PyNN: a unified interface for neuronal network simulators

Andrew Davison
Paris-Saclay Institute of Neuroscience, CNRS

CNS*2020 Online
22nd July 2020

SenseLab

ModelDB

Find Models by Simulation Environment

Click on a link to show a list of models implemented in that simulation environment or programming language.

Simulation Environment	Homepage	Number of models
BioPAX (web link to model)		1
Brian		4
C or C++ program		34
C or C++ program (web link to model)		19
CONTENT		1
CSIM		1
CSIM (web link to model)		3
CalC Calcium Calculator		1
CalC Calcium Calculator (web link to model)		7
Catacomb (web link to model)		1
CellExcite (web link to model)		1
CellML		0
CellML (web link to model)		1
Chemesis		2
Dynamics Solver		1
Emergent/PDP++		3
FORTRAN		4
FORTRAN (web link to a model)		1
GNUstep NeXTStep/OpenStep		1
Genesis		13
Genesis (web link to model)		7
IChMASCOT		0
IGOR Pro		3
IonChannelLab		0
Java		5
Java (web link to model)		2
KInNeSS (web link to model)		1
L-Neuron		0
MATLAB		57
MATLAB (web link to model)		34
MCell		1
MOOSE/PyMOOSE (web link to method)		1
MVASpike		1
MadSim		1
NCS		1
NEST (formerly BLISS/SYNOD)		2
NEURONPM (web link to tool)		2
NSL		0
Neosim		0
Network		1
NeuGen		0
Neuron		251
Neuron (web link to model)		14
NeuronC		0
NeuronetExperimenter (web link to model)		1
Octave		1
PCSIM		1
PSpice		2
Pascal (web link to model)		1
Pascal/Delphi		2
PyNN		2
Python		5
Python (web link to model)		1
QBasic/QuickBasic/Turbo Basic		2
QuB		1
R (web link to model)		1
SABER		1
SBML (web link to model)		1
SNNAP		21
SciLab		1
Scilab (web link to model)		2
Simulink		6
Sapice Symbolic SPICE		1
Surf-Hippo		0
Synthesis		0
Topographica		0
Topographica (web link to model)		1
Virtual Cell (web link to model)		3
XML (web link to model)		3
XNBC		0
XPP		50
XPP (web link to model)		7
neuroConstruct (web link to model)		1
panplex		2

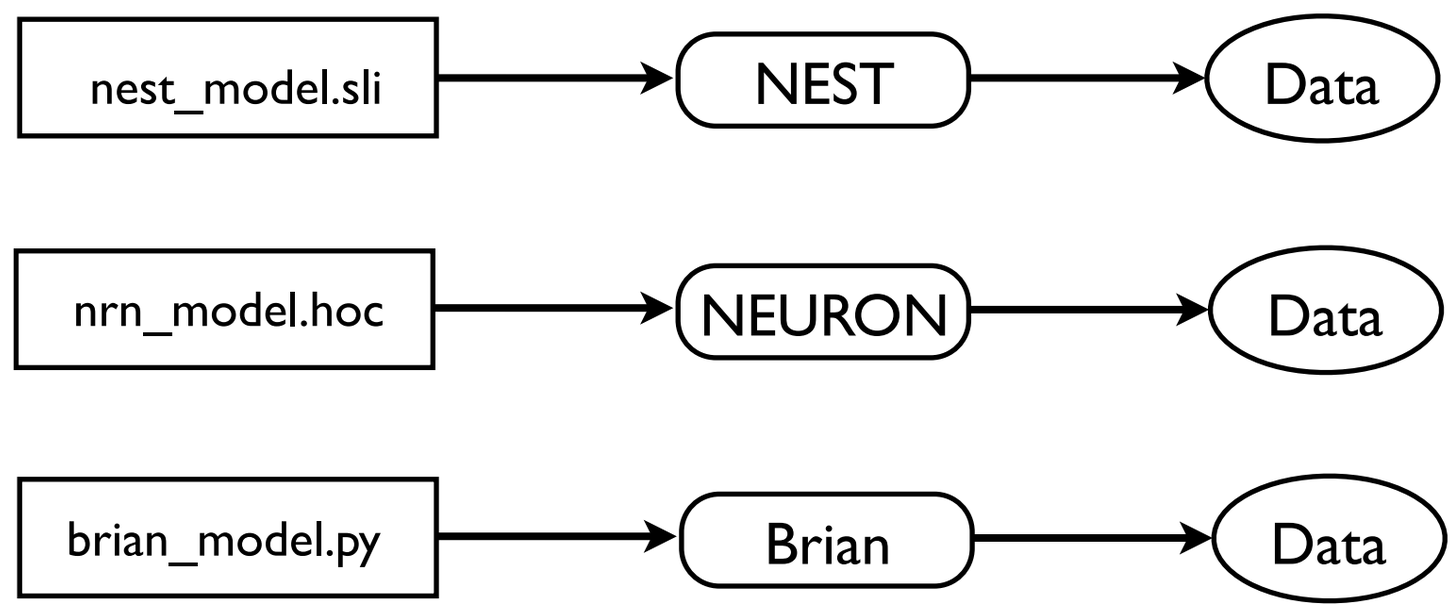
Verification, cross-checking and model sharing

A single researcher or single lab cannot hope to model everything of interest.

Need to build on previous work: re-use and extend existing models.

But almost all models only run on a single simulator, and translation is challenging and time consuming.

Hence not reusable or testable.



Simulator diversity: problem and opportunity

Cons

- Considerable difficulty in translating models from one simulator to another...
- ...or even in understanding someone else's code.
- This:
  - impedes communication between investigators,
  - makes it harder to reproduce other people's work,
  - makes it harder to build on other people's work.

Pros

- Each simulator has a different balance between efficiency, flexibility, scalability and user-friendliness → can choose the most appropriate for a given problem.
- Any given simulator is likely to have bugs and hidden assumptions, which will be revealed by cross-checking results between different simulators → greater confidence in correctness of results.

Having your cake and eating it

Simulator-independent environments for developing neuroscience models:

- keep the advantages of having multiple simulators or hardware devices
- but remove the translation barrier.

Three (complementary) approaches:

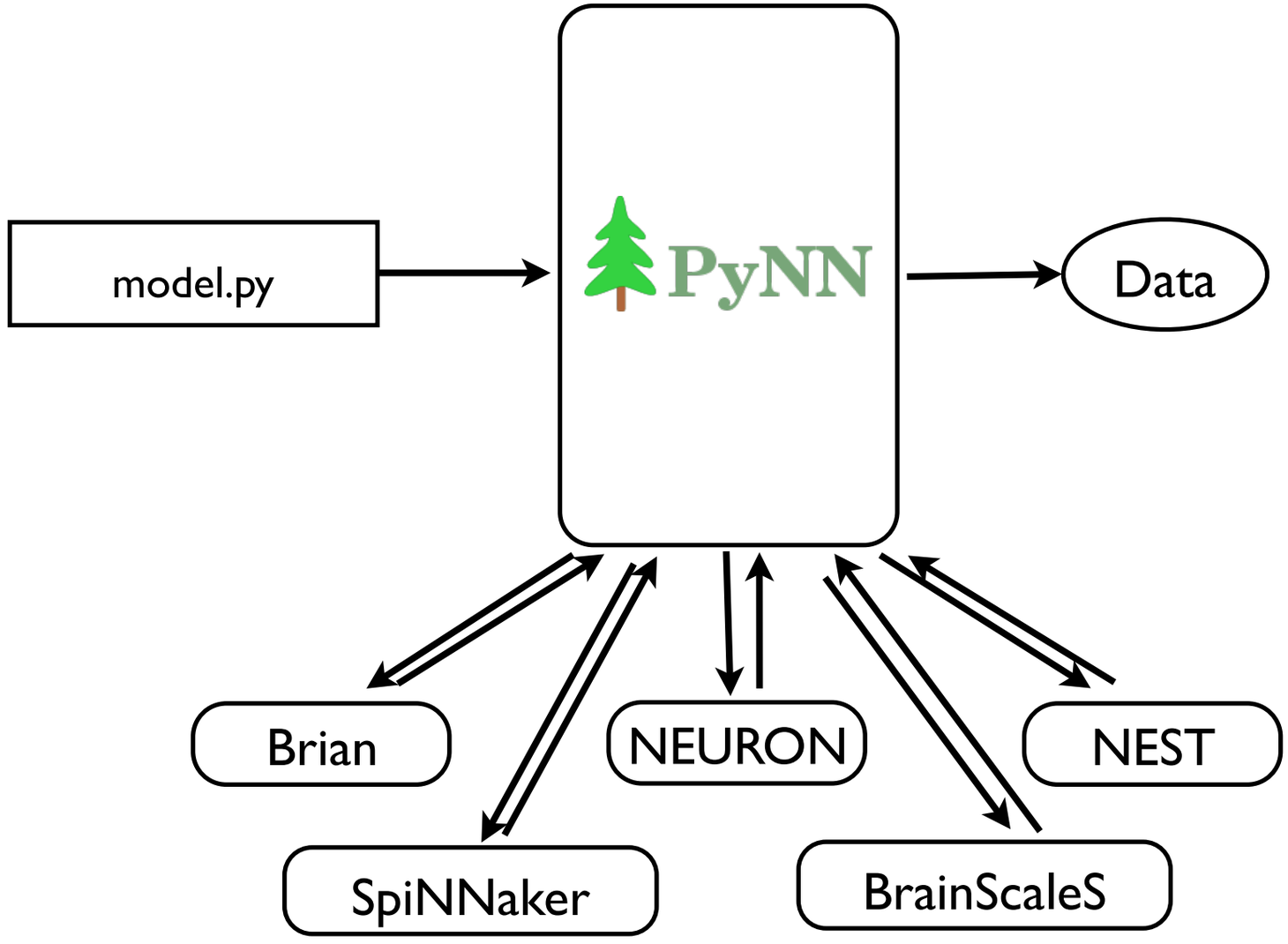
- GUI (e.g. neuroConstruct)
- XML/JSON-based language (e.g. NeuroML, NineML)
- interpreted language (e.g. Python)

Cawan Cake by Nono Fara http://www.flickr.com/photos/n-o-n-o/3280580620/

A common API for neuronal network modelling

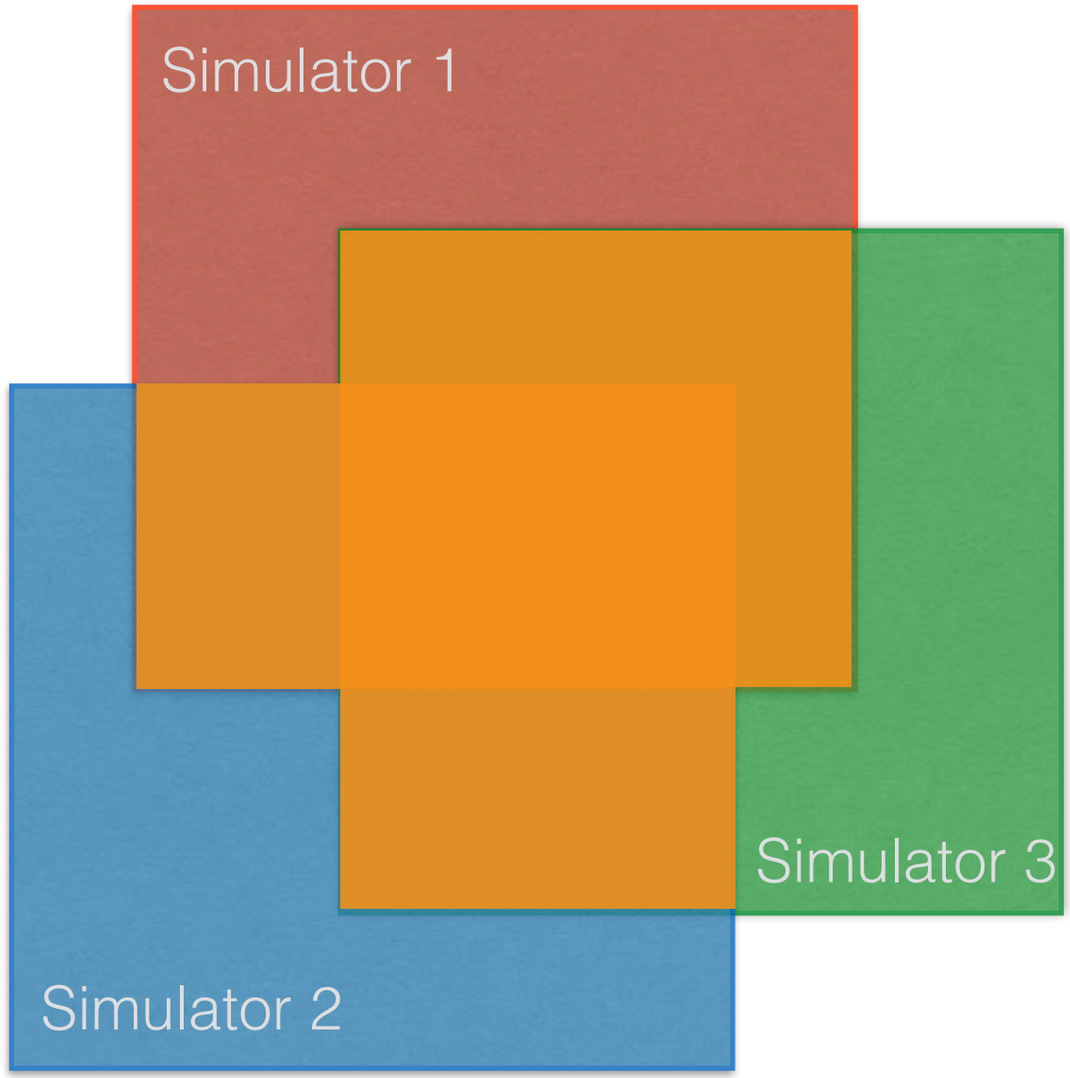
Goals:

- facilitate model sharing and reuse
- simplify validation of simulation results
- provide a common platform on which to build other tools (stimulation, analysis, visualization, GUIs)
- provide a more powerful API for neuronal network modelling (save scientist time)
- hide complexity of parallelization from user (increased computational efficiency without decreased scientist efficiency)



http://neuralensemble.org/PyNN/

Capabilities



http://neuralensemble.org/PyNN/

Installation

```
$ pip install PyNN
```

Documentation

http://neuralensemble.org/PyNN/

Licence

CeCILL (GPL-equivalent)

Mailing list

https://groups.google.com/forum/#!forum/neuralensemble

```
sim.setup(timestep=0.1)

cell_parameters = {"tau_m": 12.0, "cm": 0.8, "v_thresh": -50.0}
pE = sim.Population(2e4, sim.IF_cond_exp(**cell_parameters))
pI = sim.Population(5e3, sim.IF_cond_exp(**cell_parameters))
all = pE + pI
input = sim.Population(100, sim.SpikeSourcePoisson(
          rate=random.RandomDistribution("normal", (10.0, 2.0))))
all.inject(sim.NoisyCurrentSource(mean=0.1, stdev=0.01))

weight_distr = random.RandomDistribution("uniform", (0.0, 0.1))
DDPC = sim.DistanceDependentProbabilityConnector
connector = DDPC("exp(-d**2/400.0)", weights=weight_distr,
                 delays="0.5+0.01d")
depressing = sim.TsodyksMarkramMechanism(U=0.5, tau_rec=800.0)

exc = sim.Projection(pE, all, connector,
            synapse_type="depressing", receptor_type="excitatory")
inh = sim.Projection(pI, all,
                     connector, receptor_type="inhibitory")
```

```
import pyNN.nest as sim

sim.setup(timestep=0.1)

cell_parameters = {"tau_m": 12.0, "cm": 0.8, "v_thresh": -50.0}
pE = sim.Population(2e4, sim.IF_cond_exp(**cell_parameters))
pI = sim.Population(5e3, sim.IF_cond_exp(**cell_parameters))
all = pE + pI
input = sim.Population(100, sim.SpikeSourcePoisson(
            rate=random.RandomDistribution("normal", (10.0, 2.0))))
all.inject(sim.NoisyCurrentSource(mean=0.1, stdev=0.01))

weight_distr = random.RandomDistribution("uniform", (0.0, 0.1))
DDPC = sim.DistanceDependentProbabilityConnector
connector = DDPC("exp(-d**2/400.0)", weights=weight_distr,
                 delays="0.5+0.01d")
depressing = sim.TsodyksMarkramMechanism(U=0.5, tau_rec=800.0)

exc = sim.Projection(pE, all, connector,
            synapse_type="depressing", receptor_type="excitatory")
inh = sim.Projection(pI, all,
                     connector, receptor_type="inhibitory")
```

```
import pyNN.neuron as sim

sim.setup(timestep=0.1)

cell_parameters = {"tau_m": 12.0, "cm": 0.8, "v_thresh": -50.0}
pE = sim.Population(2e4, sim.IF_cond_exp(**cell_parameters))
pI = sim.Population(5e3, sim.IF_cond_exp(**cell_parameters))
all = pE + pI
input = sim.Population(100, sim.SpikeSourcePoisson(
            rate=random.RandomDistribution("normal", (10.0, 2.0))))
all.inject(sim.NoisyCurrentSource(mean=0.1, stdev=0.01))

weight_distr = random.RandomDistribution("uniform", (0.0, 0.1))
DDPC = sim.DistanceDependentProbabilityConnector
connector = DDPC("exp(-d**2/400.0)", weights=weight_distr,
                 delays="0.5+0.01d")
depressing = sim.TsodyksMarkramMechanism(U=0.5, tau_rec=800.0)

exc = sim.Projection(pE, all, connector,
            synapse_type="depressing", receptor_type="excitatory")
inh = sim.Projection(pI, all,
                     connector, receptor_type="inhibitory")
```

```
import pyNN.brian as sim

sim.setup(timestep=0.1)

cell_parameters = {"tau_m": 12.0, "cm": 0.8, "v_thresh": -50.0}
pE = sim.Population(2e4, sim.IF_cond_exp(**cell_parameters))
pI = sim.Population(5e3, sim.IF_cond_exp(**cell_parameters))
all = pE + pI
input = sim.Population(100, sim.SpikeSourcePoisson(
            rate=random.RandomDistribution("normal", (10.0, 2.0))))
all.inject(sim.NoisyCurrentSource(mean=0.1, stdev=0.01))

weight_distr = random.RandomDistribution("uniform", (0.0, 0.1))
DDPC = sim.DistanceDependentProbabilityConnector
connector = DDPC("exp(-d**2/400.0)", weights=weight_distr,
                 delays="0.5+0.01d")
depressing = sim.TsodyksMarkramMechanism(U=0.5, tau_rec=800.0)

exc = sim.Projection(pE, all, connector,
            synapse_type="depressing", receptor_type="excitatory")
inh = sim.Projection(pI, all,
                     connector, receptor_type="inhibitory")
```

```
import pyNN.spiNNaker as sim

sim.setup(timestep=0.1)

cell_parameters = {"tau_m": 12.0, "cm": 0.8, "v_thresh": -50.0}
pE = sim.Population(2e4, sim.IF_cond_exp(**cell_parameters))
pI = sim.Population(5e3, sim.IF_cond_exp(**cell_parameters))
all = pE + pI
input = sim.Population(100, sim.SpikeSourcePoisson(
            rate=random.RandomDistribution("normal", (10.0, 2.0))))
all.inject(sim.NoisyCurrentSource(mean=0.1, stdev=0.01))

weight_distr = random.RandomDistribution("uniform", (0.0, 0.1))
DDPC = sim.DistanceDependentProbabilityConnector
connector = DDPC("exp(-d**2/400.0)", weights=weight_distr,
                 delays="0.5+0.01d")
depressing = sim.TsodyksMarkramMechanism(U=0.5, tau_rec=800.0)

exc = sim.Projection(pE, all, connector,
            synapse_type="depressing", receptor_type="excitatory")
inh = sim.Projection(pI, all,
                     connector, receptor_type="inhibitory")
```

Overview of the PyNN API

- neuron and synapse models
- populations
- connectivity
- recording & data handling

Neuron and synapse models

- “standard” models
- native models
- NineML and NeuroML models

Standardized neuron and synapse models

```
>>> sim.list_standard_models()
['IF_cond_alpha', 'HH_cond_exp', 'IF_curr_exp', 'IF_cond_exp',
'EIF_cond_exp_isfa_ista', 'SpikeSourceArray',
'IF_cond_exp_gsfa_grr', 'IF_facets_hardware1',
'SpikeSourcePoisson', 'EIF_cond_alpha_isfa_ista',
'IF_curr_alpha']
```

```
cell_type = sim.IF_cond_exp(tau_m=12.0, cm=0.8, ...)

synapse_type = sim.TsodyksMarkramSynapse(U=0.04, tau_rec=500.0)
```

- For a given cell model:
  - *identify* (NEST, SpiNNaker, BrainScaleS) or *build* (NEURON, Brian) a model with the desired behaviour
  - map model name and parameter names and units
  - (test that each simulator gives the same results)

Standardized neuron and synapse models

Example: Leaky integrate-and-fire model with fixed firing threshold, and current-based, alpha-function synapses.

PyNN		NEURON		NEST		PCSIM	
IF_curr_alpha		StandardIF (type="current", shape="alpha")		iaf_psc_alpha		LIFCurrAlphaNeuron	
v_rest	mV	v_rest	mV	E_L	mV	Vresting	V
v_reset	mV	v_reset	mV	V_reset	mV	Vreset	V
cm	nF	CM	nF	C_m	pF	Cm	F
tau_m	ms	tau_m	ms	tau_m	ms	taum	s
tau_refrac	ms	t_refrac	ms	t_ref	ms	Trefract	s
tau_syn_E	ms	tau_syn_E	ms	tau_syn_ex	ms	TauSynExc	s
tau_syn_I	ms	tau_syn_IU	ms	tau_syn_in	ms	TauSynInh	s
v_thresh	mV	v_thresh	mV	V_th	mV	Vthresh	V
i_offset	nA	i_offset	nA	I_e	pA	Iinject	A

Native neuron and synapse models

- can wrap any model provided by/buildable with a given simulator to use with PyNN:

```
from pyNN.nest import native_cell_type, native_synapse_type

ht_neuron = native_cell_type(“ht_neuron”)
poisson = native_cell_type(“poisson_generator”)

cell_type = ht_neuron(Tau_m=20.0)
input_type = poisson(rate=200.0)

stdp = native_synapse_type(“stdp_synapse”)

synapse_type = stdp(Wmax=50.0, lambda=0.015)
```

Native neuron and synapse models

```
from nrnutils import Mechanism, Section
from pyNN.neuron import NativeCellType

class SimpleNeuron(object):

    def __init__(self, **params):
        hh = Mechanism('hh', gl=params['g_leak'], el=-65,
                       gnabar=params['gnabar'], gkbar=params['gkbar'])
        self.soma = Section(L=30, diam=30, mechanisms=[hh])
        self.soma.add_synapse('ampa', 'Exp2Syn', e=0.0, tau1=0.1,
                              tau2=5.0)
                .
                .

class SimpleNeuronType(NativeCellType):
    default_parameters = {'g_leak': 0.0002, 'gkbar': 0.036,
                          'gnabar':  0.12}
    default_initial_values = {'v': -65.0}
    recordable = ['soma(0.5).v', 'soma(0.5).ina']
    model = SimpleNeuron

cell_type = SimpleNeuronType(g_leak=0.0003)
```

More flexible cell, synapse and plasticity models

...describe model in simulator-independent way
then generate code for each simulator

More flexible cell, synapse and plasticity models

...describe model in simulator-independent way
then generate code for each simulator

Model description languages:

More flexible cell, synapse and plasticity models

...describe model in simulator-independent way
then generate code for each simulator

Model description languages:

- NeuroML v2 / LEMS - future work
- NineML - supported for NEURON, NEST
- NESTML - future work

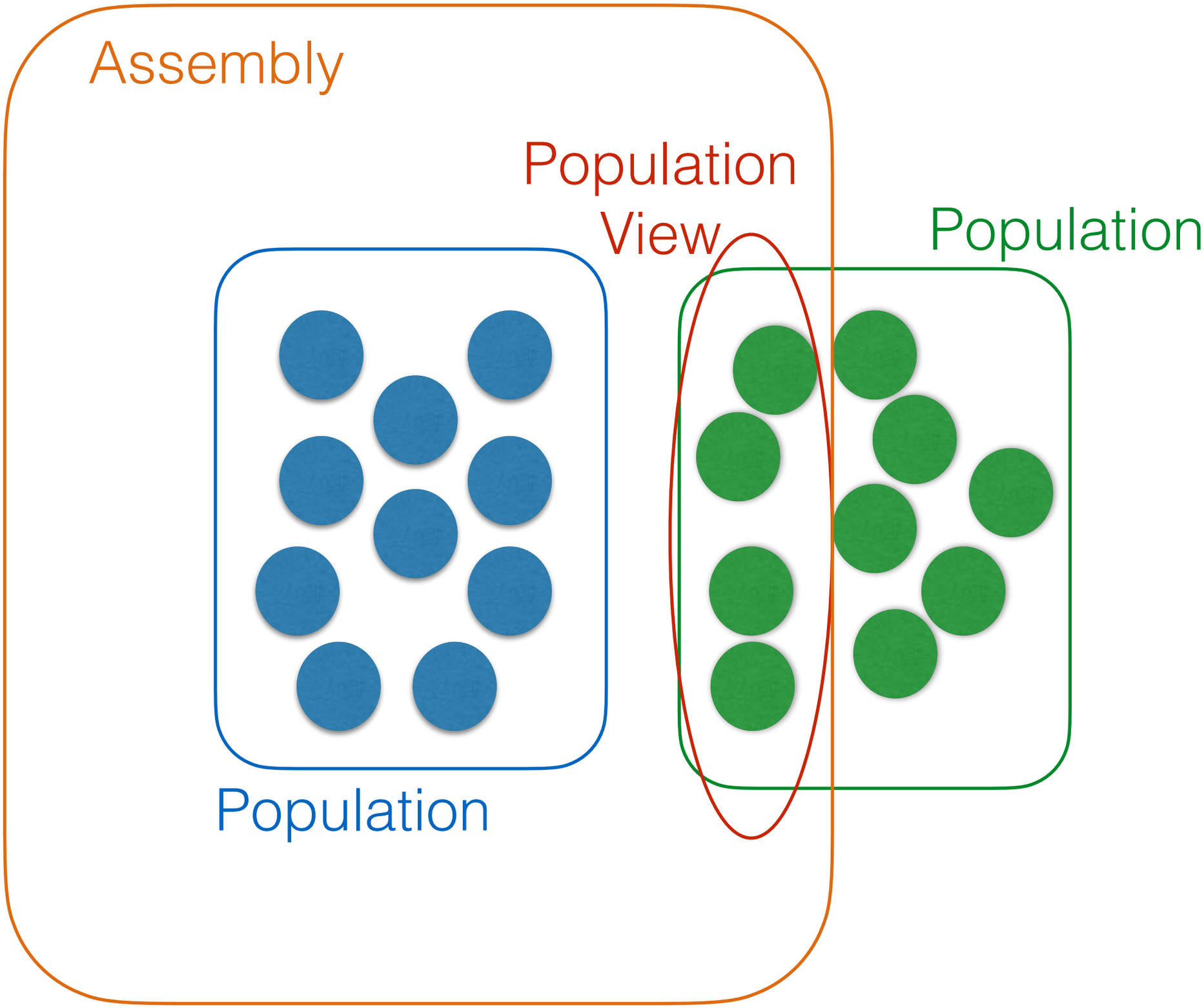
NineML cell, synapse and plasticity models

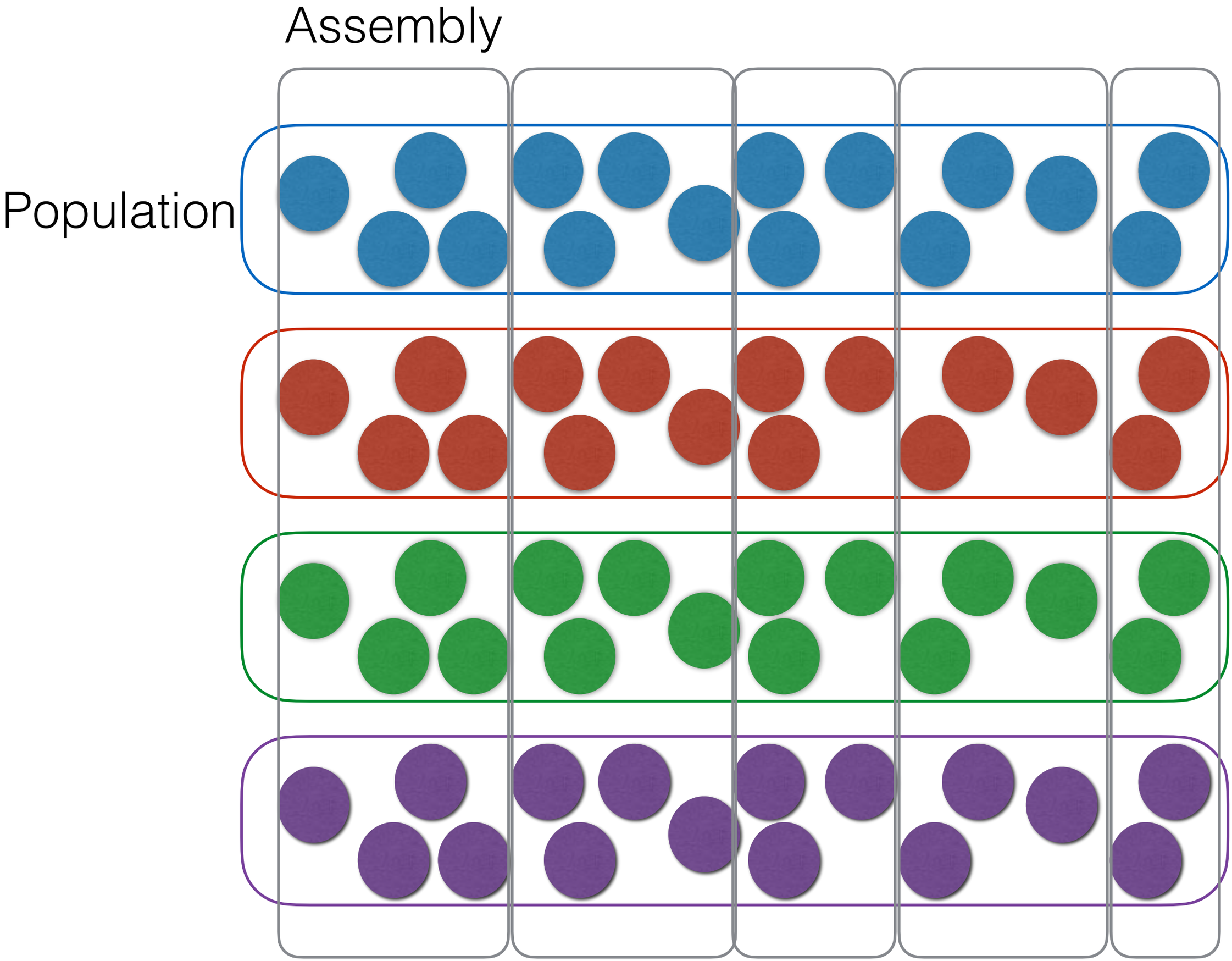
```
cell_type = nineml_cell_type("iaf_3coba",
                             "iaf.xml",
                             AMPA="coba_syn.xml",
                             NMDA="nmda_syn.xml",
                             GABAA="coba_syn.xml")

parameters = {
    'iaf.cm': 1.0, 'iaf.gl': 50.0, 'iaf.taurefrac': 5.0,
    'iaf.vrest': -65.0, 'iaf.vreset': -65.0, 'iaf.vthresh': -50.0,
    'AMPA.tau': 2.0, 'GABAa.tau': 5.0, 'AMPA.vrev': 0.0, 'GABAa.vrev': -70.0,
    'nmda.taur': 3.0, 'nmda.taud': 40.0, 'nmda.gmax': 1.2, 'nmda.E': 0.0,
    'nmda.gamma': 0.062, 'nmda.mgconc': 1.2, 'nmda.beta': 3.57
}

p = sim.Population(size, celltype_cls, parameters)
```

- allows multi-simulator use of arbitrary cell models, no more fixed standard library
- based on Python lib9ML

Overview of the PyNN API

- neuron and synapse models
- populations
- connectivity
- recording & data handling


Assembly
Population View
Population
Population


Assembly
Population

Populations

```
structure = RandomStructure(boundary=Sphere(radius=200.0))

cells = Population(100, thalamocortical_type,
                   structure=structure,
                   initial_values={'v': -70.0},
                   label="Thalamocortical neurons")


view = cells[:80]          # the first eighty neurons
view = cells[::2]          # every second neuron
view = cells[45, 91, 7]    # a specific set of neurons
view = cells.sample(50)    # 50 neurons at random


layer4 = spiny_stellates + l4_interneurons  # an Assembly
```

Overview of the PyNN API

- neuron and synapse models
- populations
- connectivity
- recording & data handling

Connectivity

```
rng = NumpyRNG(seed=64754)
sparse_connectivity = FixedProbabilityConnector(0.1, rng=rng)
weight_distr = RandomDistribution('normal', [0.01, 1e-3], rng=rng)
facilitating = TsodyksMarkramSynapse(U=0.04, tau_rec=100.0,
                                     tau_facil=1000.0,
                                     weight=weight_distr,
                                     delay=lambda d: 0.1+d/100.0)
space = Space(axes='xy')
inhibitory_connections = Projection(pre, post,
                                    connector=sparse_connectivity,
                                    synapse_type=facilitating,
                                    receptor_type='inhibitory',
                                    space=space,
                                    label="inhibitory connections")
```

Connectivity

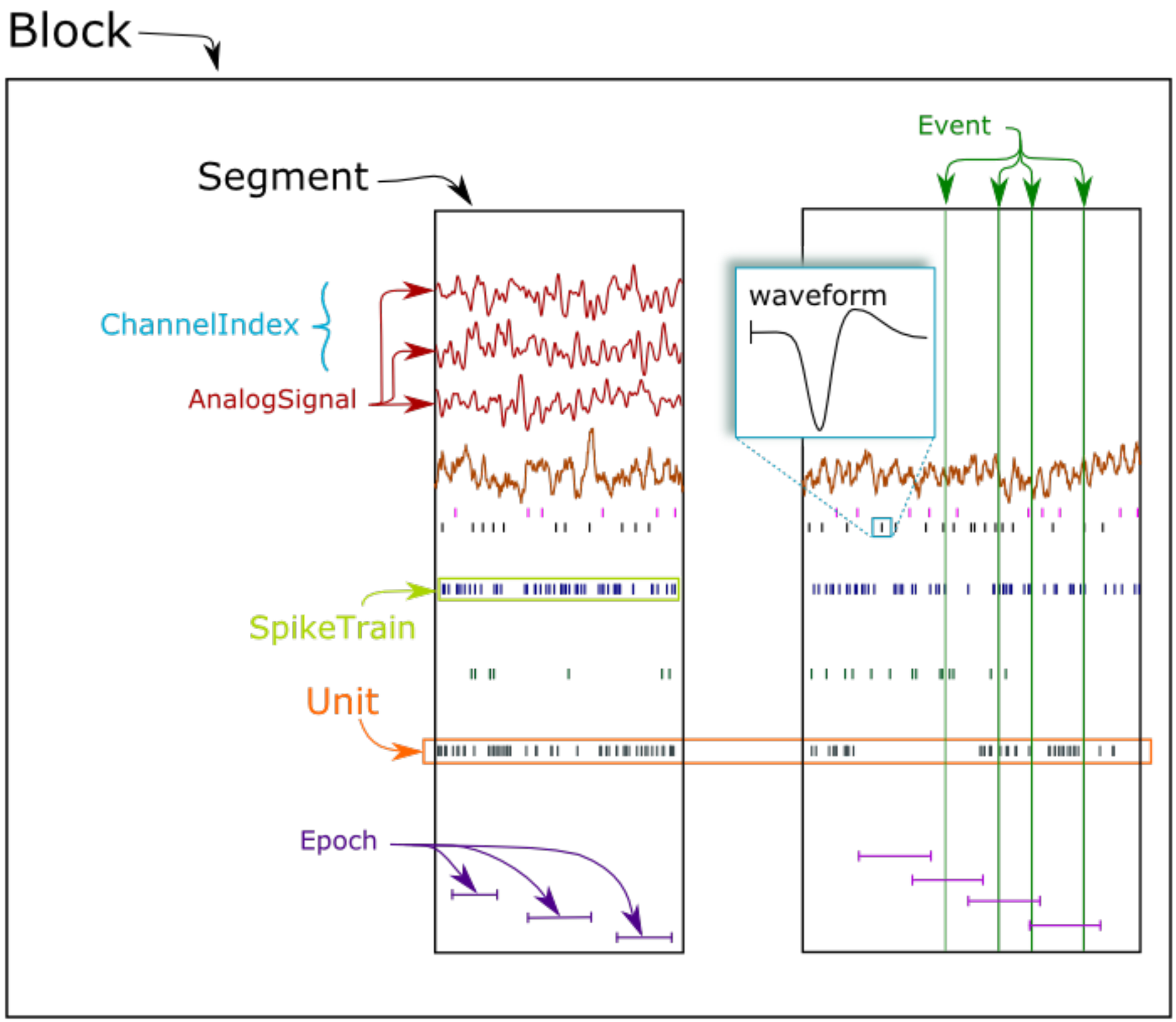
Each connectivity/wiring algorithm encapsulated in a class.

```
OneToOneConnector
AllToAllConnector
FixedProbabilityConnector
DistanceDependentProbabilityConnector
FixedNumberPreConnector
FixedNumberPostConnector
FromListConnector
FromFileConnector
CSAConnector
SmallWorldConnector
```

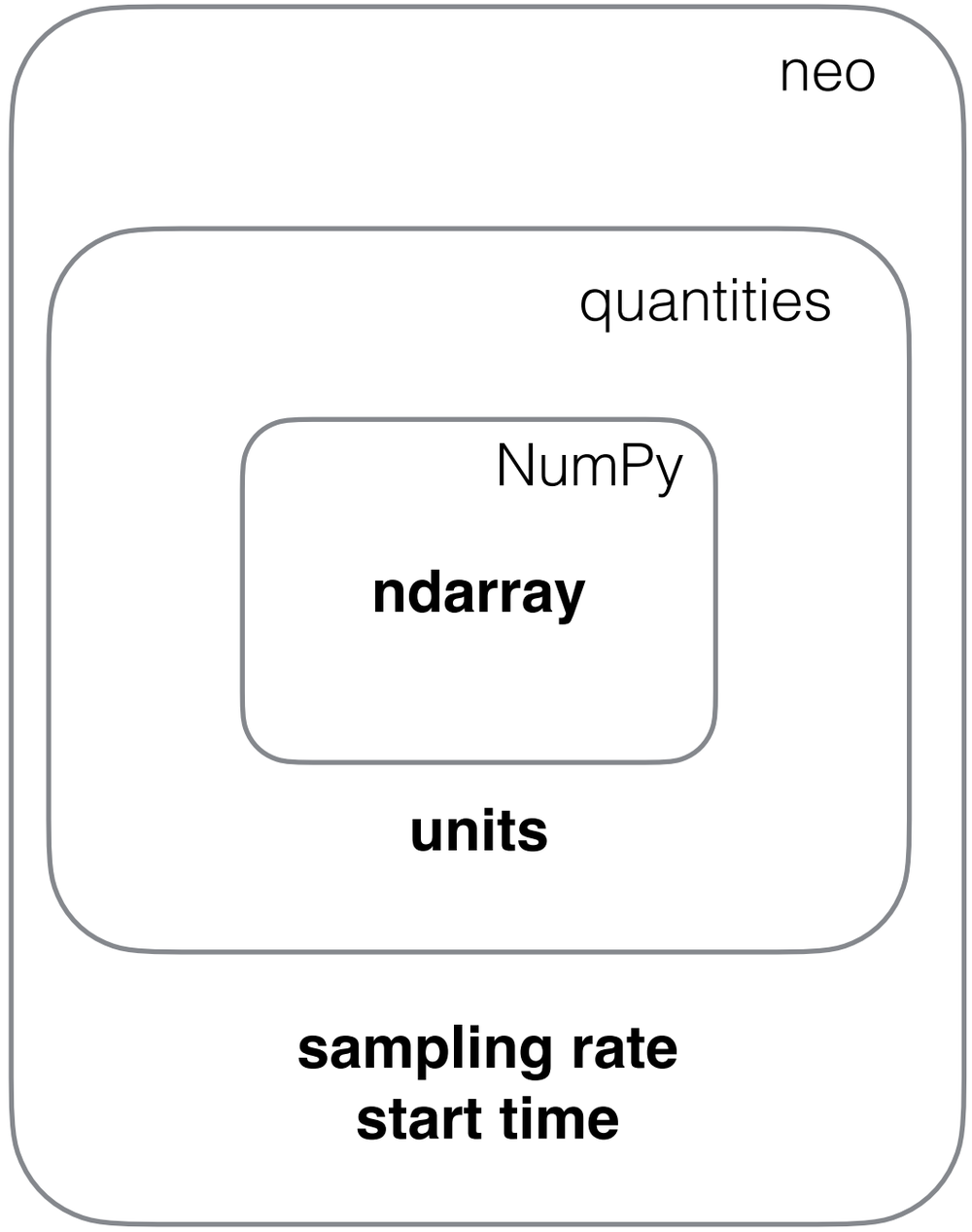
Fairly straightforward to write your own.

Overview of the PyNN API

- neuron and synapse models
- populations
- connectivity
- recording & data handling



http://neuralensemble.org/neo


neo
quantities
NumPy
ndarray
units
sampling rate
start time

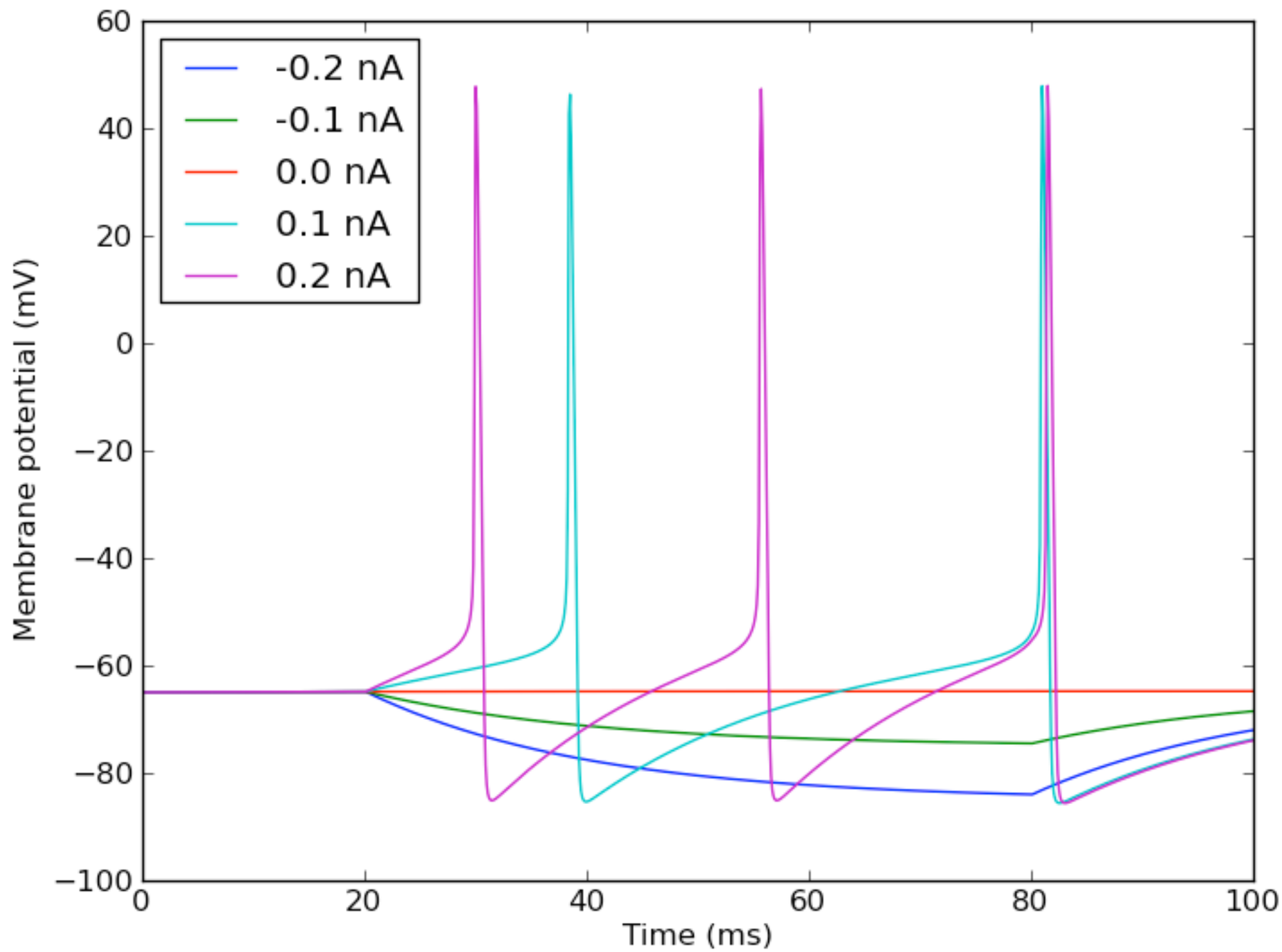
Data-handling

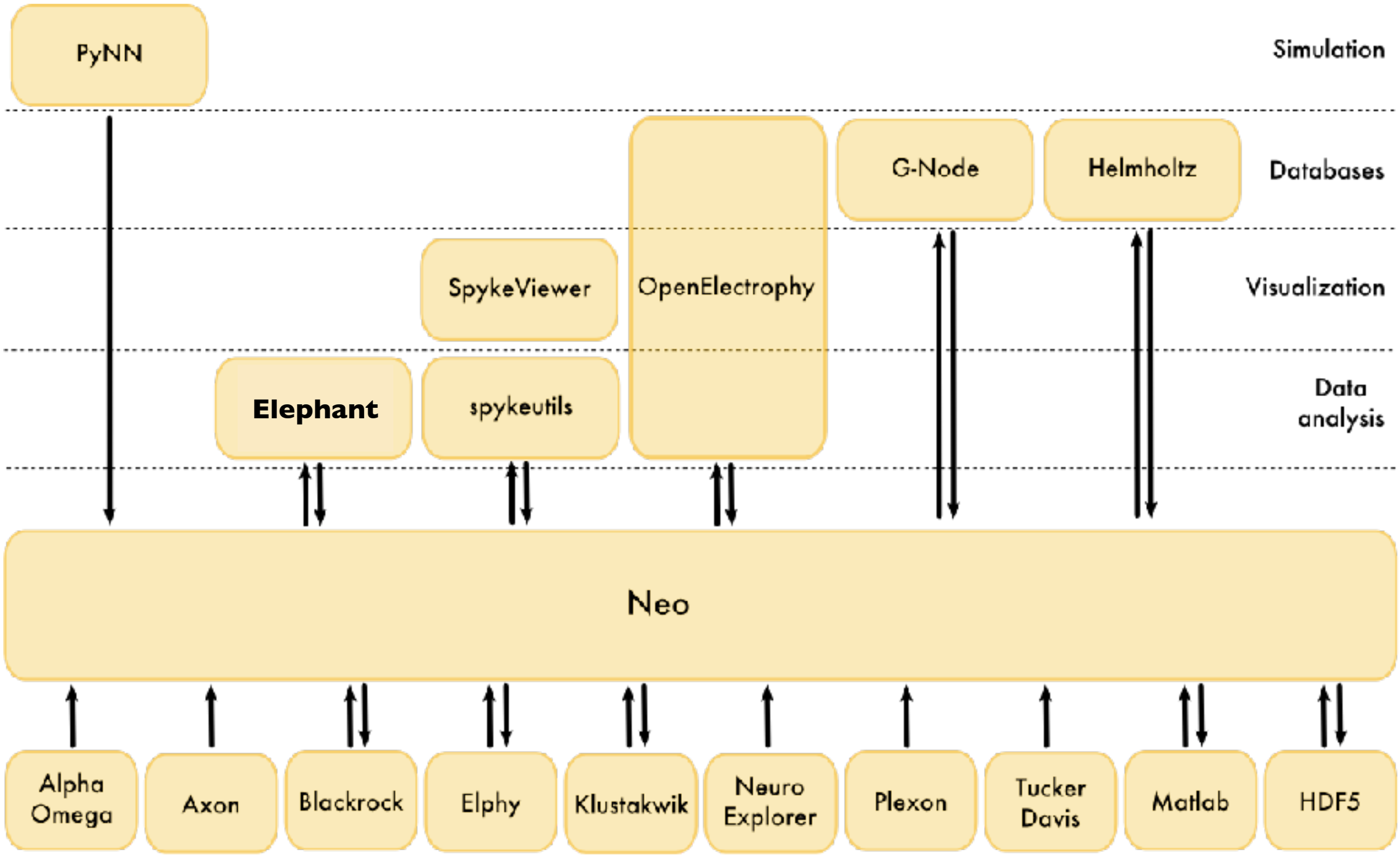
```
cell = sim.Population(1, sim.HH_cond_exp())
step_current = sim.DCSource(start=20.0, stop=80.0)
step_current.inject_into(cell)

cell.record('v')

for amp in (-0.2, -0.1, 0.0, 0.1, 0.2):
    step_current.amplitude = amp
    sim.run(100.0)
    sim.reset(annotations={"amplitude": amp*nA})

data = cell.get_data()

sim.end()

for segment in data.segments:
    vm = segment.analogsignalarrays[0]
    plt.plot(vm.times, vm,
             label=str(segment.annotations["amplitude"]))
plt.legend(loc="upper left")
plt.xlabel("Time (%s)" % vm.times.units._dimensionality)
plt.ylabel("Membrane potential (%s)" % vm.units._dimensionality)
```

http://neuralensemble.org/trac/PyNN/wiki/ChangesToDataHandling

Data-handling

```
cell = sim.Population(1, sim.HH_cond_exp())
step_curre
step_curre

cell.reco

for amp i
    step_
    sim.r
    sim.re

data = ce

sim.end()

for segme
    vm = 
    plt.p

plt.legen
plt.xlabe
plt.ylabe
```




PyNN
Simulation
G-Node
Helmholtz
Databases
SpykeViewer
OpenElectrophy
Visualization
Elephant
spykeutils
Data analysis
Neo
Alpha Omega
Axon
Blackrock
Elphy
Klustakwik
Neuro Explorer
Plexon
Tucker Davis
Matlab
HDF5

Adding a new PyNN backend

- Each backend is a separate Python package

- Implementation choices:
  - entirely independent implementation from scratch (e.g. in C++ with Python wrapper)
  - implement minimal hooks for the “common” implementation
  - anywhere in between

Adding a new PyNN backend

https://arxiv.org/abs/2003.09696

New!

PyCARL: A PyNN Interface for Hardware-Software Co-Simulation of Spiking Neural Network

Adarsha Balaji[1], Prathyusha Adiraju[2], Hirak J. Kashyap[3], Anup Das[1,2], Jeffrey L. Krichmar[3], Nikil D. Dutt[3], and Francky Catthoor[2,4]
[1]Electrical and Computer Engineering, Drexel University, Philadelphia, USA
[2]Neuromorphic Computing, Stichting Imec Nederlands, Eindhoven, Netherlands
[3]Cognitive Science and Computer Science, University of California, Irvine, USA
[4]ESAT Department, KU Leuven and IMEC, Leuven, Belgium

- Each backend is a sep

- Implementation choices:
  - entirely independent implementation from scratch (e.g. in C++ with Python wrapper)
  - implement minimal hooks for the “common” implementation
  - anywhere in between

SONATA

export

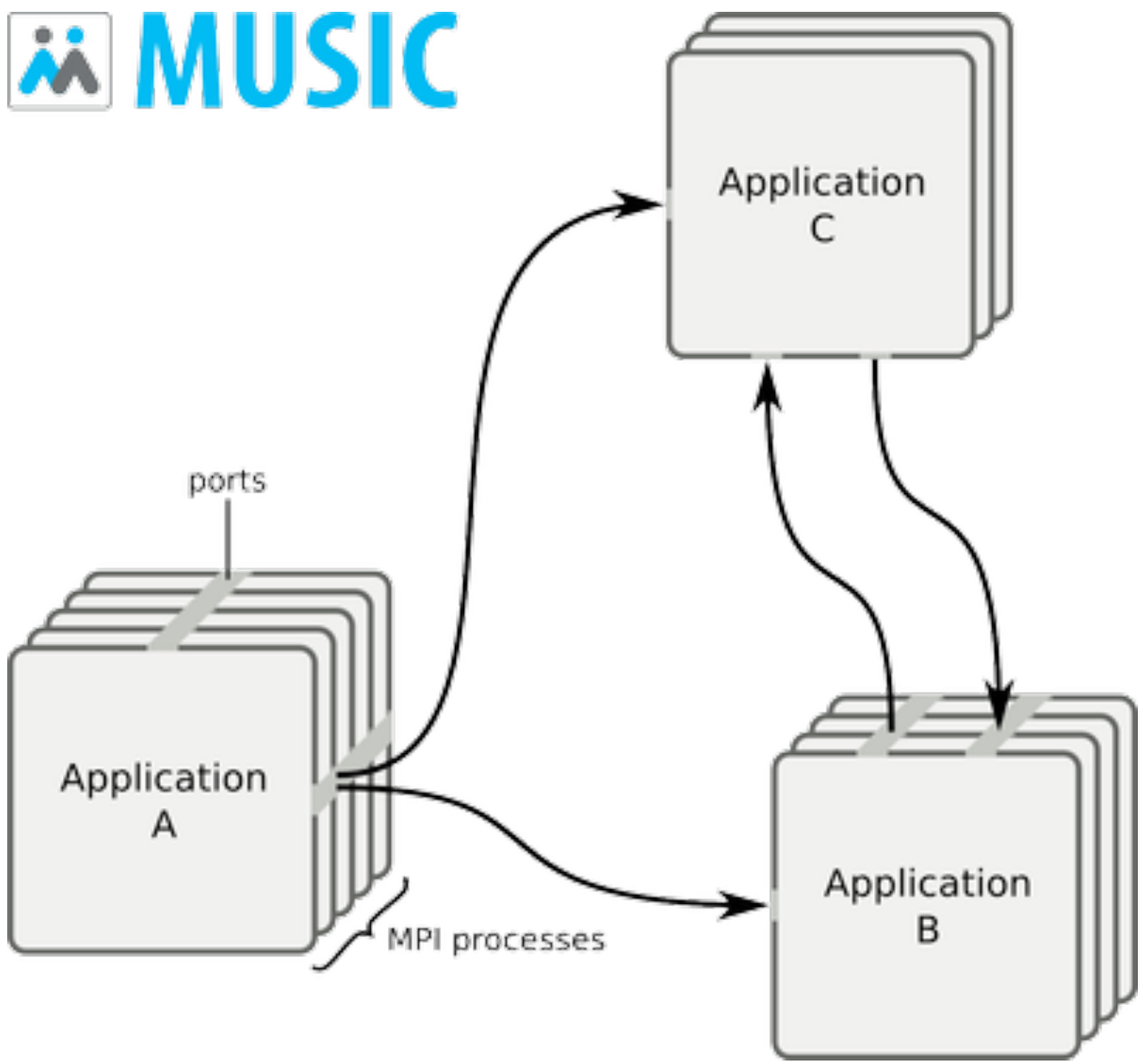
```
from pyNN.network import Network
from pyNN.serialization import export_to_sonata

sim.setup()
...
# create populations, projections, etc.
...

# add populations and projections to a Network
net = Network(pop1, pop2, ...., prj1, prj2, ...)

export_to_sonata(net, "sonata_output_dir")
```

import and run simulation

```
from pyNN.serialization import import_from_sonata, load_sonata_simulation_plan
from pyNN.serialization.sonata import SonataIO
import pyNN.neuron as sim

simulation_plan = load_sonata_simulation_plan("simulation_config.json")
simulation_plan.setup(sim)
net = import_from_sonata("circuit_config.json", sim)
simulation_plan.execute(net)

data = SonataIO("sonata_output_dir").read()
```

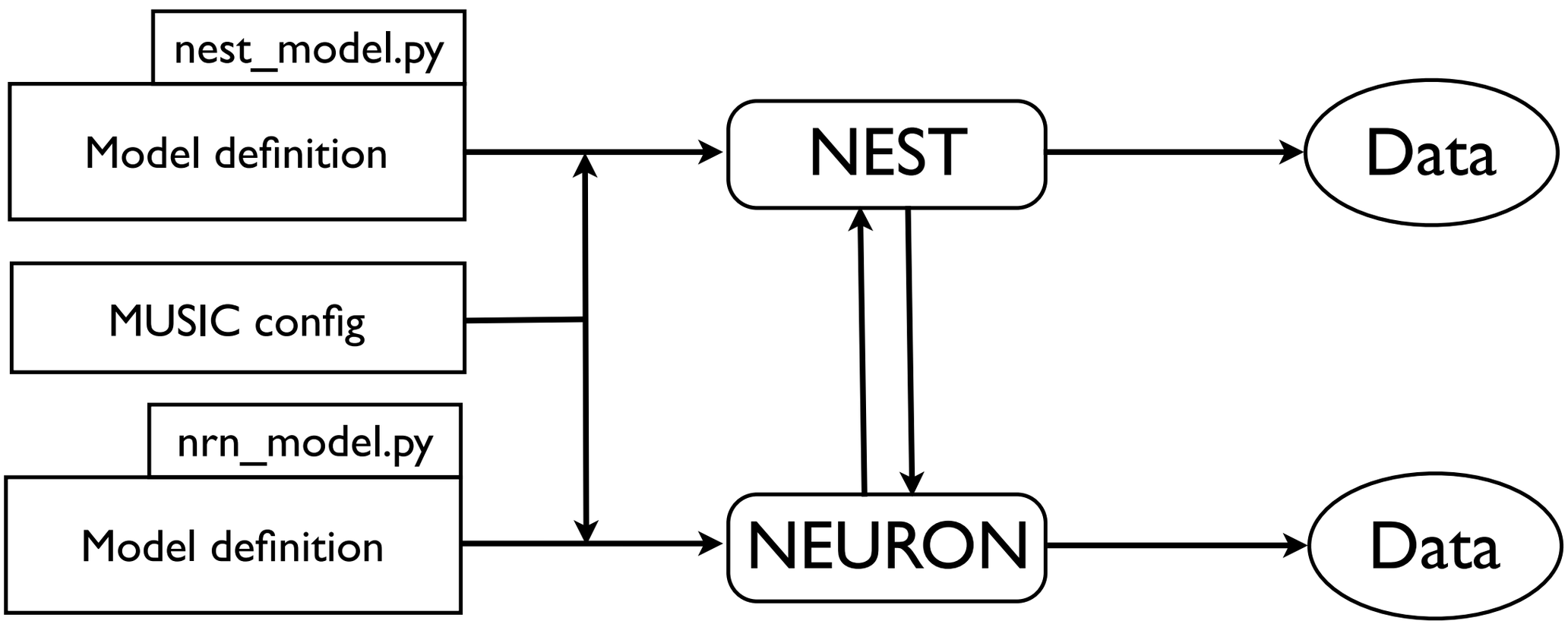
Work in progress…

Multi-simulations

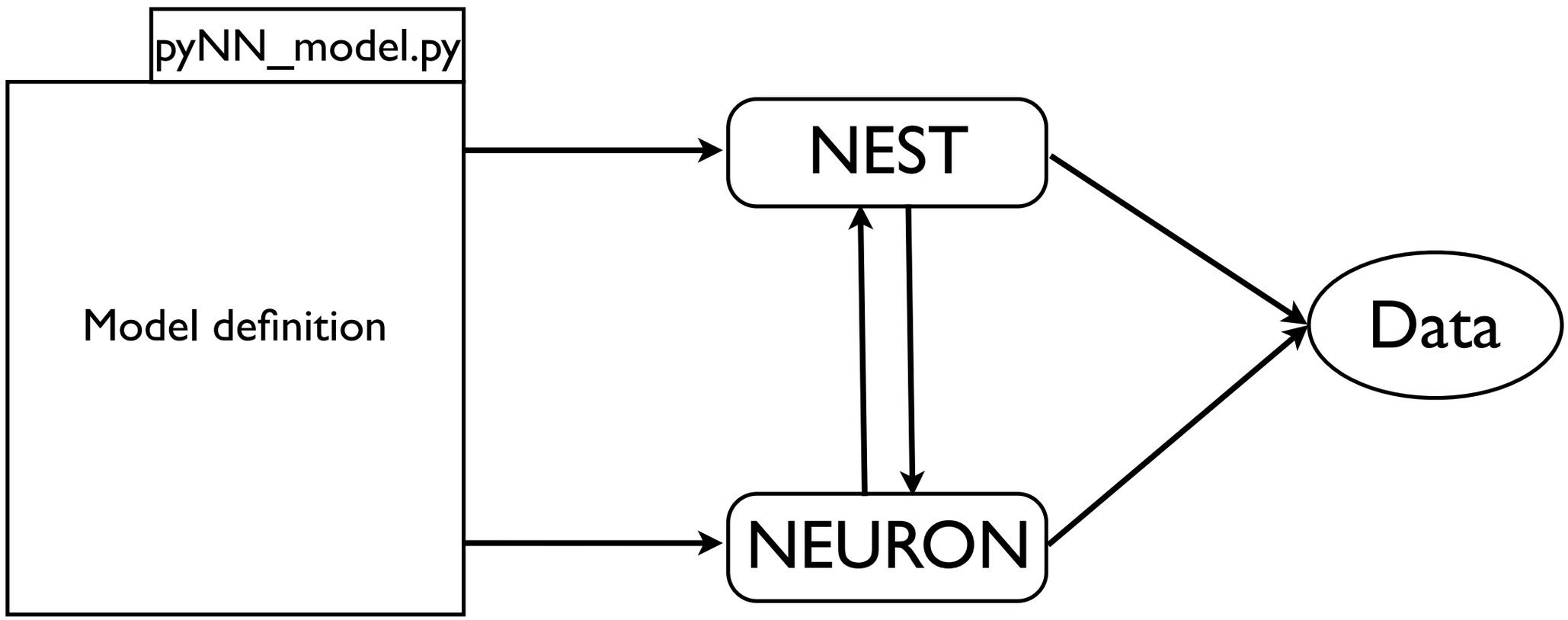


http://software.incf.org/software/music

Multi-simulations



Multi-simulations in PyNN



Multi-simulations in PyNN

```
from pyNN import music

vizapp = music.Config("vizapp", 1, "/path/to/vizapp", "args")
sim1, sim2, viz = music.setup(music.Config("neuron", 10),
                              music.Config("nest", 20),
                              vizapp)

sim1.setup(timestep=0.025)
sim2.setup(timestep=0.1)
pE = sim1.Population((100,100), sim.IF_cond_exp, label="excitatory")
pI = sim2.Population((50,50), sim.IF_cond_exp, label="inhibitory")

def connector(sim):
    DDPC = getattr(sim, "DistanceDependentProbabilityConnector")
    return DDPC("exp(-d**2/400.0)", weights=0.05, delays="0.5+0.01d")
e2e = sim1.Projection(pE, pE, connector(sim1), target="excitatory")
e2i = music.Projection(pE, pI, connector(music), target="excitatory")
i2i = sim2.Projection(pI, pI, connector(sim2), target="inhibitory")

output = music.Port(pE, "spikes", viz, "pE_spikes_viz")

music.run(1000.0)
```

PyNN 2: morphologies and ion channels

http://neuralensemble.org/docs/PyNN/2.0/

Extending the PyNN API

- The scope of PyNN was originally limited to networks of point neurons (integrate-and-fire and related models). The primary reason for this was that at the time only the NEURON simulator had both support for multicompartment models and a Python interface.
- This situation has now changed, with the release of Brian 2, the addition of Python support to MOOSE, development of the Arbor simulation library, and support for multicompartment models in the future versions of the BrainScaleS and SpiNNaker neuromorphic chips.
- We are therefore adapting the PyNN API to support both point neuron models and morphologically-and-biophysically-detailed neuron models (and mixed networks of both model types).

Design goals

The principal design goals are:

1. **maintain the same main level of abstraction**: populations of neurons and the sets of connections between populations (projections);
2. **backwards compatibility** (point neuron models created with PyNN 1.0 (not yet released) or later should work with no, or minimal, changes);
3. **integrate with other open-source simulation tools and standards** (e.g. NeuroML) wherever possible, rather than reinventing the wheel;
4. **support neuromorphic hardware systems.**

Example: ball-and-stick model

```
from neuroml import Segment, Point3DWithDiam as P
from pyNN.morphology import NeuroMLMorphology, uniform
from pyNN.parameters import IonicSpecies
import pyNN.neuron as sim

sim.setup(timestep=0.025)

soma = Segment(proximal=P(x=0, y=0, z=0, diameter=18.8),
               distal=P(x=18.8, y=0, z=0, diameter=18.8),
               name="soma", id=0)
dend = Segment(proximal=P(x=0, y=0, z=0, diameter=2),
               distal=P(x=-500, y=0, z=0, diameter=2),
               name="dendrite",
               parent=soma, id=1)


cell_class = sim.MultiCompartmentNeuron
cell_class.label = "ExampleMultiCompartmentNeuron"
cell_class.ion_channels = {"pas": sim.PassiveLeak, "na": sim.NaChannel,
                           "kdr": sim.KdrChannel}

cell_type = cell_class(morphology=NeuroMLMorphology(segments=(soma, dend)),
                       cm=1.0, Ra=500.0,
                       pas={"conductance_density": uniform("all", 0.0003), "e_rev":-54.3},
                       na={"conductance_density": uniform("soma", 0.120), "e_rev": 50.0},
                       kdr={"conductance_density": uniform("soma", 0.036), "e_rev": -77.0})


cells = sim.Population(2, cell_type, initial_values={'v': [-60.0, -70.0]})
```

morphology defined using libNeuroML

standard library of ion channels

Example: morphology from SWC

```
from pyNN.morphology import load_morphology, uniform, random_section, dendrites,
apical_dendrites, by_distance
from pyNN.parameters import IonicSpecies
import pyNN.neuron as sim

sim.setup(timestep=0.025)

pyr_morph = load_morphology("oi15rpy4-1.CNG.swc")
```

morphology read from SWC

standard library of ion channels and synaptic receptors

```
cell_class = sim.MultiCompartmentNeuron
cell_class.label = "ExampleMultiCompartmentNeuron"
cell_class.ion_channels = {"pas": sim.PassiveLeak, "na": sim.NaChannel,
                           "kdr": sim.KdrChannel}
cell_class.post_synaptic_entities = {"AMPA": sim.CondExpPostSynapticResponse,
                                     "GABA_A": sim.CondExpPostSynapticResponse}

cell_type = cell_class(morphology=pyr_morph),
                       cm=1.0, Ra=500.0,
                       pas={"conductance_density": uniform("all", 0.0003), "e_rev":-54.3},
                       na={"conductance_density": uniform("soma", 0.120), "e_rev": 50.0},
                       kdr={"conductance_density": uniform("soma", 0.036), "e_rev": -77.0}
                       AMPA={"density": uniform('all', 0.05),  # number per µm
                             "e_rev": 0.0, "tau_syn": 2.0},
                       GABA_A={"density": by_distance(dendrites(), lambda d: 0.05 * (d < 50.0)),
                               "e_rev": -70.0, "tau_syn": 5.0})
```

Recording and injecting current

named segments

```
step_current = sim.DCSource(amplitude=0.1, start=50.0, stop=150.0)
step_current.inject_into(cells[0:1], location="soma")

cells.record('spikes')
cells.record(['na.m', 'na.h', 'kdr.n'], locations=['soma'])
cells.record('v', locations=['soma', 'dendrite'])
```

selecting neurite locations

```
step_current = sim.DCSource(amplitude=5.0, start=50.0, stop=150.0)
step_current.inject_into(cells[1:2], location=random_section(apical_dendrites()))

cells.record('spikes')
cells.record(['na.m', 'na.h', 'kdr.n'], locations={'soma': 'soma'})
cells.record('v', locations={'soma': 'soma', 'dendrite': random_section(apical_dendrites())})
```

Networks

```
i2p = sim.Projection(
    inputs,
    pyramidal_cells,
    connector=sim.AllToAllConnector(
        location_selector=random_section(apical_dendrites())),
        synapse_type=sim.StaticSynapse(weight=0.5, delay=0.5),
        receptor_type="AMPA"
    )
)
```

Contributions welcome

- PyNN is a community-developed project, we welcome contributions from anyone who is interested in the project.

http://neuralensemble.org/docs/PyNN/developers_guide.html

http://neuralensemble.org/PyNN

Pierre Yger	Eilif Muller
Daniel Brüderle	Jochen Eppler
Jens Kremkow	Dejan Pecevski
Mike Hull	Michael Schmuker
Mikael Djurfeldt	Bernhard Kaplan
Subhasis Ray	Yury Zaytsev
Jan Antolik	Alexandre Gravier
Thomas Close	Oliver Breitwieser
Jannis Schücker	Maximilian Schmidt
Christian Roessert	Shailesh Appukuttan
Elodie Legouée	Joffrey Gonin
Ankur Sinha	Håkon Mørk

@apdavison

http://andrewdavison.info

PyNN