snnet 0.1.0 user's guide

Robert A McDougal

June 1, 2011

Contents

1 Introduction 1

2 Installation 3

2.1 Integrators . . . 3

2.1.1 fast_xpp_integrator . . . 3

2.1.2 xpp_integrator . . . 4

2.1.3 matlab_integrator . . . 4

2.1.4 matlab_mex_integrator . . . 4

2.2 Optional additional software . . . 4

3 Defining the simulation 7

3.1 Model equations . . . 7

3.1.1 Comments . . . 8

3.1.2 Differential equations . . . 8

3.1.3 Algebraic expressions . . . 8

3.1.4 Functions . . . 9

3.1.5 Variable name rules . . . 9

3.1.6 Numbers and probability distributions . . . 10

3.1.7 Current time 10
3.1.8 Special considerations 10
3.1.9 Converting models from xppaut 12
3.1.10 Example: Hodgkin-Huxley equations 12
3.1.11 Example: Fast-Spiking Interneuron (Golomb et al 2007) 14
3.2 Simulation protocol 15
3.2.1 Collections of cells 16
3.2.2 Global dynamics 19
3.2.3 Synapses 19
3.2.4 Noise 22
3.2.5 Parameters and other algebraic expressions 24
3.2.6 Organelles 27
3.2.7 Integration 28
3.2.8 Forking 29
3.3 Example: One Hodgkin-Huxley Cell 30
3.4 Example: Wagner et al, 2004 31

4 Running simulations **35**

4.1 Basic usage 35
4.1.1 Save types 36
4.1.2 Repeated runs 36
4.1.3 Parallelization 37
4.1.4 Integrators 37

4.2 Parameter sweeps 39

4.2.1 Modifying previous simulation parameters 39

4.3 Advanced 40

4.3.1 Temporary files 40

4.3.2 Continuing after unsuccessful integration 41

5 Analyzing model results 43

5.1 Saving data 43

5.2 Getting data 43

5.2.1 During a simulation 44

5.2.2 Immediately after a single simulation 44

5.2.3 By loading saved data 44

5.3 Working with data 44

5.3.1 Individual simulations 45

5.3.2 Groups of simulations 46

5.4 Graphics 46

5.4.1 General plotting 46

5.4.2 Raster plots 47

6 Troubleshooting and Support 49

6.1 Online discussion group 49

6.2 Interactive graphics in matplotlib 49

6.3 More helpful error messages 49

6.4 Compilation errors with fast_xpp_integrator 50

6.4.1 64-bit xppaut 50

6.4.2 Cross compiling 50

6.5 Unable to run xpp_integrator or fast_xpp_integrator 50

A snnet API **51**

A.1 Analysis 51

A.2 Control 54

A.3 Files 58

A.4 Graphics 61

A.5 Noise 62

A.6 Miscellaneous 63

B Data API **65**

B.1 Methods and Properties 65

Chapter 1

Introduction

François de la Rouchefoucauld observed that "the only thing constant in life is change." [?] Indeed, change is a property of all biological systems. Since differential equations are a means to mathematically describe change, they are often used to study biological phenomena.

Hodgkin and Huxley's work [?] with the squid giant axon was the first major biophysically-based model of a neuron's action potential. In their model, current flows through sodium and potassium channels which open or close at rates depending on the membrane potential. Other channel types and even synapses can be modeled similarly. These types of models have been used to represent everything from single cells to thousands of neurons.

A typical biological experiment consists of a given system (an animal, region of the brain, etc...) that is altered in some way (e.g. presentation of a stimuli or administration of a drug) and then observed to see how it responds. The system might then be altered and observed again. Analogously, we can mathematically simulate an experiment by defining a system of equations and repetitively altering parameters and integrating with respect to time.

Snnet is a python [?] library for the specification and simulation of neural networks designed to facilitate these sort of simulations. It is especially designed to simplify working with parameter sweeps, random networks, heterogeneities, and noise.

No knowledge of programming or python is assumed or required to use snnet[1]. A typical snnet simulation requires a line to load the snnet library and (optionally) a line to indicate the definition of a simulation protocol. The remainder of the file is devoted to defining the simulation and does not require any python keywords.

Simulation results can be analyzed from within python using the snnet library and standard

[1]See `www.oreillynet.com/pub/a/network/2000/06/02/magazine/python_first_language.html` for the case for learning python as a first language.

tools or exported for use with other packages. Python was chosen for snnet in part for its wide availability of free analysis routines. Other computational neuroscience tools have made the same choice: Brian [?], pynn [?], NEURON [?], MOOSE [?], and NEST [?] can be controlled via python as well.

Chapter 2

Installation

Snnet requires python 2.4 - 2.7; the python 3.x series is not backward compatible. In the future, snnet will be extended to work with both the 2.x and 3.x series.

The recommended and simplest way to use snnet is to include the snnet folder in every directory with snnet code. This way, as long as you do not modify or replace the folder or its contents, you always know exactly what version of snnet was used to run a particular set of simulations, ensuring reproducibility.

An alternative approach is to place the snnet folder inside of the python `site-packages` directory or any other directory listed in the `PYTHONPATH` environment variable. This approach allows running snnet from every path and avoids having duplicate copies, but does not allow for the simultaneous use of multiple versions.

2.1 Integrators

Snnet is designed to be extensible to allow the use of, in principle, any implementation of any integration algorithm. Four integration options are provided in the standard release: `fast_xpp_integrator`, `xpp_integrator`, `matlab_integrator`, and `matlab_mex_integrator`. Each requires one or more programs to be available on the system path, as follows:

2.1.1 fast_xpp_integrator

This is the recommended integrator for Linux and OSX. It is the fastest, most capable, and most tested integrator, but it does not support Windows. Needs no non-free software. Requires:

- `xppaut`[1]
- `gcc`

2.1.2 xpp_integrator

Works in Linux, Mac, and Windows. Needs no non-free software. Does not support noise, delays, or concentrations. Requires:

- `xppaut`

2.1.3 matlab_integrator

Works in Linux, Mac, and Windows. Does not support delays or concentrations. Requires:

- `matlab`[2]

2.1.4 matlab_mex_integrator

Works in Linux, Mac, and Windows. Does not support delays. Requires:

- `matlab`
- `mex`

2.2 Optional additional software

While not required for basic use, snnet uses additional packages if installed to provide better performance or additional features:

[1] `http://www.math.pitt.edu/~bard/xpp/xpp.html`
[2] `http://www.mathworks.com/products/matlab`

- `numpy`[3] – allows frequency analysis, faster calculation of coefficient of variation
- `sympy`[4] – improves algebraic simplification of model equations
- `matplotlib`[5]– plotting, raster plots
- `gnuplot`[6]– plotting
- `ImageMagick`[7,8] – converting graphics formats.

Though not used by snnet directly, `scipy`[9] is useful for transferring data to and from MATLAB via `mat` files.

[3]`http://numpy.scipy.org`
[4]`http://sympy.org`
[5]`http://matplotlib.sourceforge.net`
[6]`http://www.gnuplot.info`
[7]`http://www.imagemagick.org/`
[8]Required by the snnet interface to `gnuplot`
[9]`http://www.scipy.org`

Chapter 3

Defining the simulation

Conceptually, a biological experiment consists of two distinct parts: the cells, organisms, etc... that are being experimented with, and the protocol that is being used. Likewise, there are two phases to defining a snnet simulation: specifying the dynamics, and specifying the protocol (parameter changes, etc...).

3.1 Model equations

The dynamics of the system are typically specified in a `snnet` file, however they can also be provided as a string to the control code, see below.

A `snnet` file describes the dynamics of a cell type, organelle type, or global process. Since no two cells are identical, snnet allows parameters to be specified as constants or as elements of some probability distribution.

A `snnet` file is a simple text file, and can be created or edited with any text editor. The `snnet` distribution includes one example, `hh.snnet`, also reproduced below.

Line order does not matter in a snnet file, with the partial exception that the first differential equation declared will be assumed to represent a cell's membrane potential, unless otherwise declared. For maximum clarity, it is recommended that you always explicitly declare which variable refers to the membrane potential.

In general, snnet recognizes the order of operations, parentheses, many common functions, the arithmetic operations `+`, `-`, `*`, `/`, `**`, `^`, where `*` denotes multiplication and `**` and `^` both represent exponentiation.

3.1.1 Comments

Good programming practice encourages the use of comments, so we begin here. A comment is a human-readable description of what the code is trying to accomplish; it should not be a direct translation of the code. i.e. Avoid comments like "add `a` and `b`."

Any line or portion of a line beginning with a # in snnet or python is treated as a comment and ignored by the computer. e.g.

```
# This is a comment.  The computer will ignore it.
```

3.1.2 Differential equations

State variables have an initial value but change over time according to some differential equation. Specify the initial condition with an assignment to the state variable and indicate the derivative with a prime.

Example:

To represent the equation for logistic growth of a population P that is 500 at time 0 and has carrying capacity 1000,

$$\begin{aligned} P(0) &= 500, \\ \frac{dP}{dt} &= P\,(1000 - P) \end{aligned}$$

write

```
P = 500
P' = P * (1000 - P)
```

3.1.3 Algebraic expressions

It is often convenient to compute intermediate quantities before writing the differential equations. *Snnet does not distinguish between algebraic expressions and parameters.* This means that any expression defined as a constant or probability distribution can later be replaced with an algebraic expression during the simulation. Thus synaptic conductances can easily be modified on a per-simulation basis to be modulated by various hormones, etc...

To define an algebraic expression (or parameter), simply state the variable name, an equal sign, and then the value. Differential equations and other algebraic expressions can refer to an algebraic expression, regardless of the order they are defined in the file. In particular, this means that algebraic expressions cannot be redefined in the same file.

For example in the Hodgkin-Huxley equations, the leak current is defined by

$$I_\ell = g_\ell\,(v - v_\ell).$$

In snnet, this becomes

```
Il = gl * (v - vl)
```

3.1.4 Functions

Snnet recognizes the following functions of one variable:

`exp`, `sin`, `cos`, `tan`, `atan`, `acos`, `asin`, `log`, `log10`, `abs`, `heav`, `sinh`, `cosh`, `tanh`, `erf`, `erfc`, `sign`

and the following functions of two variables:

`max`, `min`, `atan2`

User-defined functions of any number of variables are also permitted. They may be defined by specifying the function name, a comma separated list of parameters enclosed in parentheses, an equal sign, and then an algebraic expression. For example:

```
add_three_numbers(a, b, c) = a + b + c
```

3.1.5 Variable name rules

State and algebraic variables and function names must begin with a letter and can consist only of letters, numbers, and underscore. Snnet itself is case sensitive, but some integrators (`xpp_integrator`) are not, so it is best to avoid using state variables that differ only in capitalization. Algebraic variable names never appear in generated code, so it is safe, but discouraged, to use names differing only in capitalization with them.

3.1.6 Numbers and probability distributions

At any point a number might be used in a differential equation or an algebraic expression, a probability distribution can also be specified. Whenever a cell or organelle is created from the snnet file, fixed values will be chosen from each distribution for that particular cell.

Uniform and normal distributions are supported.

For a value uniformly between a and b, write `[`a `:` b`]`. For a value from a normal distribution with mean a and standard deviation b, write a `[`b`]`. For a value from a normal distribution with mean a and standard deviation b percent of the mean, write a `[`b`%]`. Note that a and b are numbers; they are not expressions.

Very large and very small numbers can be input using the standard computer variation of scientific notation. That is, write 3.14×10^{-6} as `3.14e-6`.

3.1.7 Current time

The variable `t` contains the current time, unless it is otherwise set by the snnet file.

3.1.8 Special considerations

Capacitance.

In many models, the membrane capacitance is taken to be 1 μf/cm^2. If this is not the case, you must manually set the variable `capacitance` to the correct value, otherwise synaptic connections will not work correctly. This also means that the `capacitance` variable may not be used for any other purpose.

Organelles.

Organelles may be affected by the state of the cell that contains them. To refer to the variable `var` of the enclosing system, write `external_var`.

Concentration.

Conservation of mass dictates that if the volume of a cell changes over time, chemical concentrations must change as well. If this is the case in your model, declare the variable `volume` as either a differential variable or algebraic expression, and explicitly declare the concentrations by writing `concentrations` and then a comma separated list of concentrations. `volume` is the total volume enclosed by the object, including the volume of any nested organelles. `effective_volume` is calculated from the total volume of the object and the total volume of the objects it contains; it is the free volume not contained in any sub-object.

Internally, snnet keeps track of the corresponding mass and divides to get concentration whenever necessary.

For example, if volume is initially 1, grows according to $volume' = volume\,(2 - volume)$ and `ip3` and `ca` represent chemical concentrations within your volume, your snnet file should contain the lines

```
volume = 1
volume' = volume * (2 - volume)
concentration ip3, ca
```

Note that the initial concentrations of `ip3` and `ca` still need to be specified. If `ip3` also changes due to a reactions term `reaction`, include the line

```
ip3' = reaction
```

To specify a chemical flux from a neuron into the extracellular space or from an organelle into the neuron that contains it, use `flux`. Fluxes are not included in the derivative line. They are defined on the equations for the inner object and can only be used with variables declared as concentrations. For example, to indicate that calcium (`ca`) fluxes out of an ER into the cytosol at a rate `jip3r - jserca + jleak`, include

```
ca flux jip3r - jserca + jleak
```

in the snnet file for the ER.

Note that chemical concentration support is new in version 0.1.0, and is one of the least tested features of snnet. Only use fluxes between compartments, changing volume, and the concentration keyword if you are willing to extensively test your code and provide feedback. In the constant volume case, there should be no difference between specifying the `concentration` keyword and not specifying it, except that neither `xpp_integrator` nor `matlab_integrator` currently support concentrations. Since the support for explicit concentrations is new, the recommended approach is to omit the keyword unless the volume is nonconstant.

3.1.9 Converting models from xppaut

The snnet file specification is very similar to xppaut's ode file format. Most models can be converted to snnet simply by removing line prefixes (`par`, `init`, `!`, `number`, etc...), the end-of-model marker "`done`", and integration options. Integration options can be specified later as part of the simulation protocol. If any derivatives were specified using the form `dname/dt`, these must be changed to `name'`.

Less common issues include:

- multiple neurons in one file – split into separate files to allow using snnet's network tools.
- capacitance – many models assume membrane capacitance is 1; if this is not the case, set the variable `capacitance` to the correct value, otherwise the synapses will not work correctly.
- line continuation – snnet does not support line continuation; remove backslash-newlines.
- case sensitivity – xppaut is not case sensitive, but snnet is. If the `ode` file refers to variables inconsistently, load the snnet using the `ignore_case=True` option, see below.
- comments – comments beginning with a quotation mark should be changed to begin with a # sign.
- arrays – are not supported. That is, there is no support for `[j]`, `shift`, `del_shift`.
- unsupported keywords – `delay`[1], `if`, `then`, `else`, `global`, `special`, `solve`, `bdry`, `wiener`, `table`, `markov`, `volterra`, `options`, `set`, `export`, `besselj`, `bessely`, `hom_bcs`, `sum`.

3.1.10 Example: Hodgkin-Huxley equations

The following is the classic model of Hodgkin and Huxley. Note that `iapp` is defined as a probability distribution with mean 0 and standard deviation 1, so different cells will have different values for applied current.

[1] `fast_xpp_integrator` supports synaptic delays, specified at connection time.

```
# sodium
gna = 120
vna = 50

# potassium
gk = 36
vk = -77

# leak
gl = .3
vl = -54.4

# applied current
iapp = 0 [1]

# helper functions
am = .1 * (v + 40) / (1 - exp(-(v + 40) / 10))
bm = 4 * exp(-(v + 65) / 18)
ah = .07 * exp(-(v + 65) / 20)
bh = 1 / (1 + exp(-(v + 35) / 10))
an = .01 * (v + 55) / (1 - exp(-(v + 55) / 10))
bn = .125 * exp(-(v + 65) / 80)

# ionic currents
ina = gna * m ^ 3 * h * (v - vna)
ik = gk * n ^ 4 * (v - vk)
il = gl * (v - vl)

# state variable dynamics
v' = -(ina + ik + il) + iapp
m' = am * (1 - m) - bm * m
n' = an * (1 - n) - bn * n
h' = ah * (1 - h) - bh * h

# initial conditions
v = -65
m = .05
n = .317
h = .6
```

3.1.11 Example: Fast-Spiking Interneuron (Golomb et al 2007)

This model is from Golomb et al 2007 and illustrates user-defined functions, multiple declarations on one line, alternate exponentiation syntax (`**`), and proper declaration of capacitance to allow for non-unity values.

```
# model from
# Golomb D, Donner K, Shacham L, Shlosberg D, Amita Y, Hansel D (2007).
# Mechanisms of Firing Patterns in Fast - Spiking Cortical Interneurons.
# PLoS Comput Biol 3:e156.
#
# based on the xppaut version at
# http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=97747

# this will be automatically used by synapses
capacitance = 1

# parameters
Iapp = 3.35
gA = 0.39
theta_m = -24.0
gNa = 112.5
gK = 225.0
gL = 0.25
sigma_m = 11.5
theta_h = -58.3, sigma_h = -6.7
theta_n = -12.4, sigma_n = 6.8
theta_t_h = -60, sigma_t_h = -12.0
theta_tna = -14.6, sigma_tna = -8.6
theta_tnb = 1.3, sigma_tnb = 18.7
theta_a = -50, sigma_a = 20
theta_b = -70, sigma_b = -6
tau_b = 150, tau_a = 2
power_n = 2.0
V_Na = 50.0, V_K = -90.0, V_L = -70.0

# auxilary function:
GAMMAF(VV, theta, sigma) = 1.0 / (1.0 + exp(-(VV - theta) / sigma))

# functions:
m_inf = GAMMAF(V, theta_m, sigma_m)
h_inf = GAMMAF(V, theta_h, sigma_h)
n_inf = GAMMAF(V, theta_n, sigma_n)
```

```
a_inf = GAMMAF(V, theta_a, sigma_a)
b_inf = GAMMAF(V, theta_b, sigma_b)
tau_h(V) = 0.5 + 14.0 * GAMMAF(V, theta_t_h, sigma_t_h)
tau_n1(V) = 0.087 + 11.4 * GAMMAF(V, theta_tna, sigma_tna)
tau_n2(V) = 0.087 + 11.4 * GAMMAF(V, theta_tnb, sigma_tnb)
tau_n(V) = tau_n1(V) * taun_n2(V)

# currents
ina = gNa * m_inf ** 3 * h * (V - V_Na)
ik = gK * (n ** power_n) * (V - V_K)
il = gL * (V - V_L)
ia = gA * a ** 3 * b * (V - V_K)

# odes
V' = (-ina - ik - il - ia + Iapp) / capacitance
h' = (h_inf - h) / tau_h(V)
n' = (n_inf - n) / tau_n(V)
a' = (a_inf - a) / tau_a
b' = (b_inf - b) / tau_b

# initial conditions
V = -70.038
h = 0.8522
n = 0.000208
a = 0.2686
b = 0.5016
```

3.2 Simulation protocol

Simulation protocols are specified using python, however in practice there are often only two lines of pure python syntax.

First we must load the snnet library:

```
from snnet import *
```

Then we must indicate that what is to follow is a protocol and not to be run immediately[2].

[2]This is not strictly necessary, but it provides for an automated way of doing multiple runs, parameter sweeps, and saving the results.

We do this by declaring a function in python. I usually name my function `simulation`. The corresponding declaration reads

```
def simulation():
```

Everything that follows that is part of the protocol must be indented by a uniform amount.

3.2.1 Collections of cells

Initial creation

The primary unit in snnet is a collection of cells. A collection is a set; it is impossible for a cell to belong to a given collection more than once.

Initial collections of cells are by cell types, loaded in by the `neurons` function. For example,

```
hcells = neurons('hh.snnet', 30)
```

creates thirty neurons with dynamics as described in the file `hh.snnet`. If the snnet file describes probability distributions for parameters, each neuron will have a specific value chosen independently from all the others. Instead of specifying a filename, one can also provide a string containing the model description, in the exact same format as would be written in a snnet file. If the number of neurons (here 30) is omitted, only one will be created.

The state variable whose dynamics are described first in the snnet file will be assumed to represent membrane potential. If snnet does not know which variable contains the membrane potential, then synapses, raster diagrams, and field potential calculations will not work correctly. To manually specify the potential variable, pass an argument named `potential_var` to `neurons`. For example, to declare the potential variable to be ‘v’, use

```
hcells = neurons('hh.snnet', 30, potential_var='v')
```

The `neurons` function supports two additional standard arguments: `ignore_case=True` causes snnet to treat all variable names in the file as lower case. This option is useful for importing models from descriptions that are not case-sensitive. Passing `verify_defined=False` causes snnet to not check that all the variables are defined. Generally a model can only access its own variables, those of the environment its contained in (if an organelle), and global variables, but this option allows a cell to specifically reference another cell's state variables.

The `neurons` function also allows overriding the values for algebraic expressions in the snnet file. There are other ways of doing this, see 3.2.5. As an example, to replace the value of `gna` described in `hh.snnet` with 123, use

```
hcells = neurons('hh.snnet', 30, gna=123)
```

Any number of parameters may be overridden simultaneously in this manner. All overrides specified in this way are themselves overridden by values declared by `default_value` or in the `run` function.

Set operations

Collections created via the `neurons` function automatically represent the same type of cell, however different models may suggest different organization systems. All such collections may be formed by taking subsets of existing groups and combining via set operations.

There are two main options for taking subsets of a collection: taking random subsets or taking specific cells based on index in the collection.

To take a random subset of `hcells` of size 5, use `subset`:

```
new_collection = hcells.subset(5)
```

To extract a specific cell or range of cells based on index, use bracket notation. Note that snnet follows the python convention of numbering starting at 0. Thus to extract the third cell (position 2) from `hcells`, use

```
result = hcells[2]
```

Here `result` is a collection of neurons of size 1.

To extract a range of cells, use slice notation $a:b$. By python convention, index a is included, while index b is not. Thus to create a new collection of cells from positions 0 through 14 inclusive of `hcells`, use

```
result = hcells[0:15]
```

Multiple slices and singleton extractions may be combined in one by separating each with commas inside the brackets:

```
result = hcells[2, 3, 6:10, 12:15]
```

Collections support standard set operations. If `a` and `b` are collections of cells, then:

- union: `a + b`, alternatively `a | b`
- difference: `a - b`
- symmetric difference: `a ^ b`
- intersection: `a & b`
- complement: ~`a`

Collections may also be compared to check for containment or equality: `a<b`, `a<=b`, `a>b`, `a>=b`, `a==b`, `a!=b`.

The number of objects in the collection `a` is `len(a)`.

Interactive experimentation. Snnet can run in an interactive python session (initiate one by typing "python" on the terminal window). This is useful for testing many features, but it is especially helpful in checking that you understand how snnet does set operation on collections of neurons. When a collection is displayed, it includes a list of the indices of the neurons it contains. The following example session shows the interactive creation of a large collection, then two subsets, and tests of their union and set difference:

```
>>> from snnet import *
>>> hcells = neurons('hh.snnet', 30)
>>> hcells[1:10]
NeuronGroup([1, 2, 3, 4, 5, 6, 7, 8, 9], sim, name = 1)
>>> a=hcells.subset(5)
>>> b=hcells[28, 19, 12, 14, 7]
>>> a
NeuronGroup([0, 17, 26, 19, 7], sim, name = 2)
```

```
>>> b
NeuronGroup([12, 19, 28, 14, 7], sim, name = 3)
>>> a+b
NeuronGroup([0, 7, 12, 14, 17, 19, 26, 28], sim, name = 4)
>>> a-b
NeuronGroup([0, 17, 26], sim, name = 5)
```

Names are important

Snnet remembers the names given to all collections prior to the last snnet function call (typically an `advance` function). In the above example, we named three collections: `hcells`, `a`, and `b`. Analysis functions that take a `cells` argument can take the name of a collection to study.

3.2.2 Global dynamics

Global dynamics are dynamics that potentially affect the entire system. One typical use is to model chemical dynamics in the extracellular medium. All of their state variables are accessible to any other equation in the system.

Load global dynamics with the `global_dynamics` function. Its syntax and options are identical to the `neurons` function except that by definition, there can only be one copy, so there is no option for specifying the quantity. Assign the result of the `global_dynamics` to a variable to allow modifying its algebraic expressions later.

As in example, if `dopamine.snnet` defined dopamine dynamics, then one might include the line:

```
dopamine = global_dynamics('dopamine.snnet')
```

3.2.3 Synapses

Synaptic transmission is a major form of neuronal communication. Snnet provides built-in support for ionotropic chemical synapses. At an ionotropic chemical synapse, neurotransmitters are released from a pre-synaptic neuron in response to an action potential, travel across the synaptic cleft, and activate receptors, triggering current influx (or efflux).

Connection declaration

To connect all of the pre-synaptic cells `pre` to all of the post-synaptic cells `post` using a synapse of type `syn` (see below), use

```
post.connect(pre, synapse=syn)
```

By default `connect` will not connect a neuron to itself. If for some reason, self-connections are desired, pass the option `allow_self_connect=True` to the `connect` method.

To connect each post-synaptic cell to a fixed number (without repetition) of pre-synaptic cells, specify that number as the `n` parameter of `connect`. For example, to connect each `post` cell to five `pre` cells with synapse type `syn`, use

```
post.connect(pre, n=5, synapse=syn)
```

To connect cells using a specified pair-wise probability (a number between 0 and 1, inclusive), specify the probability as the `p` parameter of `connect`. For example, to connect (on average) 30% of the pre-synaptic cells to a given post-synaptic, with the other variables as before, use

```
post.connect(pre, p=.3, synapse=syn)
```

Subset notation may be combined with the `connect` method. For example, to connect `post` cell 0 with `pre` cells 1, 6, and 10, write

```
post[0].connect(pre[1, 6, 10], synapse=syn)
```

Synaptic types

A synaptic type defines how the synaptic current depends on a synaptic gating variable, defined in the pre-synaptic cell. Snnet provides two types: `ion_synapse` (this is almost always the correct choice) and `applied_current`. Additional user-defined types may be made by subclassing `snnet.synapse.Synapse`.

Current from an `ion_synapse` has the form $s\, g_{\text{syn}}\, (v - v_{\text{syn}})$, where s is the gating variable, g_{syn} is the conductance, v is the membrane potential, and v_{syn} is the reversal potential.

Excitatory synapses have high values of v_{syn}; inhibitory synapses have low values. The variable corresponding to membrane potential is declared in the `neurons` function or, if not, is assumed to be the first variable whose derivative is specified in the snnet declaration. The values for s, g_{syn}, and v_{syn} are specified (in order) as mandatory parameters to `ion_synapse`.

For example, to connect all of `post` to all of `pre` where the synaptic gating variable was called `s` in the declaration of `pre`, and the conductance and reversal potential were called `gsyn` and `vsyn` in the declaration of `post`, write

```
post.connect(pre, synapse=ion_synapse('s', 'gsyn', 'vsyn'))
```

Numeric values may be specified in place of the strings for the parameters. `applied_current` causes a current of magnitude $s\, g_{\text{syn}}$ in the post-synaptic cell. Unlike an `ion_synapse`, `applied_current` does not depend on membrane potential, so there is no `vsyn` to specify. In the simplest case, use

```
post.connect(pre, synapse=applied_current('s', 'gsyn'))
```

It is sometimes useful to allow heterogeneities in the choice of synapses, especially in the case of delays (below). To do this, pass a list of synapses (denoted by square-brackets and separated by commas) to the `synapse` parameter. For example:

```
post.connect(pre, synapse=[syn1, syn2, syn3])
```

In this case, each synapse is chosen uniformly at random from the provided list.

Delays

Neurons are not points; it takes time for action potentials to propagate down an axon, for neurotransmitters to cross the synaptic cleft, and for post-synaptic potentials to reach the axon hillock. In certain models, these delays critically affect the dynamics. To specify the delay (either as a number of ms or as a string containing a variable defined on the pre-synaptic side), simply pass in a value to `ion_synapse` or `applied_current`'s optional parameter `delay`. For example, to include a 5 ms transmission delay between the cells `pre` and `post`, write

```
post.connect(pre, delay=5)
```

Of the integrators included with snnet, only fast_xpp_integrator supports delays. The maximum delay must be specified manually by setting the `delay` option on the integrator. For example, if the maximum delay is no more than 6 ms, declare

```
fast_xpp_integrator.delay = 6
```

before invoking the `run` function.

Metabotropic synapses

If the ultimate effect of metabotropic receptor activation is to admit an ionic current, then metabotropic receptors may be modelled identically to ionotropic receptors.

If, instead, receptor activation is important due to the resulting chemical cascade (for example, in some cells activation of mGluR5 results in the production of IP_3 which can trigger a calcium wave), then the input must be manually modified on a per-cell basis. Specify `verify_defined=False` in the call to `neurons` to allow the created neurons to reference variables in other neurons.

3.2.4 Noise

In neural modelling, there are at least two major sources of noise: the stochastic opening and closing of ion channels, and input from other cells in the nervous system that are not explicitly modelled. The first source may often be safely neglected due to the large number of ion channels in a cell, and is not presently directly supported by snnet.

Declaration

Synaptic input from cells outside the model by creating a collection of noise sources using either `square_wave_noise`, `exponentially_decaying_noise`, or a user-defined subclass of `snnet.Noise`. The two included options differ only in the shape of the synaptic gating dynamics and the specification of the parameters for defining the shape. The parameters for each are, in order, the number of sources, the firing rate, and shape parameters. They return a collection of noise sources which may be freely combined with collections of neurons.

The integrator `xpp_integrator` does not support noise, but all of the other included integrators, including `fast_xpp_integrator` do.

A note of caution: snnet does not force integrators to notice noise events. That is, if the time step is too large, it is possible for a noise event to be ignored. To avoid this issue, ensure the maximum step size is at least several times smaller than the duration of a noise event.

square_wave_noise. This function returns noise sources where the synaptic gating variable fires a series of square wave pulses. The shape is defined using two parameters: `duration` and `max_value`. In this case, the gating variable would be 0 until a pulse event, instantly rise to `max_value`, remain there until `duration` after the onset, then immediately return to zero.

```
noise_source = square_wave_noise(10, firing_rate=5, duration=2, max_value=1)
```

creates a collection of ten noise sources, each generating square wave pulses at irregular intervals according to a Poisson process with an average firing rate of 5 Hz, with each pulse maintaining a value of 1 for 2 ms.

All of the parameters are optional: by default only 1 noise source will be created, which will fire according to a Poisson process at 5 Hz with a maximum value of 1 maintained for 1 ms.

Firing rate. If `firing_rate` is set to a number, snnet has the noise sources fire irregularly according to a Poisson process at the specified frequency. To specify periodic firing instead, write the desired frequency inside a call to the `periodic` function. For example, to create one square wave noise source that fires periodically at 5 Hz, but otherwise uses the default options, write

```
periodic_source = square_wave_noise(frequency=periodic(5))
```

Arbitrary, but pre-determined firing rules (that is, rules that do not depend on the current state of the simulation) may be specified by setting `frequency` to a function that takes a `snnet.Simulation` object and returns the number of milliseconds to wait between spikes.

exponentially_decaying_noise. This noise source simulates synaptic gating variables that immediately rise to `max_value`, maintain that level for `delay` ms, then decay exponentially with time constant `decay_rate`. The default for `max_value` is 1, for `delay` is 0, and for `decay_rate` is 2. Firing rate options are identical to those for `square_wave_noise`. As an example:

```
enoise = exponentially_decaying_noise(5, firing_rate=10, delay=1)
```

Connecting

Collections of noise sources may be connected to neurons identically to collections of neurons, with the exception that the synaptic gating variable need not be specified. If a gating variable is specified, it is ignored, allowing the same code to work with both neurons and noise sources. This flexibility is useful for testing the response of a small portion of a network to artificial input. For example, to connect `ecells` to the noise sources `enoise` with probability .3, reversal potential 0 mV, and connection strength `gsyn`, write

```
ecells.connect(enoise, p=.3, synapse=ion_synapse('gsyn', 0))
```

Delays are not supported for noise sources.

3.2.5 Parameters and other algebraic expressions

Most models for cells, organelles, and global dynamics will be defined in part using parameters and other algebraic expressions.

Default values

While values for these expressions are typically provided in the model description snnet file, some protocols will wish to provide their own values, either to override a numeric value or to replace a constant with an expression.

We have already seen that the `neurons` and `global_dynamics` functions allow parameter overrides on a per-file-import basis; `default_value` works on all subsequent model declarations and takes precedence over the previous method. To use `default_value`, simply call it with a variable name and a value. For example, to make `gk` default to 123, write

```
default_value('gk', 123)
```

As a second example, even if `gna` was initially defined as a constant, we can redefine it as an expression. Here we make it a linear function of `da`:

```
default_value('gna', 'da * gna_m + gna_b')
```

No assumption is made about whether or not any models referring to `gna` define `gna_a` or `gna_b` prior to this function call. All that is required is that all of the variables are defined prior to any integration attempts.

Specifying parameters when calling the `run` function overrides any values specified by `default_value`. Furthermore, values manually set after creation have the highest precedence.

The `value` function returns the value of any override that works on a simulation-wide level. The values it returns were either specified by the `run` function call or in a `default_value` statement. For example, to print the value of the override `gk`, write:

```
print value('gk')
```

If no such override is defined, `value` returns `None`. As an alternative, `value` also accepts a second argument, which is returned instead if no override is defined.

Reading values

It is sometimes useful to read the current values of parameters assigned to a particular collection of cells. This is useful for restoring parameters to their previous values and for making relative changes in the parameter values (e.g. increase the applied current by a fixed amount to all cells).

If `hcells` is a collection of cells with parameter `iapp`, then to save a copy of the values used for `iapp` in the variable `iapp_vals`, use dot notation:

```
iapp_vals = hcells.iapp
```

The dot notation returns a `ParameterList` – which behaves much like a regular python list except it defines standard arithmetic vector operations – where the entries are ordered in the same order as the collection.

Note that if the values were originally specified as a probability distribution, this will return the values chosen from that distribution.

Setting values

Dot notation can also be used to assign values to parameters. For example, to set `hcells.iapp` to 2, write

```
hcells.iapp = 2
```

If the variable name did not previously exist, then this assignment creates it. As with `default_value`, the right-hand side may be a constant, distribution, or algebraic expression.

To make relative changes to parameter values, read the parameters into a `ParameterList`, apply arithmetic manipulations, and store the result. This can be done in one line. For example, to increase `hcells.iapp` by 2, write

```
hcells.iapp = hcells.iapp + 2
```

More concisely, one may write

```
hcells.iapp += 2
```

Globally overridden variables may be referenced by name in these assignments. To increase `hcells.iapp` by the amount in `stimulus`, use

```
hcells.iapp = hcells.iapp + 'stimulus'
```

A note of caution: the value assigned is the global override value, regardless of whether or not an expression by that name is defined within `hcells`.

Global override values may also be suffixed by a standard deviation as in a snnet declaration. For example,

```
hcells.foo = 'bar [10%]'
```

sets the expression `foo` within the `hcells` to a value chosen from a normal distribution with mean `bar` and standard deviation ten percent of the mean.

The use of global overrides is preferable to constants or python variables because it allows the amount to be varied during a parameter sweep.

Note that if a parameter was originally declared to be 0 [1], setting it to 2, and the setting it back to 0 [1] does not return it to its initial state. Instead, a new value from the same probability distribution is selected. This is rarely the desired behavior. To return a variable to its initial value, read it as described above, save that value, and then restore by setting the variable equal to the saved value.

3.2.6 Organelles

Add an organelle or other compartment to a pre-existing collection of neurons or compartments using the collection's `add_compartment` method. The options are identical to the `neurons` function, and the return value is a collection of compartments which can be manipulated just like a collection of neurons. For example, if `er.snnet` contains a model of the endoplasmic reticulum (ER), we can include it in all the cells belonging to the collection `hcells` via

```
ers = hcells.add_compartment('er.snnet')
```

If `mcells` is a group of cells that contain ERs that were defined in `'er.snnet'`, then

```
mcells['er.snnet']
```

returns a collection of the ERs belonging to the collection mcells.

Internally, organelles and neurons are numbered consecutively in the order of creation. If a simulation created three neurons – 0, 1, and 2 – and then created an organelle, that organelle would be assigned index 3.

There are two main ways an organelle may communicate with its containing cell:

First, it may read the container's state variables. It does this by prepending the variable names with '`external_`'. As an example, if an organelle is contained in a cell with membrane potential `v`, then its dynamics can refer to `external_v`.

Second, the organelle may support a chemical flux across its membrane. In the snnet declaration, this can be declared using the `flux` keyword, but fluxes may also be added dynamically at run time. To do so, use the collection's `flux` method. This method requires specifying the internal variable, the flux rate, and optionally the name of the external variable. For example, to make the variable `cai` flux into the containing neuron's variable `ca` for the compartments `ers` at a rate `jflux` write

```
ers.flux('cai', 'jflux', external_var='ca')
```

See 3.4 for an example using organelles.

3.2.7 Integration

Initial time

By default, simulations are assumed to start at time 0, but this value can be adjusted via `set_initial_time`. For example, to start integration at time 10, use:

```
set_initial_time(10)
```

Only positive times are fully supported. At present, noise sources always fire their first event at a negative time near 0.

Advancing the simulation

Two functions are provided to advance the simulation: `advance` and `run_to`. `advance` integrates for a specified duration from the current time. For example, if the current time is 10 ms,

```
advance(20)
```

runs the integrator until time $30 = 10 + 20$ ms. By contrast,

```
run_to(20)
```

stops the integrator at time 20 ms if it is called at any time prior to 20 ms. Attempting to call `run_to` with a time before the current time is an error and will raise an exception.

A caution on parameters: variable substitutions are only made at each integration. This means that if `a` is defined in terms of `b`, but the definition for `b` is changed before integration, then `a` will use the new definition for `b`.

Labeling time points

Assign names to key time points for use doing later analysis via `name_time`. This is especially useful for comparing behavior between simulations where the same event is signaled at different numeric times. For example, if your protocol presents a stimulus at the current time point, you might want to include:

```
name_time('apply_stimulus')
```

3.2.8 Forking

Often one objective of an experiment is to compare the results from two initially identical but later different protocols. For example, a test of a drug may seek to compare the performance of control cells to that of cells exposed to the drug. In working memory or decision tasks, one might want to compare a network's response to different stimuli.

In this case, there is no need to run the identical portion of the simulation more than once; when the protocols diverge, simply declare a `fork()`, then continue describing the first procedure to completion. Save or otherwise process the results manually, then return to the previous simulation state via `end_fork()`. Forks may be nested or used repeatedly to allow for multiple branches of the simulation.

```
hcell = neurons('hh.snnet')

# portion common to both branches
advance(50)

# first fork, no applied current
fork()
advance(100)
save_simulation('fork1_' + default_filename())
end_fork()

# second fork, applied current
# no need for a fork here, since never returning to this point again
```

```
hcell.iapp = hcell.iapp + 1
advance(100)
save_simulation('fork2_' + default_filename())
```

3.3 Example: One Hodgkin-Huxley Cell

This example uses the Hodgkin-Huxley model `hh.snnet` defined in 3.1.10. It simulates one cell for a total time of 200 ms, where it is at rest for the first 50 ms, then subject to an applied current for 20 ms. Results are saved, in full, to the directory `results` to be analyzed later. Here the protocol is executed using the `run` command, described in chapter 4. If this program is saved in `hh.py`, then it may be run by typing "`python hh.py`" at the command line.

```
from snnet import *

# define the protocol
def simulation():
   # create one Hodgkin-Huxley cell
   h_cell = neurons('hh.snnet')

   # run until t=50 ms
   run_to(50)

   # apply a current, run for another 20 ms, remove the current
   h_cell.iapp = 5
   advance(20)
   h_cell.iapp = 0

   # advance to t=200 ms
   run_to(200)

# run the simulation
run(simulation, 'results', integrator=xpp_integrator, save_type=full)
```

3.4 Example: Wagner et al, 2004

```
cell = """
   # total cell volume (including any compartments)
   volume = 1

   # IP3 dynamics
   vprod = .075 # uM / s (in principle depends on surface area/volume ratio)
   kprod = .4   # uM
   v1 = .001    # uM / s
   v2 = .005    # uM / s
   v3 = .02     # uM / s
   lambda = 30
   k0 = .39     # uM
   k1 = 2.5     # uM
   k2 = .5      # uM
   k3 = 30      # uM

   # declare ca and ip3 to be a concentration (necessary for using flux)
   # this means that ca depends on effective volume
   # effective volume = total volume - volume organelles
   concentration ca, ip3

   # IP_3 degradation and production
   th = ca / (ca + k0)
   jkinase = (1 - th) * v1 * ip3 / (ip3 + k1) + th * v2 * ip3 / (ip3 + k2)
   jphos = v3 * ip3 / (ip3 + k3)

   # vprod includes the effects of cell surface area / volume ratio
   jprod = vprod * ca ** 2 / (ca ** 2 + kprod ** 2)

   # still need to divide by the effective_volume
   ip3' = (jprod - lambda * (jkinase + jphos)) / effective_volume

   # initial conditions
   ca = 1
   ip3 = 1.5
"""


er = """
   # calcium regulation
   vip3r = 8.5   # /s
   kip3 = .15    # uM
   kact = .8     # uM
   vleak = .01   # /s
   vserca = 1    # uM / s (was .65 in W et al)
   kserca = .4   # uM

   # channel state
   kinh = 1.9 # uM
   tau = 2    # s

   # volume of ER
   volume = .17

   concentration ca

   ip3 = external_ip3
   cai = external_ca

   hinf = kinh / (kinh + cai)
   m = ip3 / (ip3 + kip3) * cai / (cai + kact)
   jip3r = vip3r * m ** 3 * h ** 3 * (ca - cai)
   jleak = vleak * (ca - cai)
   jserca = vserca * cai ** 2 / (kserca ** 2 + cai ** 2)

   # calcium flux (outward = positive)
   ca flux jip3r - jserca + jleak

   # IP_3 receptor inactivation
   h' = ((hinf - h) / tau)

   # initial conditions
   h = 0
   ca = 3.94117647
"""

def simulation():
   c = neurons(cell)
   c.add_compartment(er)
   advance(200)

data = run(simulation)

subplot(2, 1, 1)
```

```
plot(data.t, data.ca0)
ylabel('[Ca$^{2+}$]$_c$ in $\mu$M')

subplot(2, 1, 2)
# ca is a property of the cell, which has index 0
# h is a property of the ER, which has index 1
plot(data.ca0, data.h1)
xlabel('[Ca$^{2+}$]$_c$ in $\mu$M')
ylabel('$h$')

savefig('wagner2004.pdf')
```

Chapter 4

Running simulations

Execute simulation protocols using snnet's `run` function. The `run` function allows the specification of multiple runs (useful for studying the effects of heterogeneities or other randomness), parameter sweeps, and automatic parallelization.

4.1 Basic usage

The simplest common usage of the `run` command is to run a simulation protocol once and save the results in a directory. As an example, if we have a protocol that we named `simulation` in its `def` statement and we wish to save the results in a folder `results`, we would use

```
run(simulation, 'results')
```

This or any other `run` command should be written unindented, below the `def` block defining the protocol.

The results are saved in the file `0.sdat` in the specified directory. If the directory does not already exist, it will be created when the data is ready.

If only one simulation is run, `run` returns a `SimData` object containing the full data. This is useful for immediately performing analysis on the results. If we are confident that we will never want this data again, we can omit the directory name from the `run` function call to avoid unnecessarily using disk space. For example, one might write:

```
data = run(simulation)
```

4.1.1 Save types

By default, run saves all the state variables at all of the time points evaluated. While snnet attempts to compress its save files, large or long simulations will lead to large file sizes. To combat this problem, snnet provides multiple save options. Four are provided – `spikes`, `enhanced`, `csv`, and `full` – and the advanced user can define new save types by inheriting from `SimData` in `snnet/sim_data.py` and redefining the relevant methods.

The `spikes` save type discards most data except for the spike times. The `enhanced` type is the same as `spikes` except it also preserves field potentials. `csv` saves all the state variables at all time points in a comma separated variable file suitable for importing into other tools. Note that state variable values do not fully describe the simulation; for example the `csv` format discards connectivity information that is preserved in the other built-in save types. Finally, `full` saves all of the simulation data and is the type that is returned by `run` calls that only perform one simulation.

To specify the save type, simply provide a `save_type` argument to `run`. For example:

```
run(simulation, 'results', save_type=spikes)
```

4.1.2 Repeated runs

To study the effects of heterogeneities, random networks, or other randomness in a model, one must run that model many times. To do so, provide a `num_times` parameter to `run`, specifying the number of times to run each protocol with the same set of parameters. For example:

```
run(simulation, 'results', num_times=10)
```

runs the protocol `simulation` 10 times using identical parameters. Results are saved in the files `0.sdat`, `1.sdat`, ...`9.sdat` in the directory `results`. The number in the filename is the random seed used in that particular simulation. To make the first simulation have a nonzero random seed, provide a `start_id` parameter with a positive integer value.

4.1.3 Parallelization

Computers with multiple cores or multiple processors can take advantage of snnet's parallelization. To run multiple simulations simultaneously (in the case of repeated runs or parameter sweeps), provide a `num_parallel` parameter to `run` specifying the maximum number of simulations to perform simultaneously.

For example, on a four core processor, one might run

```
run(simulation, 'results', num_times=10, num_parallel=4)
```

Snnet keeps the entire simulation state variable history in memory until the simulation is complete. Running many large or long simulations simultaneously could potentially use up all available memory. To avoid this problem, run a test to see how much memory one simulation requires and choose `num_parallel` accordingly.

In Linux and OSX, parallelization is process-based, while in Windows – due to underlying architectural differences – parallelization is thread-based. Standard python uses a global interpretor lock, so thread-based parallelization in python is less efficient than process-based. Other implementations of python might not share this limitation.

4.1.4 Integrators

Snnet comes with four different interfaces to integration routines: `fast_xpp_integrator`, `xpp_integrator`, `matlab_integrator`, `matlab_mex_integrator`. Additional integration interfaces may be defined by creating a class inheriting from `snnet.integrator.Integrator`.

Select the integrator to use for a run by passing its name to the `integrator` parameter of `run`. For example, to run using the `xpp_integrator`, use

```
run(simulation, integrator=xpp_integrator)
```

By default, snnet uses the `fast_xpp_integrator`, which requires `xppaut` and `gcc`, on Linux and OSX. Since that interface does not support Windows, snnet uses `xpp_integrator` by default on Windows. One advantage of using either of these interfaces is that `xppaut` supports a wide range of integration methods.

Options

Different integrators support different options. They may be set using dot notation, but these options must be declared in the control code prior to invoking the `run` function. For example, to set the step size `dt` on the `xpp_integrator` to .01, write

```
xpp_integrator.dt = .01
```

Options and default values for the various integrators are as follows:

- xpp_integrator
  - `dt` = .05
  - `method` = 'qualrk'
  - `bounds` = 10000
- fast_xpp_integrator
  - `dt` = .05
  - `method` = 'qualrk'
  - `bounds` = 10000
  - `gcc_flags` = '-m32'
  - `delay` = 0
- matlab_integrator
  - `max_step` = .05
  - `method` = 'ode23'
  - `rel_tol` = 1×10^{-3}
- matlab_mex_integrator
  - `max_step` = .05
  - `method` = 'ode23'
  - `rel_tol` = 1×10^{-3}

4.2 Parameter sweeps

It is often useful to examine how a model depends on its parameters. Generally one wants a model where the qualitative results do not change for small variations in parameters, but it may be important to see how the firing rate depends on the applied current, etc... Such explorations are also useful when initially picking parameters.

In the simplest case, one or multi-dimensional parameter sweeps can be performed simply by modifying the call to `run` to include a `sweep` parameter. The sweep parameter takes a python `dict` whose keys are parameter names and whose values are the corresponding parameter values to try. When using `run` to save, the results of a sweep will be stored in a hierarchical directory system by parameter name then value (repeating as necessary) within the specified directory.

For example, to perform a two-dimensional parameter sweep, testing the protocol `sim` using all $12 = 3 \times 4$ combinations of `gsyn` in 1, 2, or 3, and `iapp` in .5, 1.5, 2.5, 3.5, and saving in the folder `data`, write

```
run(sim, 'data', sweep={'gsyn': [1, 2, 3], 'iapp': [.5, 1.5, 2.5, 3.5]})
```

The `sweep` parameter also accepts a list of `dict` objects, each one of which is swept independently.

If `num_times` > 1, then each parameter set will be tried once before any parameter choices are repeated.

4.2.1 Modifying previous simulation parameters

Sometimes it is useful to start a parameter sweep or other exploration from the results of a prior simulation or simulations. Assuming the results were saved (with any extension not consisting solely of numerals) in a directory according to the `default_filename` standards of a parameter name, an underscore, and a parameter value (repeated, with each repeat separated by an underscore), then the function `parameters_from_dir` can be used to create a list of `dict` objects suitable for using with `sweep`.

For example, to run a new simulation protocol `sim2` using the parameters used to make figures stored in the directory `sim1figs`, use:

```
params = parameters_from_dir('sim1figs')
run(sim2, sweep=params)
```

A `for` loop can be used to modify the old parameters to allow another parameter sweep. For example, to run with everything as in `sim1figs` except with `iapp` being swept over 0, 1, and 2, use:

```
params = parameters_from_dir('sim1figs')
for p in params:
   p['iapp'] = [0, 1, 2]
run(sim2, sweep=params)
```

4.3 Advanced

4.3.1 Temporary files

Snnet uses both pipes and temporary files to communicate with other programs to perform integration. The use of pipes reduces the amount of time needed for file access. For example, on Linux and OSX, the results of an integration pass directly from the integrator to snnet; no file access is needed.

Temporary files contain a representation of the model in a form understood by the integrator, and are sometimes useful for debugging new models.

To leave a copy of the temporary files used for the last integration under a protocol in the current directory, include the option `leave_temp_files=True` when calling `run`.

The option `do_cleanup=False` leaves temporary files in the location they were created in the event that a simulation fails.

To specify where to create temporary files, pass the option `temp_dir_handler=f` where `f` is a python function of no variables that returns a path on which to store temporary files. The default is to store temporary files in the current directory except under Linux, where the ramdisk `/dev/shm/` is used, if available. Use of the ramdisk speeds up data transfer and ensures that any residual temporary files will be purged automatically on reboot.

4.3.2 Continuing after unsuccessful integration

A simulation may fail if the selected integrator is not able to stably integrate the model. By default, if such a failure occurs, the `run` command raises an exception and all subsequent integration is canceled. Integration failure is typically a sign that one should adjust the step size, choose a different integration method, or alter the model.

Under rare occasions (for example, when trying to determine the parameter range where integration is stable), it is desirable to continue on with the next parameter set instead of shutting down on failure. In these cases declare `continue_after_failure=True` in the call to the `run` function.

Chapter 5

Analyzing model results

Data access is through `sim_data.SimData` objects. For a list of all supported methods and properties, see Appendix B.

Snnet provides routines for common neuroscience tasks and tools for simplifying analysis, but it is not intended to provide extensive analysis tools; instead, the user should utilize standard math libraries. Scipy maintains a large list of scientific python software at `http://www.scipy.org/Topical_Software`.

5.1 Saving data

There are two main ways to save simulation results. The first, as we have already seen, is to specify an output directory when invoking the `run` command. The second is to invoke `save_simulation` within a simulation protocol. This function takes a directory name and optionally a filename and/or a `save_type`. For example, to save the spike times for the currently running simulation in the folder `foo` with a filename according to the `default_filename` function (which separates overridden variables and their values with underscores), use

```
save_simulation('foo', default_filename(), save_type=spikes)
```

5.2 Getting data

There are three main options for getting a `SimData` object to work with, depending on when you want it:

5.2.1 During a simulation

Sometimes it is useful to be able to access the simulation data during a simulation. For example if one is doing sweeps to locate a region of parameter space where the model exhibits a certain behavior, one might wish to check the status of a simulation part-way through and abort simulations that are not going to exhibit the desired behavior.

To get the data in this case, assign the result of `simulation_data()` to a variable. For example:

```
data = simulation_data()
```

5.2.2 Immediately after a single simulation

If the `run` function is used to only run one simulation at a given time (that is, no sweeps, `num_times` is 1 or omitted), then it will return a `SimData` object. In such a case, there might not be a need to save the data to the disk. For example, to run the simulation protocol `sim` and return the result in `data`, write

```
data = run(sim)
```

5.2.3 By loading saved data

To load saved data from a file, use the `load_simulation` function and pass in the name of the file. For example, to load the data from the file `bar.sdat`, use

```
data = load_simulation('bar.sdat')
```

Note that unlike the previous two methods which always return all of the simulation data (the equivalent to saving with the `full` save type), the data available via `load_simulation` necessarily depends on the `save_type`.

5.3 Working with data

For more methods, see Appendix B.

5.3.1 Individual simulations

For data containing the state variables, there are two ways of accessing their time course information: the most flexible way is to use indexing notation where the first index specifies the variable name as a string and the second specifies the the index of the neuron in question (if you want the value of a global state, only use the first index). For example, to get the time course of neuron 3's variable `v`, use

```
data['v', 3]
```

whereas to get the value of the global variable `w`, use

```
data['w']
```

If the indices and the variable names are known in advance and the variable names do not end in a number, then the time course information can be obtained using dot notation where the variable name and id number have been concatenated together. For example, the above two examples become

```
data.v3
```

and

```
data.w
```

In principle, almost all of the data is contained in the state variables, however snnet provides some useful methods and properties on `SimData` objects (accessible via the dot notation). A partial list with brief explanations follows, for more, see B.

- active – the number of cells active at given times
- connections – connection matrix information
- cv – coefficient of variation of ISIs for specified cells
- field_potential – field potential of arbitrary group of cells
- field_potentials – dict of field potentials by cell type
- isi – interspike intervals

SAY MORE HERE

5.3.2 Groups of simulations

Snnet functions

Snnet provides several functions that look at the properties of groups of simulations. A partial list with brief descriptions follows. For more information, see Appendix A.1.

- `avg_active_over_time` – find average number of cells active at specified times
- `collect_cv` – get coefficient of variation of ISIs
- `collect_isi` – get ISIs from all simualtions

Running custom analysis code

The function `run_f_on_dir` is used to run custom analysis tools on all of the data files contained in a given directory (and, optionally, its subdirectories). It supports automatic parallelization and returns a list of the function return values. For example, to run a custom function `analyzer` on all of the data files in the directory `foo` but not its subdirectories five at a time and return the results in `bar`, write:

```
bar = run_f_on_dir(analyzer, 'foo', num_parallel=5)
```

For more details see Appendix A.1.

5.4 Graphics

5.4.1 General plotting

Snnet is not a graphics library, however it automatically provides two graphics methods if the appropriate software is installed on your machine:

matplotlib

Snnet imports `matplotlib`, if available, which allows basic plotting functions. For details, see `http://matplotlib.sourceforge.net/`.

As an example, suppose `data` is a `SimData` object of type `full`. To make a new figure, plot the value of `v` from neuron 5 (recall: numbering starts at zero, so this is the sixth neuron defined) vs time `t`, add labels, and save in the file `foo.pdf`, use:

```
figure()
plot(data.t, data.v5)
xlabel('t (ms)')
ylabel('v (mV)')
savefig('foo.pdf')
close()
```

gnuplot

Gnuplot is a graphics package available from `http://www.gnuplot.info/`. Snnet provides an interface compatible with version 4.2 or later. For details, see Appendix A.4.

The snnet interface to gnuplot requires the ImageMagick graphics conversion programs: `http://www.imagemagick.org/`.

The previous example can also be done using the `gnuplot` interface as follows:

```
gnuplot('foo.pdf', data.t, data.v5, x_label='t (ms)', y_label='v (mV)')
```

5.4.2 Raster plots

Use `raster` to plot raster diagrams of network activity, grouped by cell type. `raster` takes either a `SimData` object or a filename containing data to plot as the first argument.

In simplest usage, generate a raster plot of `data` via

```
raster(data)
```

These plots are generated using `matplotlib` and thus may be modified by other `matplotlib` graphics commands. The `raster` function does not save the image; to do so, use `matplotlib`'s `savefig` function:

```
raster(data)
savefig('data.pdf')
```

To generate raster plots for all data files in a directory, use `make_rasters`.

Additional parameters allow shading time intervals and choosing whether or not to display the field potentials. For full details, see Appendix A.4.

Chapter 6

Troubleshooting and Support

6.1 Online discussion group

Join the discussion group at `http://groups.google.com/group/snnet` to share code, discuss implementing your model in snnet, and more with other users.

6.2 Interactive graphics in matplotlib

Snnet automatically imports matplotlib, if available, because it uses the package for producing raster diagrams. To allow for working over remote connections, snnet sets the backend of matplotlib to Agg which is noninteractive.

To run an interactive graphics session with snnet, simply import matplotlib prior to importing snnet.

6.3 More helpful error messages

To get line numbers and other useful traceback information for errorhttp://www.gnuplot.info/s, set `num_parallel=1` when invoking the `run` function.

6.4 Compilation errors with fast_xpp_integrator

The fast_xpp_integrator integrator assumes that `xppaut` is compiled as a 32-bit program, even if the system is 64-bit.

6.4.1 64-bit xppaut

If, instead, snnet is to be run on a machine with a 64-bit version of `xppaut`, modify the control file to redefine the fast_xpp_integrator before use:

```
fast_xpp_integrator = integrator.GccXppIntegrator(gcc_flags='-m64')
```

6.4.2 Cross compiling

Some 64-bit Linux distributions do not include the libraries necessary for cross-compiling to work with 32-bit xppaut. Install the necessary libraries by typing

```
sudo apt-get install libc6-dev-i386
```

on the command line.

6.5 Unable to run xpp_integrator or fast_xpp_integrator

To use either of these integrators, snnet requires that xppaut is on your system path. Test if this is the problem by opening up a terminal and typing “xppaut”. If everything is setup correctly, a file-selection dialog box will appear[1]. If, instead, an error appears about not being able to find the program, either adjust the system path or move the xppaut binary to a location on the path.

[1]This is not necessarily the case. xppaut needs an X-server to display graphics, but not to perform the calculations needed by snnet.

Appendix A

snnet API

Many parameters have default values, specified in the following after an equals sign. Python allows you to specify named parameters in any order, if you use the variable name = value syntax. All of the following are loaded via `from snnet import *`.

A.1 Analysis

avg_active_over_time(dir_name, times, cells=None, num_parallel=1, activity_window=100)

Compute average number of cells active at given times.

Parameters:
 dir_name – the directory containing the simulations
 times – the times to check
 cells – which cells to look at (a list of indices or a string naming a cell type or a cell group)
 num_parallel – how many calculations to run simultaneously
 activity_window – how long ago a cell may have fired and still count as active
Note:

If cells is None or omitted, works with all cells.

collect_cv(dir_name, cells=None, time_range=(-inf, inf), combine_sims=True, num_parallel=1)

Collect the coefficient of variation (CV) for the interspike intervals.

Parameters:

dir_name – the directory containing the simulations
cells – which cells to look at (a list of indices or a string naming a cell type or a cell group)
time_range – the time interval in which to compute the CVs
combine_sims – if True, group all simulations together; else create lists for each simulation
num_parallel – how many calculations to run simultaneously

Note:

If cells is None or omitted, computes the CVs for all cells.

collect_isi(dir_name, cells=None, time_range=(-inf, inf), combine_cells=True, combine_sims=True, num_parallel=1)

Collect the interspike intervals (ISIs).

Parameters:
dir_name – the directory containing the simulations
cells – which cells to look at (a list of indices or a string naming a cell type or a cell group)
time_range – the time interval in which to compute the ISIs
combine_cells – if True, group ISIs from all cells in a simulation together, else create separate lists for each cell
combine_sims – if True, group all simulations together; else create lists for each simulation
num_parallel – how many calculations to run simultaneously

Note:

If cells is None or omitted, computes the ISIs for all cells.

compute_spike_times(t, v, threshold=0)

Return a list of the times t when v crosses threshold.

cv(items)

Returns the coefficient of variation (CV) of the items.

cv(items) = std(items) / mean(items)

discard_nans(*items)

Return a copy of the list(s) items without any NaNs.

frequencies(t, y, scale=1000)

Compute the magnitudes of frequencies present in y. This is essentially the output of the fast fourier transform algorithm.

Returns two lists, the first of frequencies, the second of magnitudes.
Assumes t is evenly sampled.

The default setting of scale=1000 assumes t in ms and we want frequency in hertz.

Requires `numpy`.

mean(x)

Compute the algebraic mean of x. The version from `numpy` is used, if available, else snnet provides one.

run_f_on_dir(f, dir_name, enter_subdirs=False, num_parallel=1, temp_dir='.')

Runs a function on the simulation data from each sdat (or pkl) file in dir_name.

Parameters:
  f – the function to run
  dir_name – the name of the directory
  enter_subdirs – should we apply f on files in subdirectories
  num_parallel – maximum number of evaluations to run simultaneously
  temp_dir – a string or function evaluating to a string representing a temporary directory

Returns:
  A list of the return values. (Or lists of lists, etc... if entering subdirectories.)

Note:

f may take 1 or 2 arguments. If f takes only 1 argument, passes the data; if f takes 2, passes data as first argument, filename as second.

Parallelization is with processes on Linux and OSX; otherwise, threads.

If num_parallel $>$ 1, then f must return a picklable object. (This is not usually a problem.)

simulation_data()

Return a SimData instance corresponding to the current simulation.

Do not use this instance after changing the simulation; that behavior is undefined and subject to change.

std(x, ddof=0)

Compute the standard deviation of x with ddof degrees of freedom. The version from `numpy` is used, if available, else snnet provides one.

A.2 Control

advance(t)

Run the simulation for an additional time t. See also `run_to`.

copy_simulation()

Return a copy of the current simulation. Typically not needed, unless using `set_simulation`.

current_simulation()

Return the current Simulation object.

This is not typically needed, but provided for advanced simulation control. For more, do a `help` on the object returned.

default_value(var, val)

Set a default value in the simulation.

This will override the values specified in the snnet file of any dynamics defined after this command has been run (but not those already defined.)

Values previously overridden (e.g. via sweep or param in run, as well as previous default_value calls) take precedence.

See also `value`.

end_fork()

End a simulation fork and return to the previous state. See also `fork`.

fork()

Initiate a new fork in the simulation. State will be saved so that snnet can return to this point. See also `end_fork`.

get_run_id()

Return the run_id, the default filename.

For simulations started via run, this is a number corresponding to the random seed used.

global_dynamics(dynamics, ignore_case=False, verify_defined=True, **params)

Load shared (global) dynamics from dynamics.

Parameters:
  dynamics – the snnet source
  ignore_case – should we drop capitalization
  verify_defined – should we check dynamics for completeness
  params – parameters with values

Note:

dynamics can be either a filename or a string containing equations.

parameters set here override the values in the snnet but are themselves overriden by sweeps or the param argument to run.

group(name)

Return the cell group with the given name. Typically not needed.

integrator_method(method=None)

Set or get the integrator. This is typically used only in simulations not controlled by `run`.

neurons(neuron, num=1, potential_var=None, ignore_case=False, verify_defined=True, **params)

Define a group of neurons from a snnet.

Parameters:
neuron – the snnet source
num – how many neurons to define
potential_var – the variable corresponding to membrane potential, if any
ignore_case – should we drop capitalization in the snnet?
verify_defined – should we check dynamics for completeness
params – any parameter changes to make to the snnet

Note:

neuron is a string containing either a filename or snnet equations.

Any changes to parameters made here can be overriden by a parameter sweep, or with the param argument to run.

run(sim, output_dir=None, params={}, sweep={}, integrator=default_integrator, num_times=1, save_type=full, num_parallel=1, post_run=None, start_id=0, do_cleanup=True, temp_dir_handler=default_temp_dir_handler, leave_temp_files=False, continue_after_failure=False)

Run a series of simulations.

Parameters:
sim – function defining the simulation protocol
output_dir – directory to store the results in (string, None, or function)
params – a dict of parameters to override with values
sweep – a dict of arguments to sweep over
integrator – which integrator to use
num_times – how many times to run each parameter set
save_type – output file_handler
num_parallel – how many simulations to run simultaneously
post_run – a function to call after each simulation
start_id – the simulation id # to start at (use for going back and running more)
do_cleanup – should we delete temp files when a simulation fails?
temp_dir_handler – function returning a string pointing to a directory to use for storing

temporary files
  leave_temp_files – should temporary files be left in the local directory after running a simulation
  continue_after_failure – if integration fails, should we keep trying?

Notes:

If only one simulation is ran, returns a Full SimData object with the results.

If output_dir is omitted or None, does not automatically save the results. Use `save_simulation` to save manually.

If output_dir is a function (or callable class), it must accept a dict of overridden parameters and return a string.

The output_dir (or the result of calling that function) is created if it does not already exist.

post_run must be either None or a function of three variables: data, run_id, params. Here, params is a dict of overriden parameters (if any).

sweep may contain multiple variables, in which case all possible combinations will be tried. This can get large very quickly.
e.g.
  run(simulation, out_dir, sweep = {'iapp': [1, 5, 10], 'gsyn': [.01, .02]})
will run $3 \times 2 = 6$ simulations.

sweep may also be a list of dicts to sweep, which can be used for doing parameter sweeps on multiple base parameter sets simultaneously. `parameters_from_dir` is helpful for setting these up.

run_to(t)

Run the simulation until time t. See also `advance`.

save_data(data=None)

Append to or get the current list of saved data. This is data that is saved in every sdat file that might not normally be saved. This list is empty unless set via this function.

set_initial_time(t)

Set the initial time for the simulation to t.

set_max_advance_time(t)

Sets the maximum time a simulation will advance in one step. Larger advances will be automatically decomposed into a series of advances under this limit.

Smaller values potentially reduce memory needs but increase run time.

Applies only to the current simulation.

It is not typically necessary to manually set this value.

set_run_id(run_id)

Sets the run_id, the default filename. For simulation protocols controlled via `run`, the original default is the random seed.

set_simulation(sim)

Used for running multiple simulations with the same initial portion.

Pass in either a Simulation or a Full instance.

Warning:

In most cases, you will want to pass in a copy of a Simulation, not the original.

See also `copy_simulation`, `fork` and `end_fork`.

value(var, val=None)

Return value of the simulation parameter var if it exists, else val.

A.3 Files

create_subdirectories(base, *subdirs)

Create subdirectories (one or many) in the directory base.

If the subdirectory already exists, it will not be modified. If the base directory path does not exist, it will be created as well. This function is useful when analyzing results.

Examples:
  create_subdirectories('base_folder', 'results')
  create_subdirectories('base_folder', 'potentials', 'rasters')

default_filename(extension=None, include_run_id=True)

Return a filename based on the parameter sweep.

Parameters:
  extension – a filename extension to append
  include_run_id – should we include the run_id in the name?

Names are generated by writing the parameter names and values separated by underscores, then the run_id (after an underscore, if include_run_id is True).

Example:

If the run command sweeps or overrides gna = 2 and gk = 1 and run_id = 0, default_filename() returns 'gk_1_gna_2_0' or 'gna_2_gk_1_0' (no guarantee is made about the order). Likewise, default_filename('csv') returns 'gk_1_gna_2_0.csv' or 'gna_2_gk_1_0.csv'

default_filename_rule(filename)

Takes a path of the form a/b/c.sdat, returns a_b_c.pdf. Useful for giving meaningful names to images corresponding to data saved during parameter sweeps with the `run` function.

export_to_csv(filename, *args, **kwargs)

Save lists to csv file.

Parameters:
  filename – output csv file
  list1, list2, ... – lists to output to the file
  separator – separator to use between data points.

Note:

filename can be a string name or a file object.
If filename is a file object, will leave open when done.

If no lists are passed, no file will be created.

separator defaults to ', '

Examples:
http://www.gnuplot.info/ export_to_csv('results', x, y)
export_to_csv('results', x, y, separator=' ')

load_simulation(filename)

Load the simulation stored in filename. If filename refers to a directory, loads the file 0.sdat from that directory.

num_sims(dir_name)

Return the number of simulations in the directory dir_name.

parameters_from_dir(directory)

Returns the parameters for files saved in the given directory.

Assumes the filenames were chosen according to default_filename.

Ignores extensions, assuming at least one character of the extension is not a number.

save_simulation(output_dir, filename=None, save_type=full)

Manually saves the current simulation. Useful for forks, saving intermediate results, and simulations initiated without using `run`.

If filename is None, uses current run_id.

A.4 Graphics

Snnet imports everything in `matplotlib.pyplot`, if installed, but those functions are not part of snnet and not covered here.

gnuplot(filename, *data, **figure_properties)

Use gnuplot to plot a line graph.

data consists of the x and y values to plot followed by any line style information.
Supported line style parameters:
  line type – solid, dash, dot, dot-dash, or a number
  line width – thin, normal, thick, or a number
  color – line color
  axis – which axis to plot against: 1 (default) or 2
  legend – identifying string for the legend

Supported figure_properties:
  x_lim
  y_lim
  x2_lim
  y2_lim
  x_label
  y_label
  grid – True or False
  title
  legend – anything that can go after 'set key' in gnuplot
  use_color – can be True (default) or False

Example:

```
gnuplot('output.pdf', x1, y1, 'legend', 'graph1', x2, y2, x3, y3, title='3 graphs')
```

Note:

Use the legend figure_property to set the location, e.g. 'left bottom'

If no legends are set, the key is not displayed.

Requires `gnuplot` and `convert` to be installed.

make_rasters(data_dir, image_dir= '.', plot_types=None, shading=None, shading_color=[.5, 1, 1], show_field_potentials=True, filename_rule=default_filename_rule, image_processing=None, num_parallel=1)

Make raster diagrams for all sdat files in a directory (and its subdirectories).

Parameters:
  data_dir – the directory with the data
  image_dir – the directory to put the image files
  plot_types – the types of neurons to plot (defined by filename)
  shading – list of time intervals to shade (or None)
  shading_color – rgb color triplet for shading
  show_field_potentials – do we plot the field potentials ignored if f.p. data not in data
  filename_rule – a function that takes a path in data_dir, returns a filename for the image
  image_processing – a function to run on the figure before it is saved (or None)
  num_parallel – maximum number of files to work on simultaneously

Note: If plot_types is None, then plots for all types of neurons.

Requires `matplotlib`.

raster(data, plot_types=None, shading=None, shading_color=[.5, 1, 1], show_field_potentials=True)

Plot a raster diagram of activity.

Parameters:
  data – the SimData object (or filename containing SimData) to plot
  plot_types – the types of neurons to plot (defined by filename)
  shading – list of time intervals to shade (or None)
  shading_color – rgb color triplet for shading
  show_field_potentials – do we plot the field potentials ignored if f.p. data not in data

Requires `matplotlib`.

A.5 Noise

exponentially_decaying_noise(num=1, firing_rate=5, decay_rate=2, delay=0, max_value=1)

Return num sources of exponentially decaying noise.

Parameters:
  num – how many noise sources to create

firing_rate – firing rate in Hz (or function)
decay_rate – in ms
delay – period synapse stays elevated before decaying
max_value – the peak value of the artificial synapse

periodic(freq)

Returns a callable object corresponding to periodic firing at a given frequency.

square_wave_noise(num=1, firing_rate=5, duration=1, max_value=1)

Return num sources of square wave noise.

Parameters:
num – how many noise sources to create
firing_rate – firing rate in Hz (or function)
duration – in ms
max_value – the peak value of the artificial synapse

A.6 Miscellaneous

name_time(name, toffset=0)

Assign a name to a particular time. This time can then be located by the analysis functions by name.

Parameters:
name – the name to assign
toffset – the offset from the current simulation time

overrides()

Return the parameter overrides specified in the run command, if any

rand(number)

Get a random number using the snnet syntax.

Examples:

rand('[3 : 5]') – returns a uniform r.v. float between 3 and 5
rand('3 [2]') – returns a normal r.v. float with mean 3, std dev 2
rand('5 [1%]') – normal r.v. float, mean 5, std dev 1% of mean (.05)

Random numbers are generated via Python's random module, which uses the Mersenne Twister algorithm (in Python 2.3+).

random_instance()

Return the random instance used by snnet.

Example:

r = random_instance()
r.uniform(5, 12) – uniform r.v. between 5 and 12
r.betavariate(alpha, beta) – beta r.v.

The return value is an instance of random.Random. See the Python documentation for more information.

Simulations initiated via run have the random seed set to the run_id.

simplify(expression)

Simplifies the expression the same way a snnet file would be simplified. Uses `sympy`, if available.

Example:
`simplify('1 + 1')` – returns 2

total_volume(volume=None)

Set or get the total volume (extracellular + cellular). This is used in experiments where cells or organelles have changing volume.

Appendix B

Data API

SimData objects are returned via snnet's `load_simulation`, `simulation_data`, and sometimes `run` functions, defined above. Depending on their save type, they will support some or all of the following properties and methods.

If `data` is a SimData object, and you wish to use the method `field_potential` defined below on the cells named `cells` and store the result in the variable `fp`, call it via

```
fp = data.field_potential(cells)
```

If the save type stores the values of state variables, then one can access (for example) variable `v` of neuron 10 from the SimData object `data` via either `data.v10` or `data['v', 10]`. The first option cannot be used with variable names ending in numbers, because that would be potentially ambiguous. The second option also has the advantage of allowing programmatic control, e.g. via python's `for` loops.

State variables for global dynamics are accessed similarly, except there is no index to specify.

B.1 Methods and Properties

active(times, cells=None, activity_window=100)

Return the number of specified cells active at times.

Parameters:
  times – list of times

cells – the cells to check (indices, cell type, or cell group)
activity_window – how long ago a cell may have fired and still be considered active

Note: If cells is None, returns the total number of active cells.

compartments

A list of the subcompartments belonging to each neuron (or organelle).

connections

A list of lists of each neuron's pre-synaptic input sources.

For example, if neuron 1 received input from neurons 2, 5, and 7, then `connections[1]` would return `[2, 5, 7]`.

cv(cells=None, time_range=(-inf, inf), combine_cells=False)

Returns the coefficient of variation (CV) of the interspike intervals of cells.

Note:

Returns the CVs of all cell types if cells is None.

Results limited to those spikes occurring in the interval time_range.

If combine_cells is False, then returns a list of CVs, one for each neuron. If True, returns the average CV of the selected cells.

data

A list of everything that was saved (in order) via `save_data`.

expression(var, id, t, global_dynamics=False)

Return the algebraic expression for a given variable at a given time.

Parameters:
var – the name of the variable
id – the cell (or global_dynamics) id

t – the time
global_dynamics – set True to get global dynamics id (instead of neuron)

field_potential(cells=None)

Return the average membrane potential of a group of cells.

Parameters:
cells – can be a named group, a filename, or a list of indices

Note:

If cells is None, returns the average membrane potential of all the cells.

Since this method returns the field potential for an arbitrary group of cells, it requires a save type that saves all membrane potentials. In the standard distribution, `full` is the only such save type. Use the `field_potentials` property with `enhanced` and any other user-defined save type to get the field potential for all cells of a given cell type.

field_potentials

A dict of the field potentials, keyed by cell type.

get_indices(cells=None)

Return the indices of the cells specified.

Parameter:
cells – a list of indices or a string naming a cell_type or cell_group

Note: If cells is None, returns all indices.

index(var, neuron_index=None)

Returns the index in the list of state vectors of variable var from the neuron with index neuron_index. If neuron_index is none, returns the index of the global variable var. This method is not typically needed.

isi(cells=None, time_range=(-inf, inf), combine_cells=False)

Returns the interspike intervals (ISIs) of cells.

Note:

Returns the ISIs of all cell types if cells is None.

Results limited to those spikes occurring in the interval time_range.

If combine_cells is False, then returns a list of lists of ISIs, one list for each neuron. If True, returns a single list of ISIs covering all of cells.

potential(i)

Return the membrane potential time course for neuron i.

run_time

The end time of the simulation.

spike_times

A list of lists of spike times for each cell.

state_variables()

Return a list of accessible state variables.

t

The vector of times at which state variables were computed.

time(t)

Return the time corresponding to t. t may be either a numeric value or the name of a saved time point.

types

A list of the cell types used in the simulation. Cell types are identified based on the string used to create them in the `neurons` function.