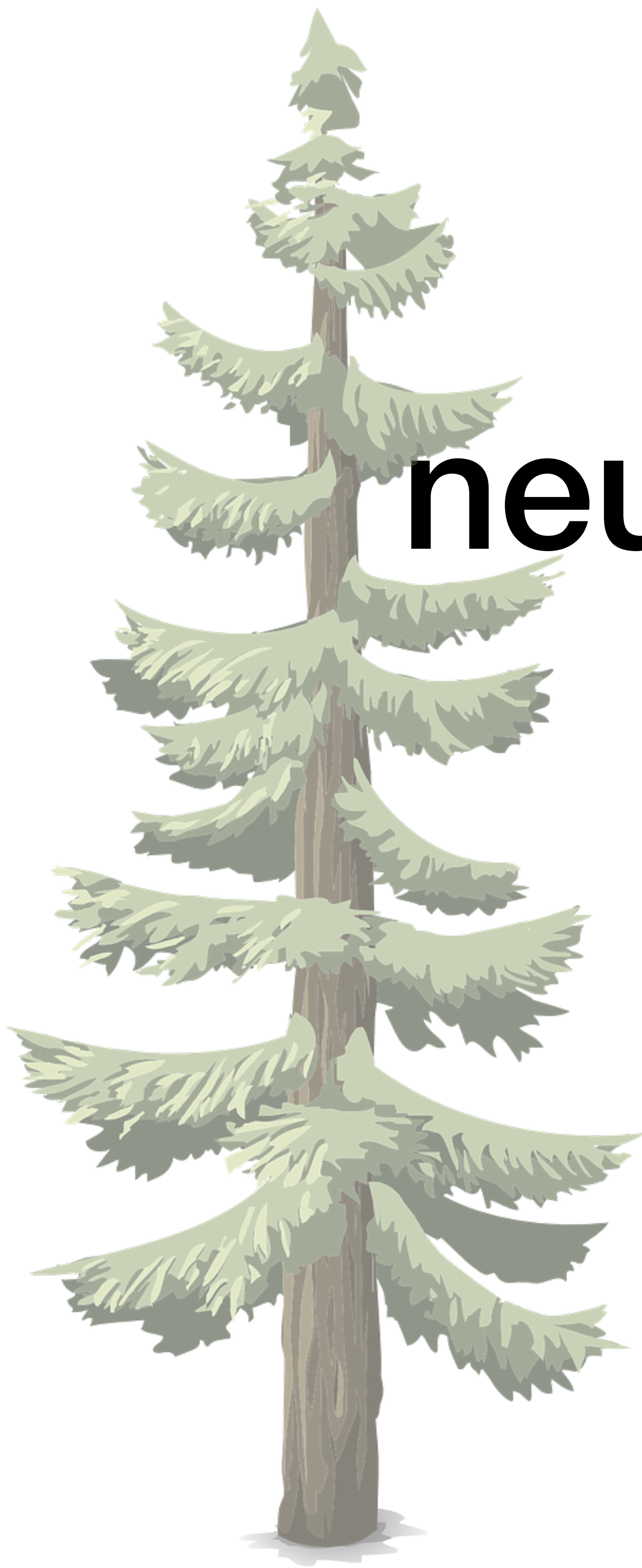


Programming neuromorphic computers: PyNN and beyond

Andrew Davison
Paris-Saclay Institute of Neuroscience
CNRS - Université Paris-Saclay

NICE 2021

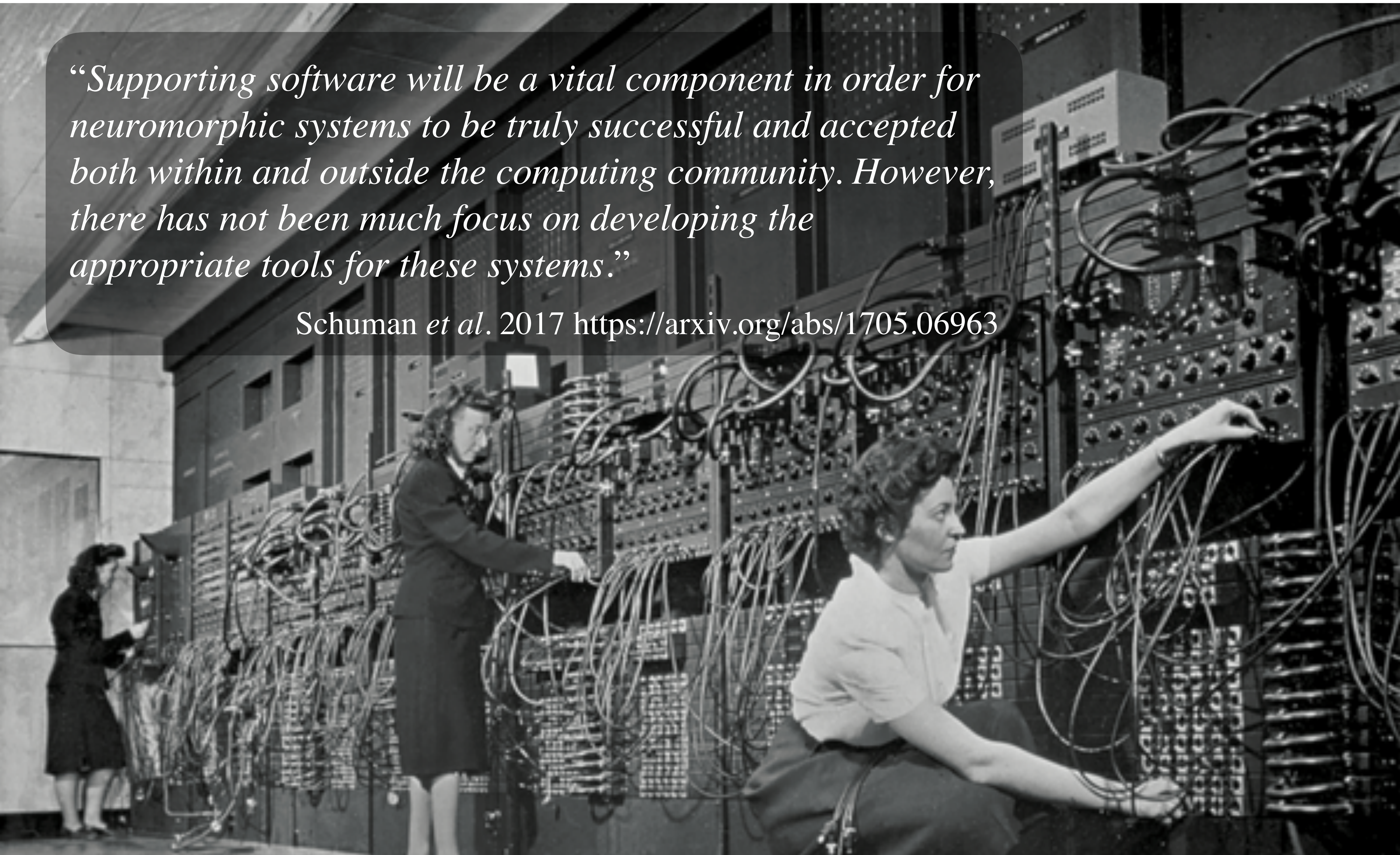
19th March 2021

NeuroPSI
PARIS-SACLAY INSTITUTE OF NEUROSCIENCE

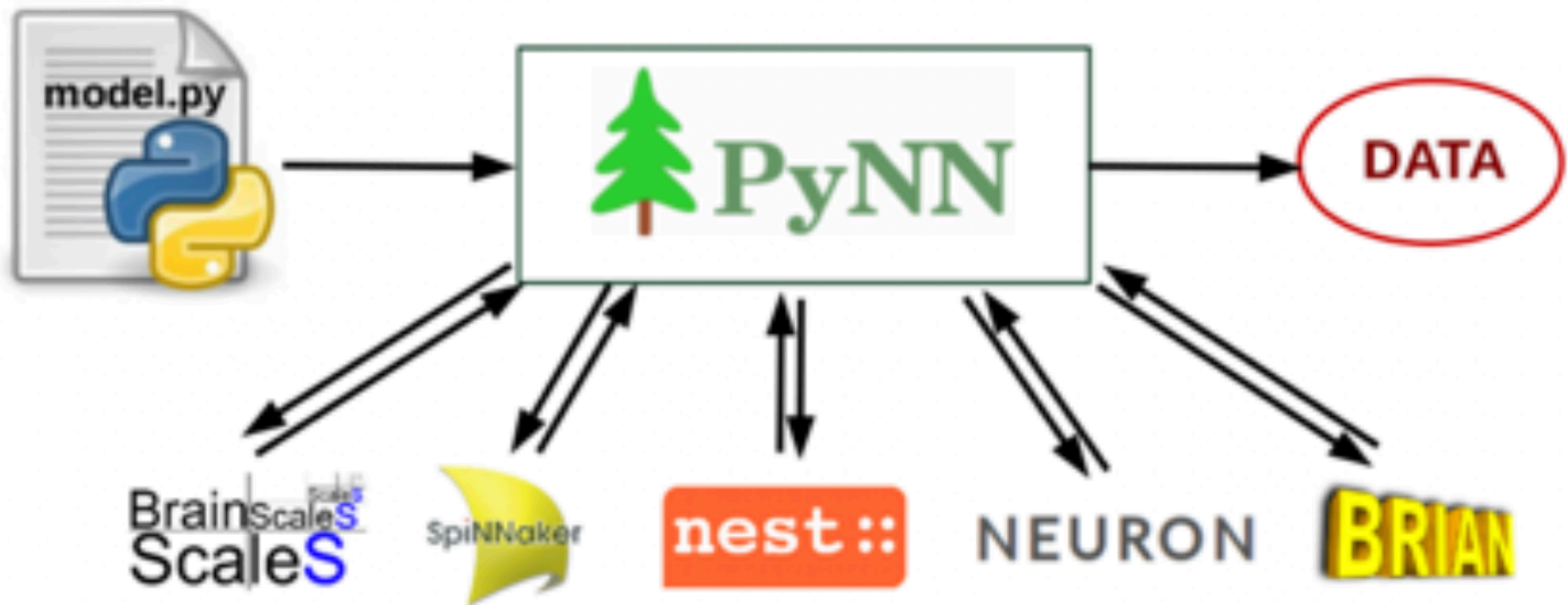
université PARIS-SACLAY

cnrs

Programming neuromorphic computers



Introduction to PyNN



```
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")
```

```python
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")
```

```python
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")
```

```python
import pyNN.brainscales 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")
```

The PyNN API

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

Neuron and synapse models

- “standard” models

available on at least 2 simulators/hardware platforms

- native models

use the PyNN API with simulator-specific models: useful in migrating to “full PyNN”

- NineML, NeuroML, NESTML models

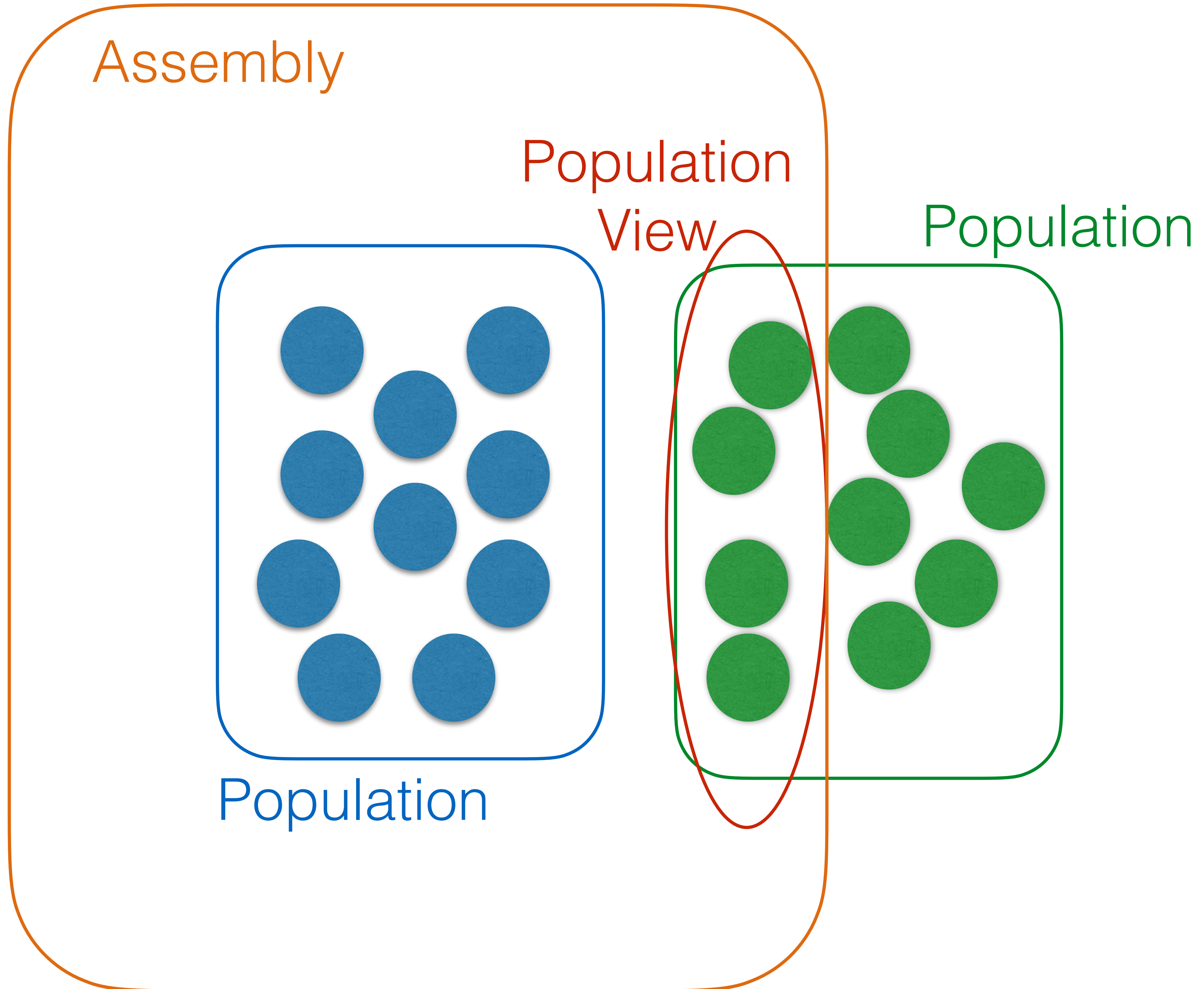
code generation for custom, simulator-independent models

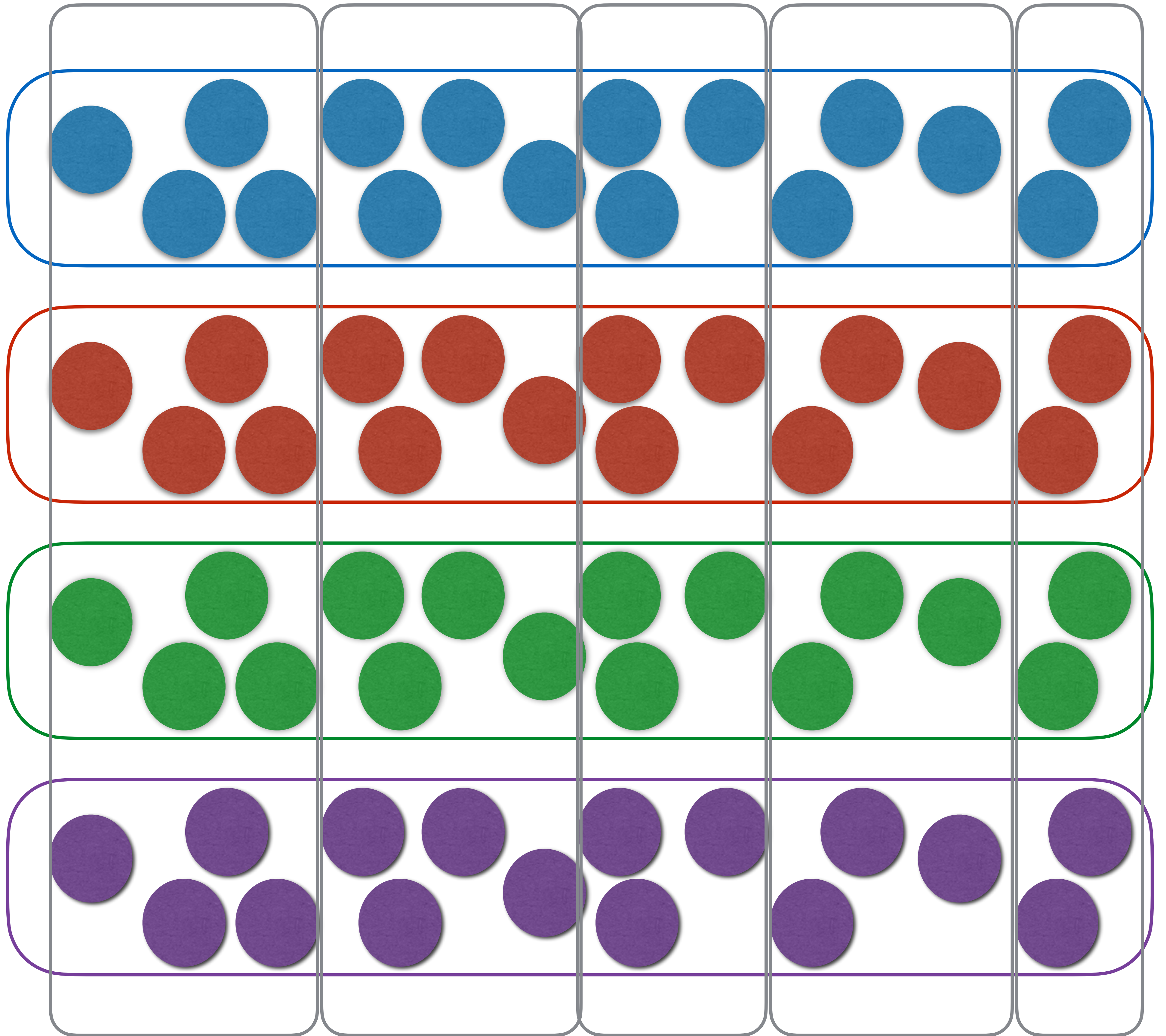
“Native” models

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)
```

Populations




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
```

Parameterization

```
parameter_space = {
    'tau_m': RandomDistribution('uniform', (10.0, 15.0)),
    'cm':    0.85,
    'v_rest': lambda i: np.cos(i*pi*10/n),
    'v_reset': np.linspace(-75.0, -65.0, num=n)}

cell_type = IF_cond_alpha(**parameter_space)
```

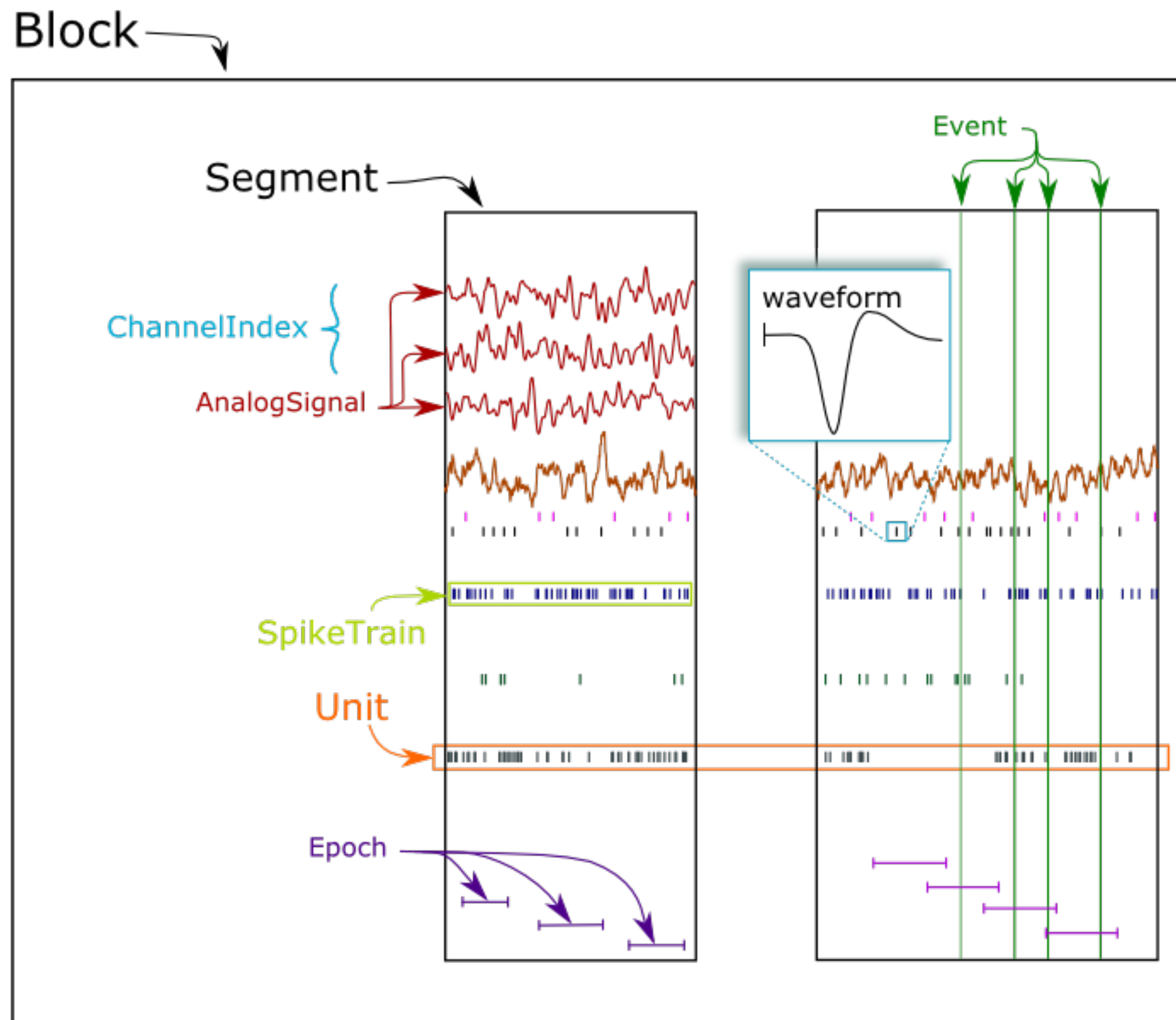
42

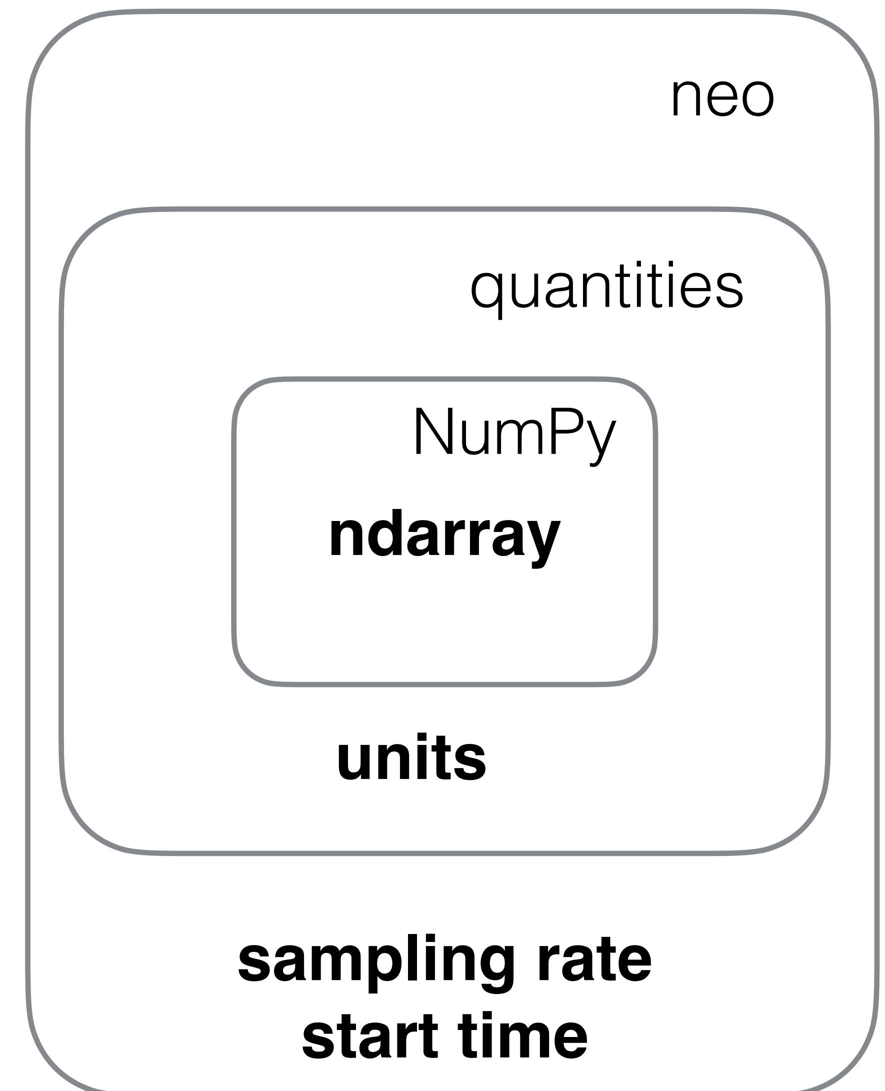
f()

Projections

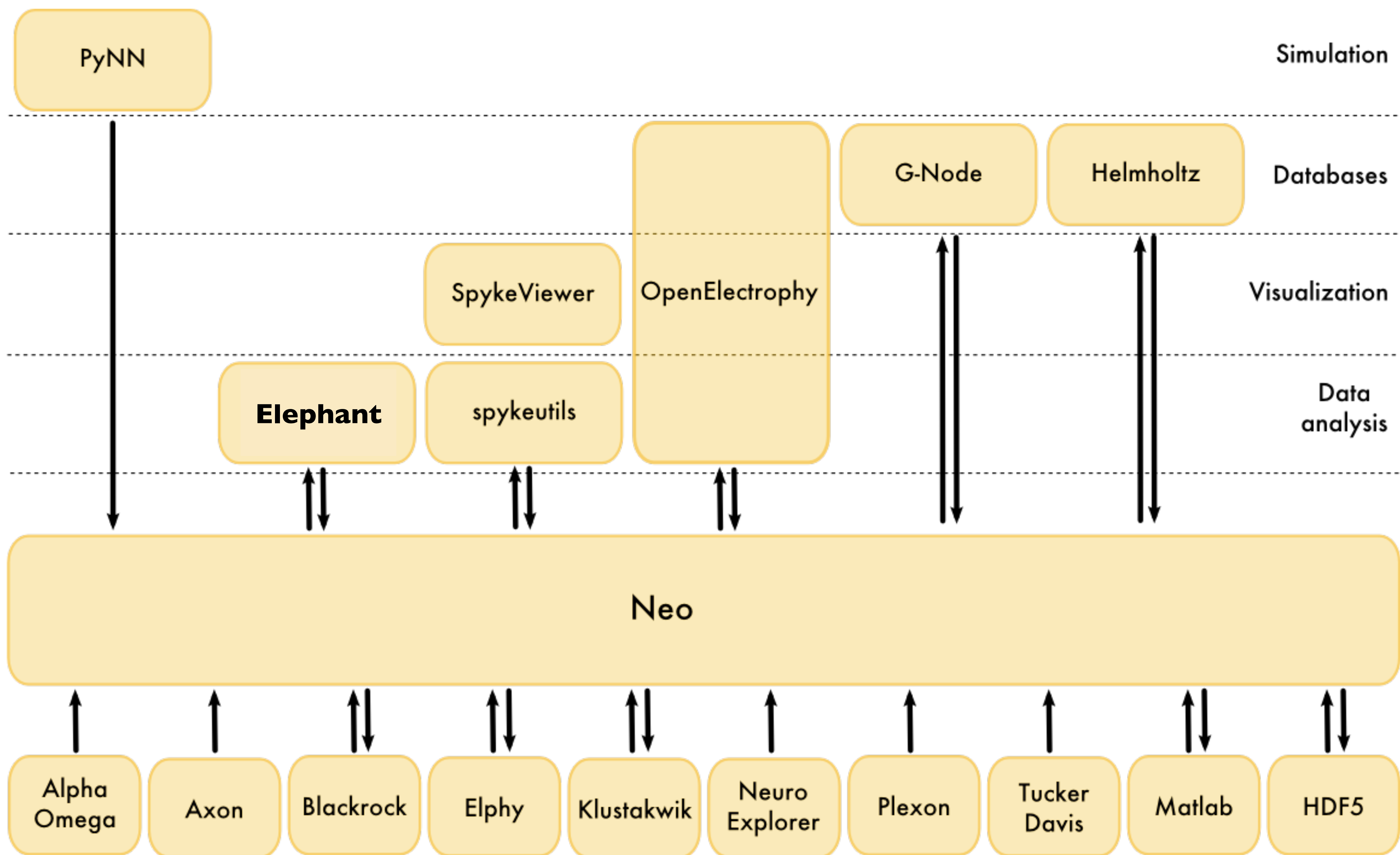
```
rng = NumpyRNG(seed=64754)
sparse_connectivity = FixedProbabilityConnector(0.1, rng=rng)
weight_distr = RandomDistribution('normal', [0.01, 1e-3], rng=rng)
facilitating = TsodyksMarkramSynapse(U=0.04, tau_rec=100.0,
                                     tau_facil=1000.0,
                                     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")
```

Data handling





http://neuralensemble.org/neo


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

PyNN usage and uptake

Star 192 | Fork 104

Dependency graph

Dependencies | Dependents | Dependabot Beta

Repositories that depend on **pynn**

56 Repositories 5 Packages

Repository	Stars	Forks
jakobj / **UP-Tasks**	0	5
ac13711 / **Spike_Sorting_GUI**	0	0
SpiNNakerManchester / **SpiNNCer**	1	0
CSNG-MFF / **mozaik**	12	16
NeuroML / **NeuroMLlite**	7	6
Katterrina / **mff_nprg045**	0	0
NeuromorphicProcessorProject / **snn_toolbox**	187	74
SpiNNakerManchester / **sPyNNaker**	75	32
ruthvik92 / **SpykeFlow**	3	4
MetaCell / **scidash**	1	2

	Cited by	Year
PyNN: a common interface for neuronal network simulators AP Davison, D Brüderle, JM Eppler, J Kremkow, E Muller, D Pecevski, ... Frontiers in neuroinformatics 2, 11	592	2009

Implementing the PyNN API

- 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

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

https://arxiv.org/abs/2003.09696

Hooks for the common implementation

```
class Population(common.Population):

    def _create_cells(self):
        ...

    def _get_parameters(self, *names):
        ...

    def _set_parameters(self, parameter_space):
        ...
```

Hooks for the common implementation

```
class Projection(common.Projection):

    def _convergent_connect(self, presynaptic_indices,
                                  postsynaptic_index,
                                  **connection_parameters):
        ...
```

Hooks for the common implementation

```
class Recorder(recording.Recorder):

    def _record(self, variable, new_ids):
        ...

    def _get_spiketimes(self, id):
        ...

    def _get_all_signals(self, variable, ids, clear=False):
        ...

    def _local_count(self, variable, filter_ids=None):
        ...
```

Hooks for the common implementation

```
class SpikeSourcePoisson(cells.SpikeSourcePoisson):

    translations = build_translations(
        ('start', 'START'),
        ('rate', 'INTERVAL', "1000.0/rate", "1000.0/INTERVAL"),
        ('duration', 'DURATION'),
    )
```

Hooks for the common implementation

```
class State(common.control.BaseState):

    def run_until(self, tstop):
        ...

    def clear(self):
        ...

    def reset(self):
        ...
```

Development model

- open source
- open community
- community governance*

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 Rössert	Shailesh Appukuttan
Elodie Legouée	Joffrey Gonin
Ankur Sinha	Håkon Mørk

CONTRIBUTIONS WELCOME!

https://github.com/NeuralEnsemble/PyNN/

*Any contributor who has had at least three pull requests accepted may be nominated as a maintainer.

Ongoing and future work

- separation of API specification from reference implementation
- revised and extended documentation
- API support for cleaner separation of model and experiment descriptions
- multicompartmental models

Separation of API specification from reference implementation

- "PyNN-like"
- API currently defined by the reference implementation
- Plan: use Python `abc` module, Python 3 type annotations to separate API from implementation
- API simplification? core & extensions

Multicompartmental models

- dendrites offer rich possibilities for computation

Review Article | Published: 11 May 2020

Illuminating dendritic function with computational models

Nature Reviews Neuroscience **21**, 303–321(2020) | Cite this article

- an increasing number of software simulators and neuromorphic systems now support multicompartment models

- time to extend PyNN from point neurons to neurons with detailed morphologies, ion channels…

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

Design goals

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

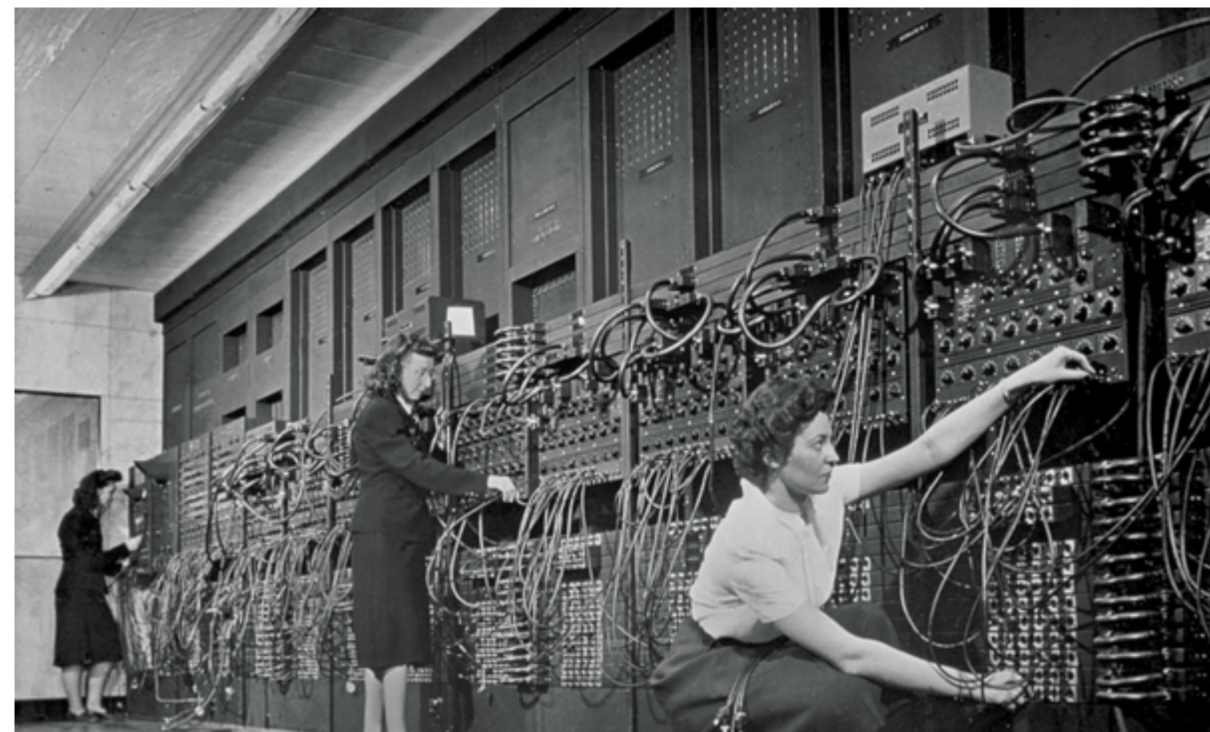
```
step_current = sim.DCSource(amplitude=5.0, start=50.0, stop=150.0)
step_current.inject_into(cells[1:2], location=random_section(apical_dendrites()))

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"
    )
)
```

Beyond PyNN

- if PyNN is assembly / VHDL / C, we need:
  - a standard library
  - higher-level languages
  - exploration of algorithms
  - very preliminary work in my group on this. Open to collaboration



More information

Documentation

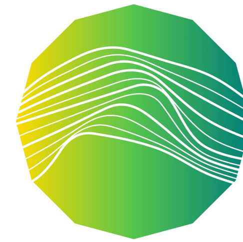
http://neuralensemble.org/PyNN/

Licence

CeCILL (GPL-equivalent)

Mailing list

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



andrew.davison@cnrs.fr

@apdavison

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 Rössert	Shailesh Appukuttan
Elodie Legouée	Joffrey Gonin
Ankur Sinha	Håkon Mørk

Co-funded by
the European Union