i.e. old models that use equations hard-coded in CxSystem1.

`neuron_model{'string'}`: Specifies which neuron model to use for point neurons (other cells than PCs). *Available neuron models*.

`excitation_model{'string'}`: Specifies the model for excitatory receptors in point neurons. *Available receptor models*.

`inhibition_model{'string'}`: Specifies the model for inhibitory receptors in point neurons. *Available receptor models*.

`pc_excitation_model{'string'}`: Specifies the model for excitatory receptors in pyramidal cells. *Available receptor models*.

`pc_inhibition_model{'string'}`: Specifies the model for inhibitory receptors in pyramidal cells. *Available receptor models*.

`background_rate{float*Hz}`: Sets the rate for excitatory background synapses (Poisson-distributed). Receptors will be modeled as `excitation_model`.

`background_rate_inhibition{float*Hz}`: Sets the rate for inhibitory background synapses (Poisson-distributed). Receptors will be modeled as `inhibition_model`.

`background_E_E_weight{float*unit}`: Weight of excitatory-to-excitatory synapses.

`background_E_I_weight{float*unit}`: Weight of excitatory-to-excitatory synapses.

`background_I_E_weight{float*unit}`: Weight of inhibitory-to-excitatory synapses.

`background_I_I_weight{float*unit}`: Weight of inhibitory-to-inhibitory synapses.

3.2.2 Neuron type-specific parameters

Neuron type-specific parameters are given as dictionaries. Subtype-specific parameters (e.g. L23_MC, L4_MC) will override parameters for the hard-coded types (MC). Parameter names must match those defined in neurodynlib. For example, to define parameters for BC neurons that are modelled as exponential integrate-and-fire (EIF) neurons, you would write:

BC	C	100 * pF
	gL	10 * nS
	EL	-70 * mV
	VT	-40 * mV
	DeltaT	2 * mV
	Ee	0 * mV
	Ei	-75 * mV
	tau_e	3 * ms
	tau_i	8 * ms
	V_res	VT - 4 * mV
	Vcut	VT + 5*DeltaT
	V_init_min	EL
	V_init_max	VT

This would define a general basket cell type. You could then continue by defining parameters for L23_NBC (L2/3 nest basket cell) and setting the `neuron_subtype` as L23_NBC for the corresponding group in the *Anatomy configuration*.

As shown in the example, you can refer to parameters (and do computations using them!) defined earlier in the Physiology configuration.

For pyramidal cell type (PC) you have additional parameters:

PC	Cm	1.0 * ufarad * cm ** -2
	Area_tot_pyram	10000 * um**2
	fract_areas	{3: array([0.58, 0.052, 0.20, 0.15, 0.01, 0.01])}
	Ra	[100,100,150,150,150] * Mohm
	spine_factor	2

Here, capacitance is defined per area unit, and total area of the PC is defined. The fract_areas provide the area fractions for each compartment as follows {N apical dendrite compartments above soma layer : array([basal_dendr, soma, apical d comp 0 at soma layer, a1 at the 1st-, a2 at the 2nd-, a3 at the 3rd layer above soma])} Ra provides the resistances between aforementioned compartments, thus len(Ra) = len(fract_areas array) - 1. The spine factor provides multiplier for capacitance for accounting the membrane surface addition by the dendritic spines.

We also use rheobase parameter to allow tonic current injections in relation to rheobase.

3.2.3 Connection- and synapse-type specific parameters

Connection weights and connections delays are also given as dictionaries (`cw` and `delay`, respectively). These are given with reference to the hard-coded cell types. Pathway-specific connection weights can be set in the anatomy

configuration using the `custom_weight` parameter. If there is no custom_weight defined, then values in the `cw` dictionary will be used. Currently there is no way to define pathway- or neuron subtype-specific delays.

These dictionaries are included in the example configuration files. Even though you might not have all the hard-coded cell types in your model, you don't need to delete the redundant lines.

3.2.4 Other simulation parameters

There are some additional parameters that are sometimes required. The most important are parameters related to short-term plasticity (STP) and to connection weight scaling by extracellular calcium concentration. To see how these are implemented, please see Methods in Hokkanen et al. 2019 Neural Computation.

For depressing synapses, you need the following parameters:

STP-Depressing `U_E{float}`: Utilization factor for depressing excitatory synapses.

`U_I{float}`: Utilization factor for depressing inhibitory synapses.

`tau_d{float*unit}`: Recovery time constant (from depression).

For facilitating synapses:

STP-Facilitating `U_f{float}`: Utilization increment for facilitating synapses.

`tau_f{float*unit}`: Facilitation decay time constant.

`tau_fd{float*unit}`: Recovery time constant for facilitating synapses.

If you want to scale synapse weights with respect to extracellular calcium level, you should define the following parameters:

calcium `calcium_concentration{float}`: Calcium concentration in mM. If set to 2.0, there is no scaling.

`flag_background_calcium_scaling{0,1}`: Sets whether background inputs are also scaled with respect to calcium level.

3.3 Batch simulations

Array run (Parallel runs) can be set using the curly braces around the target parameter. There are two special characters indicating either an explicit vector of values (ampersand, &) or numpy style range with step as the third value (vertical bar, |).

For instance, to run 3 separate simulationswith `scale=1`, `scale=2` and `scale=3`, the parameter scale should be set to:

...	scale	...
...	{1&2&3}	...

This parallel run will use the number of processes (threads) that is set using the `number_of_process` parameter, e.g. if `number_of_process=3`, then each of the 3 simulations runs in their own threads. However, if `number_of_process=2`, two processes run first the simulation for `scale=1`, and `scale=2`. The third simulation with `scale=3` will start when the first of the two simulations are completed.

The array_run could also be set in range with defined step:

...	scale	...
...	{1|5|1}	...

This parallel run will use four simulations with `scale=1`, `scale=2`, `scale=3` and `scale=4`. Note the numpy style vector excluding the last index.

When two or more parameters are set to use array runs, CxSystem can run the parallel runs either as multi-dimensional runs or independent runs. For example: suppose a simulation is to be performed for scale `{1&2&3}` with `init_vms` set to `{0,1}`. If `multidimension_array_run` flag is set to 1, the following 6 simulations will be run separately:

```
{scale=1, init_vms=0}, {scale=1, init_vms=1}, {scale=2, init_vms=0}, {scale=2, init_
↪vms=1}, {scale=3, init_vms=0}, {scale=3, init_vms=1}
```

When `multidimension_array_run flag` is set to 0, however, the array_run pattern is different and 5 simulations will be run in parallel:

```
{scale=1}, {scale=2}, {scale=3}, {init_vms=0}, {init_vms=1}
```

One might want to run each of the parallel simulations several times, e.g. to observe an effect of random initialization on a particular parameter set. For this purpose the `trials_per_config` should be set to the desired number of runs per configuration.

3.4 Running on cluster

In Cluster Run mode, CxSystem connects to a connection node of a cluster via SSH and runs instances of array run on pre-defined number of nodes in the cluster. Currently Cluster Run is tested on Taito supercluster at Finnish IT Center for Science that employs Slurm workload manager. However, with a slight modification to the Slurm template file, one can use the cluster run on other types of workload manager systems, e.g. Torque.

3.4.1 Setting up the environment

We assume that you are familiar with and have access to and available resources for a computing cluster, and that your cluster accepts ssh connections. Before submitting the batch jobs to the cluster with CxSystem, the cluster environment should be properly set up. This includes finding/copying/assigning the python environment in the Slurm file. In case you are using a custom branch you should as well clone and checkout to that branch otherwise CxSystem will clone itself and try to spawn the processes using the master branch. The parameters that are used for cluster run are as follows:

run_in_cluster
1

which triggers the cluster run.

cluster_job_file_path
./csc_taito.job

This parameter defines the address of the template workload management system file. In this case the a template for Slurm system is made available in the Github page which can be used for any cluster that utilizes the Slurm. In case the cluster of choice uses another workload management system, the template file should be specified accordingly.

cluster_number_of_nodes
10

Defines how many nodes will be requested from the cluster to CxSystem.

Important Note: the number of nodes in the cluster workload management system file should be set to 1 and instead the number of nodes should be defined here. The reason for this is that CxSystem submits separate jobs to each node in the cluster. This has some advantages, including less waiting time, and some disadvantages, complexity.

cluster_address
taito.csc.fi

Defines the URL of the cluster.

cluster_username
johnsmith

Defines the username for the SSH connection.

remote_repo_path
~/CxSystem

Defines the path of the CxSystem in the connection node of the cluster. In this example `~/CxSystem` indicates that the CxSystem clone exists in the home folder, i.e. `~/`, in the connection node. If the repository clone does not exist there, then the latest version will be cloned and used. As mentioned earlier, in case the user is using a specific branch, CxSystem should be cloned and checked out manually. Otherwise the user can rely on CxSystem cloning itself in the connection node.

cluster_workspace
/Users/cxuser/results

Defines the folder in which the results will be copied. In this example, a new folder `results` will be created in the `/Users/cxuser/` folder and the results will be saved there. Note that `cluster_workspace` must be an absolute path and in case the results reside under the home folder, it's path must as well be explicitly defined.

3.4.2 Transferring the Results

Usually clusters have an option to send users an email when the job is finished and users can transfer the results to their local machine. CxSystem can do this automatically: When a batch job is submitted, besides generating several Slurm files, CxSystem creates a `_tmp_checker_data` file. This file, which contains information about the current cluster batch job, will be used to check the status of the results in the cluster. If the results are ready, they will be copied to the local result folder defined using `output_path_and_filename` attribute in the network and model configuration file. This task can be done by directly running `cluster_run.py`.

3.5 Visualization

Visualization of simulation data often requires custom code, but we provide two ways to get a general idea of the simulated spike data.

3.5.1 rasterplot-pdf

Rasterplot-pdf allows you to create a pdf file with rasterplots of a batch of simulations. You can run it using the `cxvisualize` command (*more on the command*). For example, to create raster plots of all simulations run on 13 December 2019 at 13:37 in a particular directory, you can write

```
cxvisualize --rasterplot-pdf ~/CxWorkspace/my_simulation/ 20191213_1337
```

This creates a pdf file with the specified timestamp in the same directory. Note that this script is very basic and is intended only for screening that there is not a simple bug in the simulation configuration.

3.5.2 ViSimpl

ViSimpl allows you to visualize the time course and geometry of spiking in a single simulation. First, download the ViSimpl binary (the .AppImage binary is for Linux and the .dmg binary is for macOS). Then put the ViSimpl binary in a directory that is located in your system path. Also, remove version numbering from the binary and make it executable:

```
mv visimpl-0.1.4-x86_64.AppImage visimpl.AppImage
chmod +x visimpl.AppImage
```

After this you can visualize CxSystem2 spike data in ViSimpl using the `cxvisualize` *command*:

```
cxvisualize ~/CxWorkspace/my_simulation/my_simulation_results.gz
```

This opens up the ViSimpl main screen. You should be able to rotate the circuit and "play" the simulation without further configuration. To learn how you can customize the visualization, please see the ViSimpl website.

3.5.3 Custom visualizations

Simulation data are stored in dictionaries in the results file. For example, to access spike data in a gzipped file, you can write:

```
import zlib
import pickle

fi = open('my_simulation_results.gz', 'rb')
data_pickle = zlib.decompress(fi.read())
data = pickle.loads(data_pickle)

spike_data = data['spikes_all']
neuron_positions = data['positions_all]
```

After this, spikes are available in the `spike_data` dictionary indexed by neuron group names. Then, for each neuron group, you have a dictionary with arrays `i` (neuron index) and `t` (spike time).

Similarly, neuron positions are available in the `neuron_positions` dictionary containing two subdictionaries: `w_coord` (cortical position) and `z_coord` (retinal position). If you are not modelling the visual system, you can ignore the `z_coord` subdictionary.

3.6 Command-line interface

CxSystem2 has five commands associated with the main functionalities of CxSystem2. Each of this commands have descriptive help commands.

3.6.1 cxsystem2

This command can be used to run a simulation using anatomy and configuration files:

```
$ cxsystem2 -h
cxsystem2 -- a cortex simulation software

Usage:
  cxsystem2 (-h | --help)
  cxsystem2 -a ANATOMY_FILE -p PHYSIOLOGY_FILE
  cxsystem2 (-v | --version)

A cortex simulation framework called `CxSystem`.

Arguments:
  PORT                                              port number on which the BUI␣
↪server runs

Options:
  -h --help                                         Show this screen
  -v --version                                      Show current cxsystem version
  -a ANATOMY_FILE --anatomy=ANATOMY_FILE            Sets the anatomy file path
  -p PHYSIOLOGY_FILE --physiology=PHYSIOLOGY_FILE   Sets the physiology file path


Description:

  cxsystem2 -a ./anatomy.csv -p ./physiology.csv
    runs the cxsystem using the anatomy file called anatomy.csv and physiology file␣
↪called physiology.csv

  cxsystem2 --anatomy ./anatomy.csv --physiology ./physiology.csv
    runs the cxsystem using the anatomy file called anatomy.csv and physiology file␣
↪called physiology.csv
```

3.6.2 cxvisualize

This command can be used to visualize the result of a simualtion using `ViSimpl` (available here for download):

```
$ cxvisualize -h
cxvisualize -- cxsystem visualizer

Usage:
  cxvisualize (-h | --help)
  cxvisualize  [ -d | -c ] FILEPATH
  cxvisualize --rasterplot-pdf FOLDERPATH TIMESTAMP [--sampling-rate SAMPLINGRATE]

A tool for visualizing `CxSystem` spike data in ViSimpl.
```

(continues on next page)

(continued from previous page)

```
Arguments:
  FILENAME                                          Path to results file
  FOLDERPATH                                        Path to folder containing array␣
↪run files
  TIMESTAMP                                         Timestamp suffix of the arrayrun␣
↪files
  SAMPLINGRATE                                      Sampling rate for the rasterplots

Options:
  -h --help                                         Show this screen
  --rasterplot-pdf                                  Generate a rasterplot pdf for␣
↪arrayrun
  -c --convert                                      Convert results for use in ViSimpl
  -d --delete                                       Delete ViSimpl-related files␣
↪after visualization
  -s SAMPLINGRATE --sampling-rate=SAMPLINGRATE      Sampling rate for raster plot␣
↪(default is 1%)

Description:

  cxvisualize ./results.gz
    converts the results file into two CSVs and one JSON for ViSimpl, visualizes the␣
↪result and does not remove the temp files
    (actual visualization: ./visimpl -csv results_structure.csv results_spikes.csv -
↪se results_subsets.json)

  cxvisualize -d ./results.gz
    converts the results file into two CSVs and one JSON for ViSimpl, visualizes the␣
↪result and remove the temp files

  cxvisualize -c ./results.gz
    converts the results file into two CSVs and one JSON for ViSimpl (no␣
↪visualization)

  cxvisualize --rasterplot-pdf ./cobaeif 20191123_1353509
    Generates a pdf of rasterplots of all groups in the folder with 20191123_1353509␣
↪timestamp

  cxvisualize --rasterplot-pdf ./cobaeif 20191123_1353509 --sampling-rate=4%
    Generates a pdf of rasterplots of all groups in the folder with timestamp␣
↪20191123_1353509 and sampling rate or 4%
```

3.6.3 `cxconfig`

CxSystem2 supports both *csv* and *json* file formats for the anatomy and physiology configurations. `cxconfig` can be used to convert the *json* configuration files to *csv* and vice versa:

```
$ cxconfig -h
cxconfig -- cxsystem config file converter

Usage:
  cxconfig (-h | --help)
  cxconfig  FILEPATH
```

(continues on next page)

(continued from previous page)

```
A configuration file converter for `CxSystem`.

Arguments:
  FILENAME              Path to configuration file to convert

Options:
  -h --help             Show this screen

Description:

  cxconfig ./sample_conf.json
    converts the file `sample_conf.json` in current directory to csv and saves it as␣
↪`sample_conf.csv`

  cxconfig ./sample_conf.csv
    converts the file `sample_conf.csv` in current directory to csv and saves it as␣
↪`sample_conf.json`
```

3.6.4 cxcluster

After submitting jobs to cluster, *CxSystem* stores the data corresponding to the job to a metadata file. To retrieve the results when they are ready, you can use the `cxcluster` command as follows:

```
$ cxcluster -h
cxcluster -- cxsystem cluster result downloader

Usage:
  cxcluster (-h | --help)
  cxcluster META_FILE_PATH


Downloads result data of `CxSystem` from cluster.

Arguments:
  META_FILE_PATH            Path to cluster run metadata file

Options:
  -h --help                 Show this screen

Description:

  cxcluster ./sample_meta.pkl
    Uses the information in the metadata file to download the results if ready, and␣
↪otherwise wait for the results.
    User will be prompted for the remote password.
```

3.6.5 cxserver

You can use the `cxserver` command to run the browser user interface of the CxSystem2 in either *http* or *https* mode. Note that running it in *https* mode requires *oauth* configurations for authentication:

```
$ cxserver -h
cxserver -- web server for cxsystem2
```

(continues on next page)

(continued from previous page)

```
Usage:
  cxserver (-h | --help)
  cxserver [--port=PORT] [--no-browser]
  cxserver --https [-p PROVIDERID -c CLIENTID -r REDIRECTURI -a AUTHORIZATION] [--
↪port=PORT] [--no-browser]
  cxserver --config -p PROVIDERID -c CLIENTID -r REDIRECTURI -a AUTHORIZATION
  cxserver --config -w WORKSPACEPATH
  cxserver --config -l LOGPATH

Web server for running the BUI for `cxsystem2`

Arguments:
  PORT                                              port number on which the BUI␣
↪server runs
  PROVIDERID                                        provider id for OAuth2 client for␣
↪authentication
  CLIENTID                                          client id for OAuth2 client for␣
↪authentication
  REDIRECTURI                                       redirect url for OAuth2 client␣
↪for authentication
  AUTHORIZATION                                     authorization url for OAuth2␣
↪client for authentication
  WORKSPACEPATH                                     path to the main workspace folder
  LOGPATH                                           path to the log folder

Options:
  -h --help                                         Show this screen
  -v --version                                      Show current cxsystem version
  --https                                           Run server with ssl certificate
  --port=PORT                                       Runs the server on port PORT
  --config                                        Rewrite the oauth config file with␣
↪the new parameters
  --no-browser                                      Do not open browser after running␣
↪the server
  -p PROVIDERID --provider-id=PROVIDERID            Sets the provider id
  -c CLIENTID --client-id=CLIENTID                  Sets the client id
  -r REDIRECTURI --redirect-uri=REDIRECTURI         Sets the redirect url
  -a AUTHORIZATION --authorization=AUTHORIZATION    Sets the authorization url
  -w WORKSPACE --workspace-path=WORKSPACE           Sets the workspace path
  -l LOGPATH --log-path=LOGPATH                     Sets the log path

Description:

  cxserver
    runs the cxsystem2 server without SSL certificate on a random port

  cxserver --port=PORT
    runs the server on a specific port number PORT

  cxserver --https
    runs the cxsystem using the ssl certificate and other parameters previously saved␣
↪in the configuraiton file using --config

  cxserver --config -p HBP -c f34780ff-7842-499c-8440-5777c28e360d -r https://127.0.0.
↪1:4443 -a https://services.humanbrainproject.eu/oidc/authorize
    config the configuration yaml file with the new oauth parameters
```

(continues on next page)

(continued from previous page)

```
cxserver --config -w /cxworkspace
  config the configuration yaml file with the new workspace path

cxserver --config -l /var/log/
  config the configuration yaml file with the new log path
```

3.7 Running the BUI locally

You can run CxSystem2 as a local server similar to the Jupyter notebook server. This allows you to use the browser user interface (BUI) to run simulations locally or launch jobs on a remote cluster.

Please read the instructions on *installing CxSystem2 locally* and on using *the cxserver command*.

Visualization Command-line interface Running a local server

CHAPTER 4

Developer's Guide

This section section provides a brief guideline for potential contributors and researchers who need to change the CxSystem2 code e.g. for adding new neuron or synapse types.

4.1 Technical Overview

4.1.1 Selecting Python, C++ or GPU device

The device is selected in the model and network configuration file. Set the "device" to either `Python`, `Cpp` or `GeNN` (case insensitive). The `Cpp` (C++) device is a safe bet for most applications. `Python` skips compilation, and may be best for minor systems or when devices run into problems. `GeNN` may be beneficial for long non-array runs.

4.1.2 How the CxSystem works

The CxSystem starts by calling the main object `cxsystem()` in python interpreter. The configuration files are set either in the BUI, as command-line arguments for cxsystem2 command or python interpreter after cxsystem() call, or at the end of cxsystem.py file.

One of the strengths of the CxSystem is the ability to dynamically compile the model. This bypasses the traditional way of hard coding much of the model which would limit flexibility. This flexibility comes with some added complexity in the way the CxSystem builds the devices.

The implemented system employs the Brian2GeNN python module to generate GeNN (GPU enhanced Neuronal Network simulation environment) code for eventually running the Brian2 codes on GeNN. Note that using the GeNN device, CxSystem (via GeNN) only employs one of the GPUs in the system and therefore cannot be used in cluster. In order to understand how this system works, one should initially understand how Brian2GeNN limits Brian2. Most of the exclusions are presented in Brian2GeNN documentation. Perhaps the most effective limitation is lack of support for using Multiple networks in Brian2, i.e. only the *magic network* can be used. Using the *magic network*, only the "visible" objects, that are explicitly defined in the code will be collected. In other words, any Brian2 object that is created in a custom class, will not be collected and will eventually raise an error. We have used two solutions to address this issue, Syntax Bank and Global variables:

Creating a Syntax Bank

In this method, a syntax string is built for all Brian2 internal objects. These syntaxes are then run after the main object call. Suppose the cortical system object is named *CX* and a `NeuronGroup()` object called `NG` is created in a method inside the `CX`:

```
NG = NeuronGroup(1, eqs)
```

The `NG` will not be collected for magic network as it is inside a method of `CX`. However, we can anticipate a syntax for this neuron group and save it in a syntax bank attribute in `CX`:

```
syn1 = "NG = NeuronGroup(1, eqs)"
CX.syntax_bank = append(CX.syntax_bank, syn1)
```

All of the elements of this `CX`.syntax_bank* can then be iterated and run using the dynamic compiler, i.e. `exec` command. Note that all the sub objects of a syntax should be saved in syntax_bank as well. For instance, the last example will raise an error since `eqs` is not defined. Hence, before running the `syn1`, one should initially run the syntax for `eqs` object.

This method has a fundamental limitation: first, the syntax bank should run in a hierarchical manner. In previous example, the syntax for `eqs` should be run before `syn1`. Similarly, `NeuronGroup()` syntaxes should be run before `Synapses()` and `synapses.connect()` should be run after `Synapses()`. This process was manually coded into the main file for running the codes in a hierarchical manner, which we consider an untidy solution.

The syntax bank approach call for prefixes for object names. For instance, all of the `NeuronGroup()` have a prefix of `NG`.

For each neuron group, similar prefixes are also needed for variables such as:

- Number of neurons in each group: `NN`
- Equation: `NE`
- Threshold value: `NT`
- Reset value: `NRes`
- Refraction value: `NRef`
- Namespace: `NS`

Several prefixes are also demanded for `Synapses()` objects:

- Synaptic object: `S`
- Synaptic equation: `SE`
- Pre Synaptic group equation: `SPre`
- Post Synaptic group equation: `SPost`
- Namespace: `SNS`
- .connect(): `SC`
- weight: `SW`

And similar prefixes for monitors:

- Spike Monitors: `SpMon`
- State Monitors: `StMon`

Updating Globals()

Although mentioned as a dangerous method in the literature, updating the `Globals()` directly, is a practical approach in our case. This method uses aforementioned prefixes and corresponding variables. However, there is no need for the newly generated variables to *wait* in the syntax bank so to be run after `CxSystem()` module. They can be implicitly executed while CxSystem is running and still magic network of Brian2 would be able to access them since they are in `Globals()`. Thus, the user does not have to face a manual syntax-executer outside of the main object call.

Accordingly, most of the `exec` commands inside the main object `CxSystem()` are creating the required variables and making them visible to *magic network* of Brian2 by updating the `Globals()`. In the following example, the `NG0` is put into the `Globals()`:

```
globals().update({'NG0':NG0})
```

Fig.1 illustrates the schematic of the cortical system internal component:

4.2 Documentation

4.2.1 Visual Studio Code

We strongly suggest to use the *vscode* for updating the documentation. After installing the *vscode*, add the following useful extensions:

- *Code Spell Checker*: used for spell checking
- *reStructuredText*: shows the format errors in the *rst* files
- *rewrap*: fix the line wrap of a paragraph with *Alt+q* shortcut

At this point all the errors that corresponds to *reStructuredText* format will be shown in the *vscode*. After adding paragraphs, make sure to press *Alt+q* so that the long lines are wrapped. You can also go to setting and setting the *Editor: Word Wrap* to *wordWrapColumn*. However, you will still need to press *Alt+q* to fix the wrap for *rst* format, otherwise *reStructuredText* extension will underline the line as a long line (*D001 error*).

4.2.2 Building the documentation

We highly recommend building the documentation locally specially when updating/adding docstrings. After changes to the documentation you can use *make html* to rebuild the documentation and check the *index.html* to make sure your changes are reflected correctly.

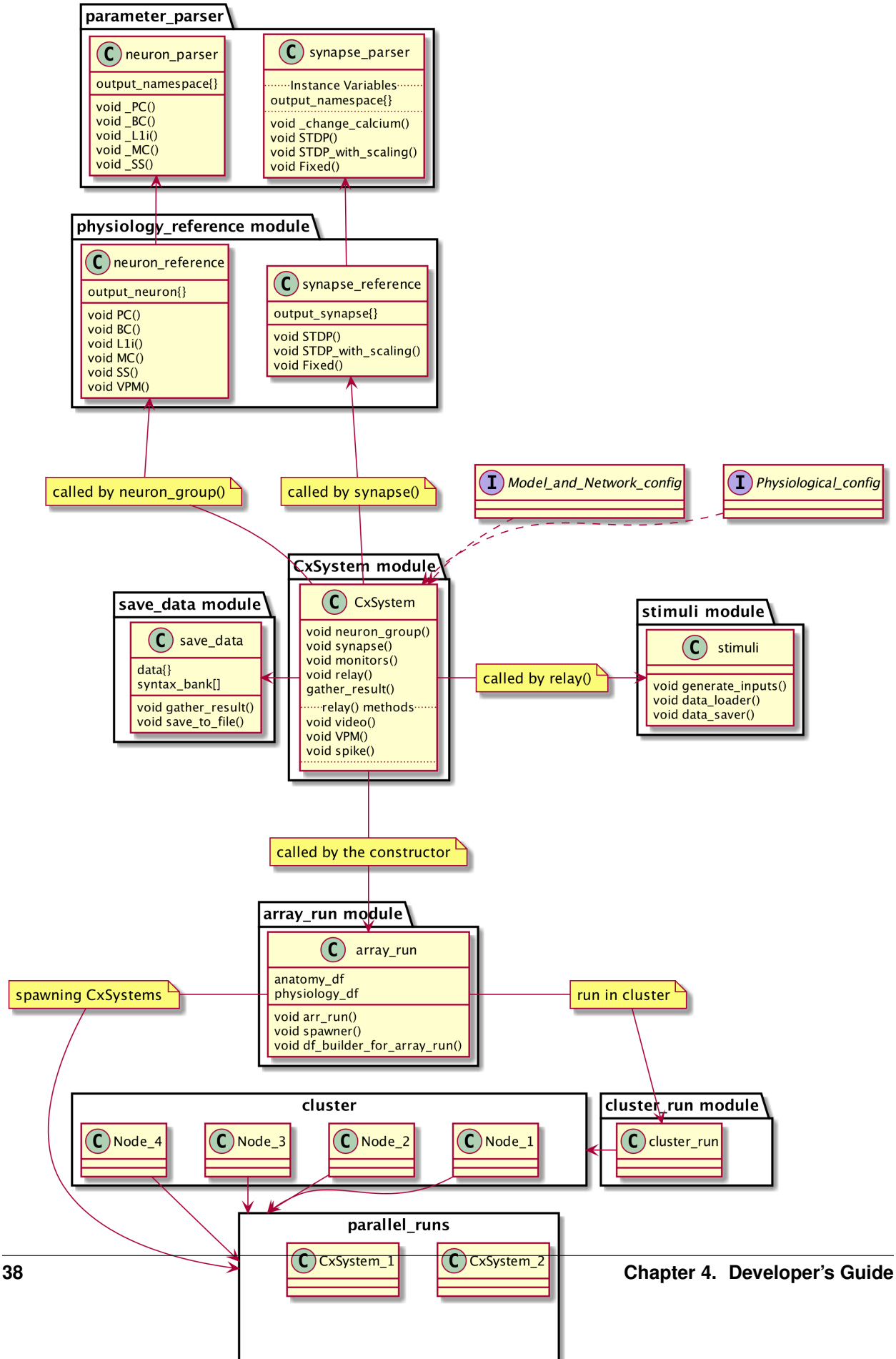
Sphinx is well documented, yet we will provide the essentials for improving the documentation of the CxSystem. The documentation for reStructuredText is available here.

To build the documentation locally you will need the *sphinx* and *sphinx-rtd-theme* packages which are already included in the *requirements*. So if you have created an environment for setting up CxSystem2, you can activate the environment, otherwise install the packages with *pip*.

When sphinx is installed, you can build the documentation using the following command:

```
$ cd CxSystem2/docs
$ make html
```

After compiling the documentation, you can open the updated local documentation by opening the following file in the browser:


parameter_parser
neuron_parser
output_namespace{}
void _PC()
void _BC()
void _L1i()
void _MC()
void _SS()
synapse_parser
Instance Variables
output_namespace{}
void _change_calcium()
void STDP()
void STDP_with_scaling()
void Fixed()
physiology_reference module
neuron_reference
output_neuron{}
void PC()
void BC()
void L1i()
void MC()
void SS()
void VPM()
synapse_reference
output_synapse{}
void STDP()
void STDP_with_scaling()
void Fixed()
called by neuron_group()
called by synapse()
Model_and_Network_config
Physiological_config
CxSystem module
CxSystem
void neuron_group()
void synapse()
void monitors()
void relay()
gather_result()
relay() methods
void video()
void VPM()
void spike()
save_data module
save_data
data{}
syntax_bank[]
void gather_result()
void save_to_file()
called by relay()
stimuli module
stimuli
void generate_inputs()
void data_loader()
void data_saver()
called by the constructor
array_run module
array_run
anatomy_df
physiology_df
void arr_run()
void spawner()
void df_builder_for_array_run()
spawning CxSystems
run in cluster
cluster
Node_4
Node_3
Node_2
Node_1
cluster_run module
cluster_run
parallel_runs
CxSystem_1
CxSystem_2

```
CxSystem2/docs/build/html/index.html
```

4.2.3 Docstrings and *reference.rst*

Sphinx generates a documentation based on the docstrings of the functions, methods and classes. The definition of the entities that are parsed are defined in the *reference.rst*. The following example shows how we can show the target methods and functions to *sphinx* for parsing:

```
.. module:: cxsystem2.neurodynlib.tools.spike_tools
.. autoclass:: PopulationSpikeStats
      :members:

      .. automethod:: PopulationSpikeStats.__init__

.. autofunction:: get_spike_time
.. autofunction:: get_spike_stats
```

The first line of this example defines the file that should be parsed. In this case the filename is *spike_tools.py* and it is located at *cxsystem2/neurodynlib/tools/spike_tools.py*. The *autoclass* directive defines the class that should be parsed from that file and the *automethod* defines the method of that class that should be parsed. In this case only *__init__()* will be parsed. Next, the *autofunction* directive defines which functions from the *spike_tools.py* should be parsed.

For the docstrings, make sure to use the *sphinxy* docstring for consistency. Moreover, make sure at least you include the parameters and return names and types and the description of the method/function. This is a simple example how the docstring should be formatted:

```
def test_func(name, state=None):

      """This function does something.

      :param name: The name to use.
      :type name: str.
      :param state: Current state to be in.
      :type state: bool.
      :returns:  int -- the return code.
      :raises: AttributeError, KeyError

      """
```

Lots of more examples of docstrings are available here.

4.3 *pypi* package

To release a new version of the *cxsystem2* in *pypi*, you should first update the version. The version of the *cxsystem2* are read from the *__version__* variable available in *cxsystem2/__init__.py*. If you update this variable, all other references to the version number will be updated globally.

We will use the following versioning format for the 3rd number, from software versioning in wiki:

```
0 for alpha (status)
1 for beta (status)
2 for release candidate
3 for (final) release
```

For instance:

```
1.2.0.1 instead of 1.2-a1
1.2.1.2 instead of 1.2-b2 (beta with some bug fixes)
1.2.2.3 instead of 1.2-rc3 (release candidate)
1.2.3.0 instead of 1.2-r (commercial distribution)
1.2.3.5 instead of 1.2-r5 (commercial distribution with many bug fixes)
```

After updating the version number, make sure *twine* package is installed:

```
$ pip install twine
```

Then go to the root of the repository, and create the new distribution:

```
$ python setup.py sdist bdist_wheel
```

Finally upload the new package to *pypi*:

```
$ twine upload dist/*
```

At this point you will be prompted for the username and password for the *cxsystem2* pypi account.

4.4 Continuous Integration

4.4.1 Travis yaml file

CxSystem2 uses *Travis CI* for continuous integration. The configuration for *Travis* is available in *.travis.yml* file in the root folder of the repository. All the details of the *Travis* yaml file is available here in details.

Most of the used parameters are self-explanatory, e.g. *language*, *os*, *python*. You can install dependencies *before_install* and install using *install* keyword. You can run different scripts during the tests. We can run all the tests using *pytest tests*. This way, all the tests that are subsequently added to the tests folder will be automatically tested in travis.

4.4.2 Encrypted Parameters

One might need some encrypted parameters during the testing process, for instance to run cluster run tests, cluster password is required but cannot be saved in the configuration file in the plain text. *Travis* provides a command line client that allows us to encrypt parameters and add them to the configuration file. Install the client using:

```
sudo gem install travis
```

Then log into the client:

```
travis login
```

After logging in and testing it with *travis whoami*, you can add encrypted parameters to the configuration file:

```
travis encrypt MY_SECRET_ENV=super_secret --add env.global
```

This way you can encrypt a value in an environment variable which would be secure on Travis side and use it in the code.

4.5 Browser User Interface (BUI)

4.5.1 Folder Structure

For a better understanding of the BUI sub-module, let's first take a look at the folder structure:

```
├── __init__.py
├── bui.py
└── cx_bui
    ├── __init__.py
    ├── cx_bui
    │   ├── __init__.py
    │   └── settings.py
    ├── db.sqlite3
    ├── editor
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── migrations
    │   ├── models.py
    │   ├── templates
    │   │   └── editor
    │   │       └── index.html
    │   ├── tests.py
    │   ├── urls.py
    │   └── views.py
    ├── examples
    │   ├── cobaeif_anatomy_config.json
    │   ├── cobaeif_physiology_config.json
    │   ├── csv_configs
    │   ├── markram_step1_anatomy_config.json
    │   ├── markram_step1_physiology_config.json
    │   ├── markram_step2_anatomy_config.json
    │   ├── markram_step2_physiology_config.json
    │   ├── pytest_anatomy_config.json
    │   └── pytest_physiology_config.json
    ├── manage.py
    └── static
        ├── css
        ├── images
        └── scripts
```

The *bui.py* is the main file of the module that includes *RunServer* class. The top *cx_bui* folder is the main django folder. The *settings.py* under nested *cx_bui* folder, includes the server settings.

The *editor* folder is the main application of the django server. Nested under these folders, the important files are *views.py* which contain the view functions. The *urls.py* includes the URLs of the server that have a view in the *view.py*. The *static* folder contains the html page ingredients, including *css*, *scripts* and *images*. In the next sections, we learn some of the main points for improving CxSystem.

4.5.2 Adding Examples

The *examples* folder includes all examples that can be loaded on the client side. Inside *views.py* we have the function that loads the examples and send them back to the client. Checking the following snippet, we see that the view looks for the examples based on the file names:

```
if config_type == 'anatomy':
    filename = example_name + '_anatomy_config.json'
else:
    filename = example_name + '_physiology_config.json'
```

So, to add other examples, the configuration files should be in the following file name formats:

```
EXAMPLENAME_anatomy_config.json
EXAMPLENAME_physiology_config.json
```

After adding the example, you can edit the *index.html* template and add the example in the *Example* section in the following format:

```
<a href="#" onclick="load_example('EXAMPLENAME')">Click here</a> to load the␣
↪configuration for the EXAMPLENAME.
```

4.6 Command Line Interface

CxSystem commands are defined under *cxsystem2/cmd* folder. Each command has a separate file that includes a main function, which processes the arguments. Each of those *main* functions are defined as an entry point in the *setup.py* pypi configuration file:

```
entry_points={
    'console_scripts': [
        'cxsystem2 = cxsystem2.cmd.cxsystem_cmd:main',
        'cxconfig = cxsystem2.cmd.cxconfig_cmd:main',
        'cxcluster = cxsystem2.cmd.cxcluster_cmd:main',
        'cxvisualize = cxsystem2.cmd.cxvisualize_cmd:main',
        'cxserver = cxsystem2.cmd.cxserver_cmd:main'
    ],
}
```

Another main component of the CLI is the *docopt* strings on top of each of the files. These strings define the allowed arguments for that command. You can use the current commands as examples to create newer commands, but more examples are also available in the docopt website.

4.7 Parameters and Models

4.7.1 Adding parameters

The parameters of the Model & Network configuration file are defined and deployed in the main module, i.e. `CxSystem.py`. There are two steps to add a new parameter:

- **Defining a function:**

  This is usually a setter function that sets a module attribute using a given attribute:

```
def set_sample(self,*args):
    self.sample = eval(args[0])
```

In this example, `set_sample()` will set the sample attribute based on a given argument.

- **Add the parameter name to "*parameter_to_method_mapping*" dictionary:**

This is where the defined function in the previous step is used: by just appending the variable and function name as a key:value pair to `parameter_to_method_mapping` dictionary:

```
{
...,
'sample_parameter': [n,self.set_sample]}
```

Where `n` is the priority value in setting the parameter. The priority value is useful when one wants to define a parameter based on another but they might not be inserted in the correct order. By default, the priority value could follow the last number in the `parameter_to_method_mapping` dictionary. The priority value does not need to be unique, two values could have the same priority and they run one after another. For example:

```
self.parameter_to_method_mapping = {
'device': [0,self.set_device],
'save_input_video': [1,self.save_input_video],
}
```

Here, `save_input_video` will be set using `self.save_input_video` after `device`, that is set using `self.set_device`.

4.7.2 Adding Neuron Model

Adding a new neuron group

The hard-coded neuron types (L1i, BC, MC, SS, PC) are in the class *NeuronReference* in physiology_reference.py. Under this class, there is a separate method for each neuron group containing code that builds the membrane equation(s). Adding a new neuron group can be done by copy-pasting the method of eg. the BC neuron group and renaming it (ChC for chandelier cell):

```
def ChC(self):
     x = neuron_factory().get_class(self.neuron_model)
     x.set_excitatory_receptors(self.excitation_model)
     x.set_inhibitory_receptors(self.inhibition_model)
     ...
```

You also need to add the neuron type to the list of accepted types under the init of *NeuronReference*:

```
NeuronReference._celltypes = array([...existing neuron types..., 'ChC'])
```

Similarly, add the neuron type also to the list of accepted types under the init of `NeuronParser` (in `parameter_parser.py`), and create a method for parameter processing. Often, parameters can be used as such in the equations, so the method becomes:

```
def _ChC(self, output_neuron):
   pass
```

Please note the underscore here before the neuron group name. Now, you can use the name 'ChC' to define the connectivity and biophysical parameters in the CSV configuration files.

Note that you might need to add connection weights and delays in physiological configuration file based on the synapses you are going to use with your new neuron model. Also, neuron group equations in CxSystem must have x and y coordinates, so adding the following at the end of the equation block is neccessary:

```
self.output_neuron['equation'] += Equations('''x : meter
y : meter''')
```

After this, the neuron equation parameters should be added to Physiological configuration file.

Adding alternative neuron models to existing groups

Instead of adding a hard-coded neuron type, you typically need to add an alternative neuron model. Point neuron models are now located in the `cxsystem2.neurodynlib` submodule (soon multicompartment models as well), where you can create a class for the new neuron model. Using `PointNeuron` as a base class gives access to basic functionalities for exploring the model behavior.

For example, to add the adaptive exponential integrate-and-fire (AdEx) model, we have written:

```
class AdexNeuron(PointNeuron):

    default_neuron_parameters = {
            'EL': -70.0 * mV,
            'V_res': -51.0 * mV,
            'VT': -50.0 * mV,
            'gL': 2 * nS,
            'C': 10 * pF,
            'DeltaT': 2 * mV,
            'a': 0.5 * nS,
            'b': 7.0 * pA,
            'tau_w': 100.0 * ms,
            'refractory_period': 2.0 * ms,
            'Vcut': -30.0 * mV
    }

    neuron_model_defns = {'I_NEURON_MODEL': 'gL*(EL-vm) - w + gL * DeltaT * exp((vm-
↪VT) / DeltaT)',
                          'NEURON_MODEL_EQS': 'dw/dt = (a*(vm-EL) - w) / tau_w : amp'}
    model_info_url = 'http://neuronaldynamics.epfl.ch/online/Ch6.S1.html'
```

Then we added the init method:

```
def __init__(self):

    super().__init__()
    self.threshold_condition = 'vm > Vcut'
    self.reset_statements = 'vm = V_res; w += b'
    self.initial_values = {'vm': None, 'w': 0*pA}  # None in vm => vm initialized at␣
↪EL
    self.states_to_monitor = ['vm', 'w']  # Which state variables to monitor by␣
↪default

    # Parameters and their units that have not been defined in the base class:
    new_parameter_units = {'DeltaT': mV, 'Vcut': mV, 'a': nS, 'b': pA, 'tau_w': ms}
    self.parameter_units.update(new_parameter_units)
```

After these initial definitions, you can add model-specific methods. Finally, you need to add the neuron model to the factory method via which CxSystem accesses neurodynlib:

```
class neuron_factory:
    def __init__(self):
        self.name_to_class = {...existing models... , 'ADEX': AdexNeuron}
```

After this you can use change the *neuron_model* parameter to 'ADEX' to use the AdEx equations in your point neurons.

4.7.3 Adding Synapse Model

Synapse models are currently located in `cxsystem2.core` in physiology_reference.py. Similarly to adding new neuron groups, you need to add the new synapse types to the lists of accepted types. For example, we wanted to add a 'Depressing' synapse type (a form of short-term synaptic plasticity). First, in the init of `SynapseReference` (`physiology_reference.py`):

```
SynapseReference.syntypes = array([...existing synapse types..., 'Depressing'])
```

Then similarly in the init of *synapse_parser* (parameter_parser.py):

```
synapse_parser.type_ref = array ([...existing synapse types..., 'Depressing'])
```

Equations for the new synapse type were then added as a method under *SynapseReference*:

```
def Depressing(self):

    self.output_synapse['equation'] = Equations('''
    wght : siemens
    R : 1
    ''')

    self.output_synapse['pre_eq'] = '''
    R = R + (1-R)*(1 - exp(-(t-lastupdate)/tau_d))
    %s += R * U * wght
    R = R - U * R
    ''' % (self.output_synapse['receptor'] + self.output_synapse['post_comp_name'] +
↪'_post')
```

After these changes, one can use the reference 'Depressing' in the Anatomy configuration when defining connections between neuron groups.

CHAPTER 5

Reference Documentation

5.1 Core module

class cxsystem2.core.cxsystem.**CxSystem**(*anatomy_and_system_config=None*, *physiology_config=None*, *output_file_suffix=''*, *instantiated_from_array_run=0*, *cluster_run_start_idx=-1*, *cluster_run_step=-1*, *array_run_in_cluster=0*, *array_run_stdout_file=None*)

The main object of cortical system module for building and running a customized model of cortical module based on the configuration files.

__init__(*anatomy_and_system_config=None*, *physiology_config=None*, *output_file_suffix=''*, *instantiated_from_array_run=0*, *cluster_run_start_idx=-1*, *cluster_run_step=-1*, *array_run_in_cluster=0*, *array_run_stdout_file=None*)

Initialize the cortical system by parsing both of the configuration files.

Parameters

- **anatomy_and_system_config** – could be either the path to the anatomy and system configuration file, or the dataframe containing the configuration data.
- **output_file_suffix** – switch the GeNN mode on/off (1/0), by default GeNN is off
- **instantiated_from_array_run** – this flag, 0 by default, determines whether this instance of CxSystem is instantiated from another instance of CxSystem which is running an array run.
- **stdout_file_path** – this is only used for saving arrayrun stdout Main internal variables:

- **customized_neurons_list: This list contains the NeuronReference instances. So for each neuron group target line** there would be an element in this list which contains all the information for that particular neuron group.
- **customized_synapses_list: This list contains the SynapseReference instances. Hence, for each synapse custom lin** there would be an element in this list, containing all the necessary information.

- neurongroups_list: This list contains name of the NeuronGroup() instances that are placed in the Globals().
- synapses_name_list: This list contains name of the Synapses() instances that are placed in the Globals().
- monitor_name_bank: The dictionary containing the name of the monitors that are defined for any NeuronGroup() or Synapses().
- **default_monitors: In case –> and <– symbols are used in the configuration file, this default monitor will be appli** the target lines in between those marks.
- save_data: The save_data() object for saving the final data.

gather_result()

After the simulation and using the syntaxes that are previously prepared in the syntax_bank of save_data() object, this method saves the collected data to a file.

monitors(*mon_args*, *object_name*)

This method creates the Monitors() in brian2 based on the parameters that are extracted from a target line in configuration file.

Parameters

- **mon_args** – The monitor arguments extracted from the target line.
- **object_name** – The generated name of the current object.

Main internal variables:

- mon_tag: The tag that is extracted from the target line every time.
- mon_name: Generated variable name for a specific monitor.
- mon_str: The syntax used for building a specific StateMonitor.
- **sub_mon_tags: The tags in configuration file that are specified for a StateMonitor(), e.g. in record=True which is** [rec]True in configuration file, [rec] is saved in sub_mon_tags
- **sub_mon_args: The corresponding arguments of sub_mon_tags for a StateMonitor(), e.g. in record=True which** [rec]True in configuration file, True is saved in sub_mon_args.

neuron_group()

The method that creates the NeuronGroups() based on the parameters that are extracted from the configuration file in the __init__ method of the class.

Main internal variables:

- mon_args: contains the monitor arguments extracted from the target line.
- **net_center: center position of the neuron group in visual field coordinates, description can be found in** configuration file tutorial.
- _dyn_neurongroup_name: Generated variable name for the NeuronGroup() object in brian2.
- _dyn_neuronnumber_name: Generated variable name for corresponding Neuron Number.
- _dyn_neuron_eq_name: Generated variable name for the NeuronGroup() equation.
- _dyn_neuron_thres_name: Generated variable name for the NeuronGroup() threshold.
- _dyn_neuron_reset_name: Generated variable name for the NeuronGroup() reset value.
- _dyn_neuron_refra_name: Generated variable name for the NeuronGroup() refractory value.
- _dyn_neuron_namespace_name: Generated variable name for the NeuronGroup() namespace.

- **ng_init: NeuronGroups() should be initialized with a random vm, ge and gi values. To address this, a 6-line code** is generated and put in this variable, the running of which will lead to initialization of current NeuronGroup().

relay()

The method that creates the relay NeuronGroups based on the parameters that are extracted from the configuration file in the __init__ method of the class. Note that the SpikeGeneratorGroup() does not support the locations and synaptic connection based on the distance between the input, and the target neuron group. For this reason, a "relay" neuron group is created which is directly connected to the SpikeGeneratorGroup(). Unlike SpikeGeneratorGroup() this relay group supports the locations. With this workaround, the synaptic connection between the input and the Neuron group can be implemented based on the distance of the neurons then.

Note: extracting the input spikes and time sequences for using in a SpikeGeneratorGroup() is done in this method. This procedure needs using a "run()" method in brian2. However, one of the limitations of the Brian2GeNN is that the user cannot use multiple "run()" methods in the whole script. To address this issue, the GeNN device should be set after using the first run(), hence the unusual placement of "set_device('genn')" command in current method.

Note2: The radius of the VPM input is determined based on the Markram et al. 2015: The radius of the system is 210 um and the number of VPM input is 60 (page 19 of supplements). As for the radius of the VPM input, it is mentioned in the paper (page 462) that "neurons were arranged in 310 mini-columns at horizontal positions". considering the area of the circle with radius of 210um and 60/310 mini-columns, the radius will be equal to 92um.

Main internal variables:

- inp: an instance of stimuli() object from stimuli module.
- relay_group: the dictionary containing the data for relay NeuronGroup()
- _dyn_neurongroup_name: Generated variable name for the NeuronGroup() object in brian2.
- _dyn_neuronnumber_name: Generated variable name for corresponding Neuron Number.
- _dyn_neuron_eq_name: Generated variable name for the NeuronGroup() equation.
- _dyn_neuron_thres_name: Generated variable name for the NeuronGroup() threshold.
- _dyn_neuron_reset_name: Generated variable name for the NeuronGroup() reset value.
- sg_syn_name: variable name for the Synapses() object that connects SpikeGeneratorGroup() and relay neurons.

following four variables are build using the load_input_seq() method in stimuli object:

- spikes_str: The string containing the syntax for Spike indices in the input neuron group.
- times_str: The string containing the syntax for time indices in the input neuron group.
- sg_str: The string containing the syntax for creating the SpikeGeneratorGroup() based on the input .mat file.
- number_of_neurons: The number of neurons that exist in the input .mat file.

synapse()

The method that creates the Synapses() in brian2, based on the parameters that are extracted from the configuration file in the __init__ method of the class.

Main internal variables:

- mon_args: contains the monitor arguments extracted from the target line.

- **args: normally args contains a set of arguments for a single Synapses() object.** However, this changes when the post-synaptic neuron is the first (with index of 0) compartment of a multi-compartmental neuron. In this case, one might intend to target all three sub-compartments, i.e. Basal dendrites, Soma and proximal apical dendrites. So the single set of arguments will be changed to 3 sets of arguments and a for loop will take care of every one of them.
- dyn_syn_name: Generated variable name for the Synapses() object in brian2.
- _dyn_syn_eq_name: Generated variable name for the Synapses() equation.
- _dyn_syn_pre_eq_name: Generated variable name for pre_synaptic equations, i.e. "on_pre=…"
- _dyn_syn_post_eq_name: Generated variable name for post_synaptic equations, i.e. "on_post= …"
- _dyn_syn_namespace_name: Generated variable name for the Synapses() namespace.
- **syn_con_str: The string containing the syntax for connect() method of a current Synapses() object. This string** changes depending on using the [p] and [n] tags in the configuration file.

class cxsystem2.core.parameter_parser.**NeuronParser**(*output_neuron*, *physio_config_df*)

This class embeds all parameter sets associated to all neuron types and will return it as a namespace in form of dictionary

__init__(*output_neuron*, *physio_config_df*)

Initialize self. See help(type(self)) for accurate signature.

class cxsystem2.core.parameter_parser.**SynapseParser**(*output_synapse*, *physio_config_df*)

This class contains all the variables that are required for the Synapses() object namespaces. There are several reference dictionaries in this class for:

- cw: connection weights for any connection between NeuronGroup()s.
- sp: Sparseness values for any connection between NeuronGroup()s.
- STDP: values for A_pre, A_post, Tau_pre and Tau_post for any connection between NeuronGroup()s.
- dist: distribution of the neurons for connection between NeuronGroup()s.

There are also some important internal variables:

- Cp: Synaptic potentiation coefficient according to van Rossum J Neurosci 2000
- Cd: Synaptic depression coefficient according to van Rossum J Neurosci 2000
- stdp_Nsweeps: 60 in papers one does multiple trials to reach +-50% change in synapse strength. A-coefficient will be divided by this number
- stdp_max_strength_coefficient: This value is to avoid runaway plasticity.
- conn_prob_gain: This is used for compensation of small number of neurons and thus incoming synapses

__init__(*output_synapse*, *physio_config_df*)

The initialization method for namespaces() object.

Parameters **output_synapse** – This is the dictionary created in NeuronReference() in brian2_obj_namespaces module. This contains all the information about the synaptic connection. In this class, Synaptic namespace parameters are directly added to it. Following values are set after initialization: Cp, Cd, sparseness, spatial_decay. Other variables are then set based on the type of the synaptic connection (STDP,Fixed, etc).

CPlastic()

The CPlastic method for assigning the parameters to the customized_synapses() object.

This contains all the information about the synaptic connection. In this method, STDP parameters are directly added to this variable. Following values are set in this method: Apre, Apost, Tau_pre, Tau_post, wght_max, wght0.

Depressing()
Depressing synapse

Facilitating()
Facilitating synapse

Fixed()
The Fixed method for assigning the parameters for Fixed synaptic connection to the customized_synapses() object.

Fixed_calcium()
The Fixed method for assigning the parameters for Fixed synaptic connection to the customized_synapses() object. This synapse was used in the 1st submitted version, but was later deemed non-valid in terms of calcium scaling

Fixed_const_wght()
The Fixed method with constant weight for assigning the parameters for Fixed synaptic connection to the customized_synapses() object.

Fixed_multiply()
The Fixed multiply method using constant weight but assigning multiplier further on to synaptic weight. This enables array search for loaded connection strengths.

STDP()
The STDP method for assigning the STDP parameters to the customized_synapses() object.

This contains all the information about the synaptic connection. In this method, STDP parameters are directly added to this variable. Following STDP values are set in this method: Apre, Apost, Tau_pre, Tau_post, wght_max, wght0.

STDP_with_scaling()
The STDP method for assigning the STDP parameters to the customized_synapses() object.

scale_by_calcium(*ca*, *cw=None*)
Scales synaptic weight depending on calcium level

Parameters

- **ca** – float, calcium concentration in mM
- **cw** – float, connection weight with calcium level 2.0mM (optional)

Returns float, scaled synaptic weight

class cxsystem2.core.physiology_reference.**NeuronReference**(*idx*, *number_of_neurons*, *cell_type*, *layers_idx*, *general_grid_radius*, *min_distance*, *physio_config_df*, *network_center=0j*, *cell_subtype='–'*)
Using this class, a dictionary object is created which contains all parameters and variables that are needed to create a group of that customized cell. This dictionary will eventually be used in process of building the cortical module. New types of neurons should be implemented in this class.

__init__(*idx*, *number_of_neurons*, *cell_type*, *layers_idx*, *general_grid_radius*, *min_distance*, *physio_config_df*, *network_center=0j*, *cell_subtype='–'*)
initialize the NeuronReference based on the arguments.

Parameters

- **number_of_neurons** – number of neurons in the NeuronGroup() object.
- **cell_type** – type of cell in the NeuronGroup: currently PC, SS, BC, MC and L1i.
- **layers_idx** – indicating the layer in which the cell group is located. In case of SS, BC, MC and L1i it is an integer but for PC which is a multi-compartmental neuron, it is a tuple array. This tuple numpy array defines the first and last layers in which the neuron resides. So np.array([4,1]) means that the soma resides in layer 4 and the apical dendrites which are (2 compartments) extend to layer 2/3 and 1. To avoid confusion, layer 2 is used as the indicator of layer 2/3. Hence, if the last compartment of a neuron is in layer 2/3, use number 2.
- **network_center** – as the name implies, this argument defines the center of teh NeuronGroup() in visual field coordinates. The default value is 0+0j.
- **resolution** – resolution for formation of neurons in the grid. Default value is 0.1

Main internal variables:

- **output_neuron: the main dictionary containing all the data about current Customized_neuron_group including:** reset, refractory, neuron type, soma position(layer), dendrites layer, total number of compartments, namespace, equation, positions (both in cortical and visual coordinates).

BC()
This method build up the equation for BC neurons. The final equation is then saved in output_neuron['equation'].

- The equation of the neuron is as follows:

```
dvm/dt = (gL*(EL-vm) + gL * DeltaT * exp((vm-VT) / DeltaT) + ge_
↪soma * (Ee-vm) + gi_soma * (Ei-vm)) / C : volt (unless refractory)
dge_soma/dt = -ge_soma/tau_e : siemens
dgi_soma/dt = -gi_soma/tau_i : siemens
x : meter
y : meter
```

CI()
This method build up the equation for CI neurons. CI stands for current injection as timed array directly to neuron model. The final equation is then saved in output_neuron['equation'].

- The equation of the neuron is as follows:

```
dvm/dt = (gL*(EL-vm)  + ge_soma * 1 * vm + gi_soma * 1 * vm + I_
↪ext(t,i) / C : volt (unless refractory)
dge_soma/dt = -ge_soma/tau_e : siemens
dgi_soma/dt = -gi_soma/tau_i : siemens
x : meter
y : meter
I_ext : amp
```

L1i()
This method build up the equation for Layer 1 inhibitory (L1i) neurons. The final equation is then saved in output_neuron['equation'].

- The equation of the neuron is as follows:

```
dvm/dt = (gL*(EL-vm) + gL * DeltaT * exp((vm-VT) / DeltaT) + ge_
↪soma * (Ee-vm) + gi_soma * (Ei-vm)) / C : volt (unless refractory)
dge_soma/dt = -ge_soma/tau_e : siemens
dgi_soma/dt = -gi_soma/tau_i : siemens
x : meter
y : meter
```

MC ( )

This method build up the equation for MC neurons. The final equation is then saved in output_neuron['equation'].

- The equation of the neuron is as follows:

```
dvm/dt = (gL*(EL-vm) + gL * DeltaT * exp((vm-VT) / DeltaT)
        + ge_soma * (Ee-vm) + gi_soma * (Ei-vm)) / C : volt (unless␣
↪refractory)
dge_soma/dt = -ge_soma/tau_e : siemens
dgi_soma/dt = -gi_soma/tau_i : siemens
x : meter
y : meter
```

NDNEURON ( )

NDNEURON type, just for testing Neurodynlib

Returns

PC ( )

This method build up the equation for PC neurons based on the number of compartments. The final equation is then saved in output_neuron['equation'].

Main internal variables:

- **eq_template_soma: Contains template somatic equation, the variables in side the equation could be** replaced later using "Equation" function in brian2. :

```
dvm/dt = (gL*(EL-vm) + gealpha * (Ee-vm) + gealphaX * (Ee-vm) +␣
↪gialpha * (Ei-vm)
         + gL * DeltaT * exp((vm-VT) / DeltaT) +I_dendr) / C : volt␣
↪(unless refractory)
dge/dt = -ge/tau_e : siemens
dgealpha/dt = (ge-gealpha)/tau_e : siemens
dgeX/dt = -geX/tau_eX : siemens
dgealphaX/dt = (geX-gealphaX)/tau_eX : siemens
dgi/dt = -gi/tau_i : siemens
dgialpha/dt = (gi-gialpha)/tau_i : siemens
x : meter
y : meter
```

- eq_template_dend: Contains template dendritic equation:

```
dvm/dt = (gL*(EL-vm) + gealpha * (Ee-vm) + gealphaX * (Ee-vm) +␣
↪gialpha * (Ei-vm)
         + I_dendr) / C : volt (unless refractory)
dge/dt = -ge/tau_e : siemens
dgealpha/dt = (ge-gealpha)/tau_e : siemens
dgeX/dt = -geX/tau_eX : siemens
dgealphaX/dt = (geX-gealphaX)/tau_eX : siemens
dgi/dt = -gi/tau_i : siemens
dgialpha/dt = (gi-gialpha)/tau_i : siemens
```

SS()
This method build up the equation for SS neurons. The final equation is then saved in output_neuron['equation'].

- The equation of the neuron is as follows:

```
dvm/dt = (gL*(EL-vm) + gL * DeltaT * exp((vm-VT) / DeltaT) + ge_
↪soma * (Ee-vm) + gi_soma * (Ei-vm)) / C : volt (unless refractory)
dge_soma/dt = -ge_soma/tau_e : siemens
dgi_soma/dt = -gi_soma/tau_i : siemens
x : meter
y : meter
```

VPM()
This method build up the equation for VPM neurons. No equation is needed.

class cxsystem2.core.physiology_reference.**SynapseReference**(*receptor*, *pre_group_idx*, *post_group_idx*, *syn_type*, *pre_type*, *post_type*, *physio_config_df*, *post_comp_name='_soma'*, *custom_weight='–'*, *multiply_weight=1*)
In this class, a dictionary object is created which contains all parameters and variables that are needed to create a Synapses() object between two neuron group. This dictionary will eventually be used in process of building the cortical module. New types of synapses should be implemented in this class.

__init__(*receptor*, *pre_group_idx*, *post_group_idx*, *syn_type*, *pre_type*, *post_type*, *physio_config_df*, *post_comp_name='_soma'*, *custom_weight='–'*, *multiply_weight=1*)
initializes the SynapseReference based on its arguments.

Parameters

- **receptor** – defines the type of the receptor in the synaptic connection. Currently ge and gi are implemented.
- **pre_group_idx** – The index of the pre-synaptic group.
- **post_group_idx** – The index of the post-synaptic group.
- **syn_type** – Type of the synaptic connection, currently STDP and Fixed are implemented.
- **pre_type** – Type of the pre-synaptic NeuronGroup.
- **post_type** – Type of the post-synaptic NeuronGroup.
- **post_comp_name** – Name of the target compartment in the cells of the post-synaptic NeuronGroup. The default value is "_soma" as usually soma is being targeted. In case other compartments are targeted in a PC cell, e.g. basal or apical dendrites, _basal or _apical will be used.

Main internal variables:

- **output_synapse: the main dictionary containing all the data about current customized_synapse_group including** (model, pre, post), type of synapse, type of receptor, index and type of pre- and post-synaptic group, namespace for the Synapses() object, sparseness, spatial_decay.
- _name_space: An instance of brian2_obj_namespaces() object which contains all the constant parameters for this synaptic equation.

CPlastic()
The method for implementing the plastic synaptic connection according to Clopath_2010_NatNeurosci.

Depressing()
Depressing non-stochastic Tsodyks-Markram synapse

Facilitating()
Facilitating non-stochastic Tsodyks-Markram synapse

Fixed()
The method for implementing the Fixed synaptic connection.

Fixed_calcium()
The method for implementing the Fixed synaptic connection.

Fixed_const_wght()
The method for implementing the Fixed synaptic connection.

Fixed_multiply()
The method for implementing the Fixed synaptic connection which is multiplied with factor coming from Anatomy csv.

STDP()
The method for implementing the STDP synaptic connection.

STDP_with_scaling()
The method for implementing the STDP synaptic connection.

class cxsystem2.core.workspace_manager.**Workspace**(*workspace_path*, *suffix*)
As the name implies, this module is used for gathering the data and saving the result.

__init__(*workspace_path*, *suffix*)
Initializes the save_data object.

Parameters

- **save_path** – The path for saving the data.
- **suffix** – the string containing date and time that is supposed to be unique for each simulation and is used as a suffix for file names.

Main internal variables:

- data: the main variable to be saved. It contains all the data about the positions of the NeuronGroup()s as well as the monitor results.
- **syntax_bank: Since the monitors are explicitly defined in the Globals(), extracting the data from them requires a** explicitely. To automatize this process, the syntaxes for extracting the data from the target monitors are generated and saved in this variable, so that they can be run at the end of the simulation.

create_connections_key(*key*)
In case the user wants to save a peculiar variable, this method can be used to check and create a new key in data dictionary (if does not exist).

Parameters **key** – name of the key to be created in the final data variable.

create_results_key(*key*)
In case the user wants to save a peculiar variable, this method can be used to check and create a new key in data dictionary (if does not exist).

Parameters **key** – name of the key to be created in the final data variable.

class cxsystem2.core.equation_templates.**EquationHelper**(*neuron_model='EIF'*, *is_pyramidal=False*, *compartment='soma'*, *exc_model='SIMPLE_E'*, *inh_model='SIMPLE_I'*, *custom_strings=None*)

Helper class for switching swiftly between neuron/receptor models in CxSystem. Currently used only for the pyramidal cell (PC) cell type; point neuron models have been migrated to neurodynlib.

__init__(*neuron_model='EIF'*, *is_pyramidal=False*, *compartment='soma'*, *exc_model='SIMPLE_E'*, *inh_model='SIMPLE_I'*, *custom_strings=None*)

Initialize self. See help(type(self)) for accurate signature.

class cxsystem2.core.stimuli.**Stimuli**(*duration*, *input_mat_path*, *output_folder*, *output_file_extension*, *output_file_suffix=''*)

[Extracted from VCXmodel] This is the stimulation object for applying the input to a particular NeuronGroup(). Currently only video input is supported.

__init__(*duration*, *input_mat_path*, *output_folder*, *output_file_extension*, *output_file_suffix=''*)

Initializes the input module for and instance of CxSystem.

Parameters

- **duration** –
- **input_mat_path** – .mat file location
- **output_folder** – location of the saved output
- **output_file_suffix** – suffix for the output file
- **output_file_extension** – extension for the output file, i.e. gz, bz2, pickle

calculate_input_seqs()

Calculating input sequence based on the video input.

generate_inputs(*freq*)

The method for generating input based on the .mat file, using the internal _initialize_inputs() and _calculate_input_seqs() methods.

Parameters **freq** – frequency.

get_input_positions()

Extract the positions from the .mat file.

load_input_seq(*input_spike_file_location*)

Loads spikes from file.

Parameters **input_spike_file_location** – Location of the file to load spikes.

cxsystem2.core.tools.**write_to_file**(*save_path*, *data*)

cxsystem2.core.tools.**load_from_file**(*load_path*)

cxsystem2.core.tools.**parameter_finder**(*df*, *keyword*)

cxsystem2.core.tools.**change_anat_file_header_value**(*filepath*, *save_path*, *parameter*, *new_value*)

cxsystem2.core.tools.**read_config_file**(*conf*, *header=False*)

This function reads the file and convert it to csv from json if necessary. It only works by loading the csv without headers. (header=none) If you need the first row as header, do it manually :param header: :param conf: :return:

5.2 neurodynlib module

class cxsystem2.neurodynlib.multicompartment_models.**LeakyCompartment**
Not implemented! A leaky capacitor to be used as a template for pyramidal cell compartments.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

class cxsystem2.neurodynlib.multicompartment_models.**MulticompartmentNeuron**
Not implemented! A base class for multicompartmental neuron models.

__init__()
Initialize self. See help(type(self)) for accurate signature.

class cxsystem2.neurodynlib.multicompartment_models.**LegacyPyramidalCell**(*n_apical=3*)
Not implemented! The kind of pyramidal cell used in Heikkinen et al. 2015 J Neurosci, Andalibi et al. 2019 Neural Computation, and Hokkanen et al. 2019 Neural Computation. See cxsystem2.core.equation_templates for the current implementation.

__init__(*n_apical=3*)
Initialize self. See help(type(self)) for accurate signature.

class cxsystem2.neurodynlib.neuron_models.**PointNeuron**(*is_pyramidal=False*, *compartment='soma'*)
Base class for point neurons

__init__(*is_pyramidal=False*, *compartment='soma'*)
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

add_external_current(*current_name='I_ext'*, *current_eqs=None*)
Adds an external current to the neuron

Parameters

- **current_name** (*string*) – name of the current
- **current_eqs** (*string*) – equations describing the external current

Returns

add_model_definition(*key*, *string_to_add*)
Append a string to a template placeholder.

Parameters

- **key** (*string*) – placeholder name
- **string_to_add** – string to append to the placeholder

add_tonic_current(*tonic_current=50. * pamp*, *tau_rampup=None*)
Adds tonic current injection to the neuron

Parameters

- **tonic_current** – amplitude of current injection (in amps)
- **tau_rampup** – time constant for current ramp-up (in milliseconds)

add_vm_noise(*noise_sigma=2. * mvolt*)
Adds a stochastic component to the membrane equation as explained in Brian2 documentation

Parameters **noise_sigma** –

Returns

get_compartment_equations(*compartment_name*)
Compiles the membrane equation and adds compartment name to all compartment-specific variables

Parameters **compartment_name** (*string*) – name of the compartment

Returns string

get_initial_values()
Get initial values. If the initial value of vm is None, will replace it with EL (reversal potential of leak).

Returns

get_json(*include_neuron_name=True*)
Creates a JSON string of parameter names and values (units are discarded).

Parameters **include_neuron_name** (*bool*) – whether to include the neuron name

Returns string

get_membrane_equation(*substitute_ad_hoc=None*, *return_string=True*)
Compiles the membrane equation from the template for use in Brian2. This should be the only function where the template equation is used.

Parameters

- **substitute_ad_hoc** (*dict*) – dictionary of temporary values to use in the equation template
- **return_string** (*bool*) – If True, returns equations as a string. Otherwise, returns a b2.Equations object.

Returns string (or b2.Equations object)

get_neuron_equations()
Returns the membrane equations in a form that prints out nicely in Jupyter Notebook

Returns b2.Equations object

get_neuron_parameters()
Shows all the current parameter names and values

Returns dict

get_parameter_names()
Shows all the parameter names that can/must be defined

Returns list

get_refractory_period()
Get the refractory period (after a spike).

Returns duration (typically in ms)

get_reset_statements()
Get the statements that will be executed once the neuron hits the threshold.

Returns string

get_states_to_monitor()
Get state variables to monitor (for method plot_states())

Returns list

get_threshold_condition()
Get the spike threshold condition.

Returns string

getting_started(*step_amplitude=1.2 * namp*, *sine_amplitude=2.5 * namp*, *sine_freq=150. * hertz*, *sine_dc=2. * namp*)
Simple example that stimulates the neuron with a step and a sinusoidal current.

Parameters

- **step_amplitude** – step current amplitude (in amps)
- **sine_amplitude** – sine current amplitude (in amps)
- **sine_freq** – sine current frequency (in Hz)
- **sine_dc** – constant current to inject during the sine stimulation (in amps)

Returns

list_neurons_in_json(*filename*)
List neuron types (sets of parameters) in a JSON file

Parameters **filename** (*string*) – Path to file

Returns list

make_neuron_group(*n*)
Makes a Brian2 NeuronGroup

Parameters **n** (*int*) – number of neurons

Returns b2.NeuronGroup object

plot_fi_curve(*min_current=0. * amp*, *max_current=1. * namp*, *step_size=10. * pamp*, *plot=True*, *max_rate=None*, *save_name=None*)
Plot the frequency-current (f-I) curve.

Parameters

- **min_current** – minimum current (in amps)
- **max_current** – maximum current (in amps)
- **step_size** – current step (in amps)
- **plot** (*bool*) – whether to plot the results or not
- **max_rate** – maximum frequency to show in the plot

Returns steps, counts (if plot is False)

plot_states(*state_monitor*, *parameters=None*)
Plots pre-defined state variables from a state monitor

Parameters

- **state_monitor** – b2.StateMonitor

- **parameters** – list of parameters

plot_vm(*state_monitor*)
Plots the vm from a state monitor

Parameters **state_monitor** – b2.StateMonitor with vm recording

read_json(*filename*, *neuron_name=None*)
Read and load parameters from a JSON file.

Parameters

- **filename** (*string*) – Path to file
- **neuron_name** (*string*) – the name of the neuron to read from the file

save_json(*filename=None*)
Saves the neuron parameters in a JSON file.

Parameters **filename** (*string*) – Path to file. If None, will save as neuron_name.json

set_excitatory_receptors(*receptor_name*)
Sets the excitatory receptors.

Parameters **receptor_name** (*string*) – name of receptor model (see neurodynlib.receptor_models)

set_inhibitory_receptors(*receptor_name*)
Sets the inhibitory receptors.

Parameters **receptor_name** (*string*) – name of receptor model (see neurodynlib.receptor_models)

set_model_definition(*key*, *string_to_set*)
Set the value of a template placeholder.

Parameters

- **key** (*string*) – placeholder name
- **string_to_set** (*string*) – placeholder value

set_neuron_parameters(***kwargs*)
Set neuron parameters. If you don't know the correct units, use get_neuron_parameters() first to get the default parameters with correct units.

Parameters **kwargs** – new parameter values are given as arguments

simulate_neuron(*I_stim=<brian2.input.timedarray.TimedArray object>*, *simulation_time=1. * second*, ***kwargs*)
Simulate/stimulate the neuron

Parameters

- **I_stim** – input stimulus (use the input_factory to create the stimulus)
- **simulation_time** – duration (usually in milliseconds, eg. 3000*ms)
- **kwargs** – custom neuron parameters can be given as arguments

Returns b2.StateMonitor, b2.SpikeMonitor

what_is_this()
Method to query for the URL describing the neuron model

Returns url

class cxsystem2.neurodynlib.neuron_models.**LifNeuron**
Leaky Intergrate-and-Fire (LIF) model. See Neuronal Dynamics, Chapter 1 Section 3.

Requires setting the following parameters: EL, gL, C, V_res, VT.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

class cxsystem2.neurodynlib.neuron_models.**EifNeuron**
Exponential Integrate-and-Fire (EIF) model. See Neuronal Dynamics, Chapter 5 Section 2.

Requires setting the following parameters: EL, gL, C, V_res, VT, DeltaT, Vcut.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

getting_started(*step_amplitude=0.8 * namp*, *sine_amplitude=1.6 * namp*, *sine_freq=150. * hertz*, *sine_dc=1.3 * namp*)
Simple example that stimulates the neuron with a step and a sinusoidal current.

Parameters

- **step_amplitude** – step current amplitude (in amps)
- **sine_amplitude** – sine current amplitude (in amps)
- **sine_freq** – sine current frequency (in Hz)
- **sine_dc** – constant current to inject during the sine stimulation (in amps)

Returns

class cxsystem2.neurodynlib.neuron_models.**AdexNeuron**
Adaptive Exponential Integrate-and-Fire (ADEX) model. See Neuronal Dynamics, Chapter 6 Section 1.

Requires setting the following parameters: EL, gL, C, V_res, VT, DeltaT, Vcut, a, b, tau_w.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

getting_started(*step_amplitude=65. * pamp*, *sine_amplitude=125. * pamp*, *sine_freq=150. * hertz*, *sine_dc=100. * pamp*)
Simple example that stimulates the neuron with a step and a sinusoidal current.

Parameters

- **step_amplitude** – step current amplitude (in amps)

- **sine_amplitude** – sine current amplitude (in amps)
- **sine_freq** – sine current frequency (in Hz)
- **sine_dc** – constant current to inject during the sine stimulation (in amps)

Returns

plot_states(*state_monitor*, *spike_monitor=None*, *save_name=None*)
Visualizes the state variables: w-t, vm-t and phase-plane w-vm

Parameters **state_monitor** – b2.StateMonitor

class cxsystem2.neurodynlib.neuron_models.**HodgkinHuxleyNeuron**
Implementation of a Hodgkin-Huxley neuron with Na, K and leak channels (SIMPLE_HH). See Neuronal Dynamics, Chapter 2 Section 2

Requires setting the following parameters: EL, gL, C, EK, ENa, gK, gNa, V_spike.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

getting_started(*step_amplitude=7.2 * uamp*, *sine_amplitude=3.6 * uamp*, *sine_freq=150. * hertz*, *sine_dc=2.9 * namp*)
Simple example that stimulates the neuron with a step and a sinusoidal current.

Parameters

- **step_amplitude** – step current amplitude (in amps)
- **sine_amplitude** – sine current amplitude (in amps)
- **sine_freq** – sine current frequency (in Hz)
- **sine_dc** – constant current to inject during the sine stimulation (in amps)

Returns

plot_states(*state_monitor*)
Plots the state variables vm, m, n, h vs. time.

Parameters **state_monitor** – b2.StateMonitor

class cxsystem2.neurodynlib.neuron_models.**IzhikevichNeuron**
Izhikevich model (IZHIKEVICH). See Neuronal Dynamics, Chapter 6 Section 1

Here, we use the formulation and parameters presented in Izhikevich & Edelman 2008 PNAS.

Requires setting the following parameters: EL, C, V_res, VT, k, a, b, d, Vcut.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

plot_states(*state_monitor*)
Visualizes the state variables: u-t, vm-t and phase-plane u-vm

Parameters **state_monitor** – b2.StateMonitor

class cxsystem2.neurodynlib.neuron_models.**LifAscNeuron**
Leaky Integrate-and-Fire with After-spike Currents (LIFASC). One of the generalized LIF (GLIF_3) models used in the Allen Brain Institute.

For more information, see http://celltypes.brain-map.org/ , http://help.brain-map.org/display/celltypes/Documentation?_ga=2.31556414.1221863260.1571652272-1994599725.1571652272 , or Teeter et al. 2018 Nature Comm. https://www.nature.com/articles/s41467-017-02717-4.

Requires setting the following parameters: EL, gL, C, V_res, VT, A_asc1, A_asc2, tau_asc1, tau_asc2.

__init__()
Initializes the point neuron object

Parameters

- **is_pyramidal** (*bool*) – whether the point neuron model is part of a pyramidal/multicompartmental cell
- **compartment** (*string*) – name of the compartment (eg. soma, basal, a3)

read_abi_neuron_config(*neuron_config*)
Method for importing parameters from the Allen Brain Institute's cell type atlas. Parameters can be obtained by downloading the json from their website.

You can also use the AllenSDK:

```
from allensdk.api.queries.glif_api import GlifApi
neuron_config = GlifApi().get_neuron_configs([neuronal_model_id])[neuronal_
↪model_id]
```

Parameters **neuron_config** –

Returns

class cxsystem2.neurodynlib.neuron_models.**neuron_factory**

__init__()
Initialize self. See help(type(self)) for accurate signature.

class cxsystem2.neurodynlib.receptor_models.**ReceptorModel**(*receptor_model*)

__init__(*receptor_model*)
Initialize self. See help(type(self)) for accurate signature.

cxsystem2.neurodynlib.tools.input_factory.**get_step_current**(*t_start*, *t_end*, *unit_time*, *amplitude*, *append_zero=True*)
Creates a step current. If t_start == t_end, then a single entry in the values array is set to amplitude.

Parameters

- **t_start** (*int*) – start of the step
- **t_end** (*int*) – end of the step
- **unit_time** – unit of t_start and t_end. e.g. 0.1*brian2.ms

- **amplitude** – amplitude of the step. e.g. 3.5*brian2.uamp
- **append_zero** – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array (=amplitude) for all indices > t_end.

Returns Brian2.TimedArray

cxsystem2.neurodynlib.tools.input_factory.**get_ramp_current**(*t_start*, *t_end*, *unit_time*, *amplitude_start*, *amplitude_end*, *append_zero=True*)

Creates a ramp current. If t_start == t_end, then ALL entries are 0.

Parameters

- **t_start** (*int*) – start of the ramp
- **t_end** (*int*) – end of the ramp
- **unit_time** – unit of t_start and t_end. e.g. 0.1*ms
- **amplitude_start** – amplitude of the ramp at t_start. e.g. 3.5*uamp
- **amplitude_end** – amplitude of the ramp at t_end. e.g. 4.5*uamp
- **append_zero** (*bool*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array (=amplitude_end) for all indices > t_end.

Returns Brian2.TimedArray

cxsystem2.neurodynlib.tools.input_factory.**get_sinusoidal_current**(*t_start*, *t_end*, *unit_time*, *amplitude*, *frequency*, *direct_current*, *phase_offset=0.0*, *append_zero=True*)

Creates a sinusoidal current. If t_start == t_end, then ALL entries are 0.

Parameters

- **t_start** (*int*) – start of the sine wave
- **t_end** (*int*) – end of the sine wave
- **unit_time** – unit of t_start and t_end. e.g. 0.1*ms
- **amplitude** – maximum amplitude of the sinus e.g. 3.0*uamp
- **frequency** – Frequency of the sine. e.g. 0.5*kHz
- **direct_current** – DC-component (=offset) of the current, e.g. 1.5*uamp
- **phase_offset** (*float*) – phase at t_start. Default = 0
- **append_zero** (*bool*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array for all indices > t_end.

Returns Brian2.TimedArray

cxsystem2.neurodynlib.tools.input_factory.**get_zero_current**()

Returns a TimedArray with one entry: 0 Amp

Returns: TimedArray

cxsystem2.neurodynlib.tools.input_factory.**get_spikes_current**(*t_spikes*, *unit_time*, *amplitude*, *append_zero=True*)

Creates a two dimensional TimedArray wich has one column for each value in t_spikes. All values in each column are 0 except one, the spike time as specified in t_spikes is set to amplitude. Note: This function is provided to easily insert pulse currents into a cable. For other use of spike input, search the Brian2 documentation for SpikeGeneration.

Parameters

- **t_spikes** (*int*) – list of spike times
- **unit_time** – unit of t_spikes . e.g. 1*ms
- **amplitude** – amplitude of the spike. All spikes have the same amplitude
- **append_zero** (*bool*) – if true, 0Amp is appended at t_end+1. Without that trailing 0, Brian reads out the last value in the array for all indices > t_end.

Returns Brian2.TimedArray

cxsystem2.neurodynlib.tools.input_factory.**plot_step_current_example**()

Example for get_step_current.

cxsystem2.neurodynlib.tools.input_factory.**plot_ramp_current_example**()

Example for get_ramp_current

cxsystem2.neurodynlib.tools.input_factory.**plot_sinusoidal_current_example**()

Example for get_sinusoidal_current

cxsystem2.neurodynlib.tools.input_factory.**getting_started**()

cxsystem2.neurodynlib.tools.plot_tools.**plot_voltage_and_current_traces**(*voltage_monitor*, *current*, *title=None*, *firing_threshold=None*, *legend_location=0*, *save_name=None*)

Not implemented! plots voltage and current .

Args: voltage_monitor (StateMonitor): recorded voltage current (TimedArray): injected current title (string, optional): title of the figure firing_threshold (Quantity, optional): if set to a value, the firing threshold is plotted. legend_location (int): legend location. default = 0 (="best") save_name (string, optional): if set, the figure is saved to this file name

Returns: the figure

cxsystem2.neurodynlib.tools.plot_tools.**plot_network_activity**(*rate_monitor*, *spike_monitor*, *voltage_monitor=None*, *spike_train_idx_list=None*, *t_min=None*, *t_max=None*, *N_highlighted_spiketrains=3*, *avg_window_width=None*, *sup_title=None*, *figure_size=(10, 4)*)

Not implemented! Visualizes the results of a network simulation: spike-train, population activity and voltage-traces.

Args: rate_monitor (PopulationRateMonitor): rate of the population spike_monitor (SpikeMonitor): spike trains of individual neurons voltage_monitor (StateMonitor): optional. voltage traces of some (same as in spike_train_idx_list) neurons spike_train_idx_list (list): optional. A list of neuron indices whose spike-train is plotted. If no list is provided, all (up to 500) spike-trains in the spike_monitor are plotted. If None, the the list in voltage_monitor.record is used. t_min (Quantity): optional. lower bound of the plotted time interval. if t_min is None, it is set to the larger of [0ms, (t_max - 100ms)] t_max (Quantity): optional. upper bound of the plotted time interval. if t_max is None, it is set to the timestamp of the last spike in N_highlighted_spiketrains (int): optional. Number of spike trains visually highlighted, defaults to 3 If N_highlighted_spiketrains==0 and voltage_monitor is not None, then all voltage traces of the voltage_monitor are plotted. Otherwise N_highlighted_spiketrains voltage traces are plotted. avg_window_width (Quantity): optional. Before plotting the population rate (PopulationRateMonitor), the rate is smoothed using a window of width = avg_window_width. Defaults is 1.0ms sup_title (String): figure suptitle. Default is None. figure_size (tuple): (width,height) tuple passed to pyplot's figsize parameter.

Returns: Figure: The whole figure Axes: Top panel, Raster plot Axes: Middle panel, population activity Axes: Bottom panel, voltage traces. None if no voltage monitor is provided.

cxsystem2.neurodynlib.tools.plot_tools.**plot_ISI_distribution**(*spike_stats*, *hist_nr_bins=50*, *xlim_max_ISI=None*)

Not implemented! Computes the ISI distribution of the given spike_monitor and displays the distribution in a histogram

Args: spike_stats (neurodynex.tools.spike_tools.PopulationSpikeStats): statistics of a population activity hist_nr_bins (int): Number of histrogram bins. Default:50 xlim_max_ISI (Quantity): Default: None. In not None, the upper xlim of the plot is set to xlim_max_ISI. The CV does not change if this bound is set.

Returns: the figure

cxsystem2.neurodynlib.tools.plot_tools.**plot_spike_train_power_spectrum**(*freq*, *mean_ps*, *all_ps*, *max_freq*, *nr_highlighted_neurons=2*, *mean_firing_freqs_per_neur*
plot_f0=False)

Not implemented! Visualizes the power spectrum of the spike trains.

Args: freq: frequencies (= x axis) mean_ps: average power taken over all neurons (typically all of a subsample). all_ps (dict): power spectra for each single neuron max_freq (Quantity): The x-lim of the plot is [-0.05*max_freq, max_freq] mean_firing_freqs_per_neuron (float): None or the mean firing rate averaged

across the neurons. Default is None in which case the value is not shown in the legend plot_f0 (bool): if true, the power at frequency 0 is plotted. Default is False and the value is not plotted.

Returns: the figure and the index of the random neuron for which the PS is computed: all_ps[random_neuron_index]

cxsystem2.neurodynlib.tools.plot_tools.**plot_population_activity_power_spectrum**(*freq*, *ps*, *max_freq*, *average_At=None*, *plot_f0=False*)

Not implemented! Plots the power spectrum of the population activity A(t)

Args: freq: frequencies (= x axis) ps: power spectrum of the population activity max_freq (Quantity): The data is plotted in the interval [-.05*max_freq, max_freq] plot_f0 (bool): if true, the power at frequency 0 is plotted. Default is False and the value is not plotted.

Returns: the figure

class cxsystem2.neurodynlib.tools.spike_tools.**PopulationSpikeStats**(*nr_neurons*, *nr_spikes*, *all_ISI*, *filtered_spike_trains*)

Not implemented! Wraps a few spike-train related properties.

__init__(*nr_neurons*, *nr_spikes*, *all_ISI*, *filtered_spike_trains*)

Args: nr_neurons: nr_spikes: mean_isi: std_isi: all_ISI: list of ISI values (can be used to plot a histrogram) filtered_spike_trains the spike trains used to compute the stats. It's a time-window filtered copy of the original spike_monitor.all_spike_trains.

Returns: An instance of PopulationSpikeStats

CV
Coefficient of Variation

all_ISI
all ISIs in no specific order

filtered_spike_trains
a time-window filtered copy of the original spike_monitor.all_spike_trains

mean_isi
Mean Inter Spike Interval

nr_neurons
Number of neurons in the original population

nr_spikes
Nr of spikes

std_isi
Standard deviation of the ISI

cxsystem2.neurodynlib.tools.spike_tools.**get_spike_time**(*voltage_monitor*, *spike_threshold*)

Not implemented! Detects the spike times in the voltage. Here, the spike time is DEFINED as the value in voltage_monitor.t for which voltage_monitor.v[idx] is above threshold AND voltage_monitor.v[idx-1] is below threshold (crossing from below). Note: currently only the spike times of the first column in voltage_monitor are detected. Matrix-like monitors are not supported.

Args: voltage_monitor (StateMonitor): A state monitor with at least the fields "v: and "t" spike_threshold (Quantity): The spike threshold voltage. e.g. -50*b2.mV

Returns: A list of spike times (Quantity)

cxsystem2.neurodynlib.tools.spike_tools.**get_spike_stats**(*voltage_monitor*, *spike_threshold*)

Not implemented! Detects spike times and computes ISI, mean ISI and firing frequency. Here, the spike time is DEFINED as the value in voltage_monitor.t for which voltage_monitor.v[idx] is above threshold AND voltage_monitor.v[idx-1] is below threshold (crossing from below). Note: meanISI and firing frequency are set to numpy.nan if less than two spikes are detected Note: currently only the spike times of the first column in voltage_monitor are detected. Matrix-like monitors are not supported.

Args: voltage_monitor (StateMonitor): A state monitor with at least the fields "v: and "t" spike_threshold (Quantity): The spike threshold voltage. e.g. -50*b2.mV

Returns: tuple: (nr_of_spikes, spike_times, isi, mean_isi, spike_rate)

cxsystem2.neurodynlib.tools.spike_tools.**pretty_print_spike_train_stats**(*voltage_monitor*, *spike_threshold*)

Not implemented! Computes and returns the same values as get_spike_stats. Additionally prints these values to the console.

Args: voltage_monitor: spike_threshold:

Returns: tuple: (nr_of_spikes, spike_times, isi, mean_isi, spike_rate)

cxsystem2.neurodynlib.tools.spike_tools.**filter_spike_trains**(*spike_trains*, *window_t_min=0. * second*, *window_t_max=None*, *idx_subset=None*)

Not implemented! creates a new dictionary neuron_idx=>spike_times where all spike_times are in the

half open interval [window_t_min,window_t_max)

Args: spike_trains (dict): a dictionary of spike trains. Typically obtained by calling spike_monitor.spike_trains() window_t_min (Quantity): Lower bound of the time window: t>=window_t_min. Default is 0ms. window_t_max (Quantity): Upper bound of the time window: t<window_t_max. Default is None, in which case no upper bound is set. idx_subset (list, optional): a list of neuron indexes (dict keys) specifying a subset of neurons. Neurons NOT in the key list are NOT added to the resulting dictionary. Default is None, in which case all neurons are added to the resulting list.

Returns: a filtered copy of spike_trains

cxsystem2.neurodynlib.tools.spike_tools.**get_spike_train_stats**(*spike_monitor*, *window_t_min=0. * second*, *window_t_max=None*)

Not implemented! Analyses the spike monitor and returns a PopulationSpikeStats instance.

Args: spike_monitor (SpikeMonitor): Brian2 spike monitor window_t_min (Quantity): Lower bound of the time window: t>=window_t_min. The stats are computed for spikes within the time window. Default is 0ms window_t_max (Quantity): Upper bound of the time window: t<window_t_max. The stats are computed for spikes within the time window. Default is None, in which case no upper bound is set.

Returns: PopulationSpikeStats

cxsystem2.neurodynlib.tools.spike_tools.**_spike_train_2_binary_vector**(*spike_train*, *vector_length*, *discretization_dt*)

Not implemented! Convert the time-stamps of the spike_train into a binary vector of the given length. Note: if more than one spike fall into the same time bin, only one is counted, surplus spikes are ignored.

Args: spike_train: vector_length: discretization_dt:

Returns: Discretized spike train: a fixed-length, binary vector.

cxsystem2.neurodynlib.tools.spike_tools.**_get_spike_train_power_spectrum**(*spike_train*, *delta_t*, *subtract_mean=False*)

cxsystem2.neurodynlib.tools.spike_tools.**get_averaged_single_neuron_power_spectrum**(*spike_monito*
sampling_freque
window_t_min,
window_t_max,
*nr_neurons_*
subtract_mean=

Not implemented! averaged power-spectrum of spike trains in the time window [window_t_min, window_t_max).

The power spectrum of every single neuron's spike train is computed. Then the average across all single-neuron powers is computed. In order to limit the compuation time, the number of neurons taken to compute the average is limited to nr_neurons_average which defaults to 100

Args: spike_monitor (SpikeMonitor) : Brian2 SpikeMonitor sampling_frequency (Quantity): sampling frequency used to discretize the spike trains. window_t_min (Quantity): Lower bound of the time window: t>=window_t_min. Spikes before window_t_min are not taken into account (set a lower bound if you want to exclude an initial transient in the population activity) window_t_max (Quantity): Upper bound of the time window: t<window_t_max. nr_neurons_average (int): Number of neurons over which the average is taken. subtract_mean (bool): If true, the mean value of the signal is subtracted before FFT. Default is False

Returns: freq, mean_ps, all_ps_dict, mean_firing_rate, mean_firing_freqs_per_neuron

cxsystem2.neurodynlib.tools.spike_tools.**get_population_activity_power_spectrum**(*rate_monitor*, *delta_f*, *k_repetitions*, *T_init=100.* * *msecond*, *subtract_mean_activ*

Not implemented! Computes the power spectrum of the population activity A(t) (=rate_monitor.rate)

Args: rate_monitor (RateMonitor): Brian2 rate monitor. rate_monitor.rate is the signal being analysed here. The temporal resolution is read from rate_monitor.clock.dt delta_f (Quantity): The desired frequency res-

olution. k_repetitions (int): The data rate_monitor.rate is split into k_repetitions which are FFT'd independently and then averaged in frequency domain. T_init (Quantity): Rates in the time interval [0, T_init] are removed before doing the Fourier transform. Use this parameter to ignore the initial transient signals of the simulation. subtract_mean_activity (bool): If true, the mean value of the signal is subtracted. Default is False

Returns: freqs, ps, average_population_rate

5.3 Configuration module

class cxsystem2.configuration.config_file_converter.**ConfigConverter**(*input_data*)

__init__(*input_data*)
Initialize self. See help(type(self)) for accurate signature.

5.4 BUI module

class cxsystem2.bui.bui.**RunServer**(*ssl=False*, *port=None*, *nobrowser=False*)

__init__(*ssl=False*, *port=None*, *nobrowser=False*)
Initialize self. See help(type(self)) for accurate signature.

5.5 visualization module

class cxsystem2.visualization.spikedata_to_csvs.**SpikeData**(*filename*)

__init__(*filename*)
Initialize self. See help(type(self)) for accurate signature.

get_positions_list(*xy_multiplier=0.03*, *z_multiplier=0.03*, *return_subsets=True*)
Creates a N_neurons x 3 matrix of neurons, where N_neurons = number of neurons, column 0 = x position, column 1 = y position, and column 2 = z position. Neuron index implicitly encoded by row number (as expected by ViSimpl/StackViz). XY-multipliers set here for visually pleasing results in ViSimpl.

Parameters

- **xy_multiplier** – scalar, how much to scale x-y coordinates
- **xy_multiplier** – scalar, how much to scale z coordinate
- **return_subsets** – True/False (default x),

Returns

get_spike_data()
Reads data from a CxSystem results file

Returns dict

get_spike_list()
Creates an N_spikes x 2 matrix of spikes, where N_spikes = number of spikes, column 0 = neuron index, and column 1 = spike time. Spike times saved in seconds since this is the default unit in ViSimpl.

Returns numpy array

class cxsystem2.visualization.rasterplot_to_pdf.**rasterplot_pdf_generator**(*workspace_path*, *timestamp*, *sampling_rate*)

__init__(*workspace_path*, *timestamp*, *sampling_rate*)
Initialize self. See help(type(self)) for accurate signature.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

V

W