

What's new with



PyNN

*Andrew Davison
UNIC, CNRS*

*FACETS CodeJam #2
Gif sur Yvette, 5th-8th May 2008*



- 1 A brief introduction to PyNN
- 2 A tour of the API
- 3 Parallel simulations
- 4 Use cases
- 5 Future directions



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.



Simulator-independent model specification

(“Meta-simulators”)

Simulator-independent environments for developing neuroscience models:

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

Three (*complementary*) approaches:

- GUI (e.g. neuroConstruct)
- XML-based language (e.g. NeuroML)
- interpreted language (e.g. Python)



A common scripting language for neuroscience simulators

Simulator

PCSIM

MOOSE

MVASpike

NEST

NEURON

SPLIT

Brian

FACETS hardware

Language

C++ or Python

SLI or Python

C++ or Python

sli or Python

hoc or Python

C++ (*Python interface planned*)

Python

Python



A common scripting language for neuroscience simulators

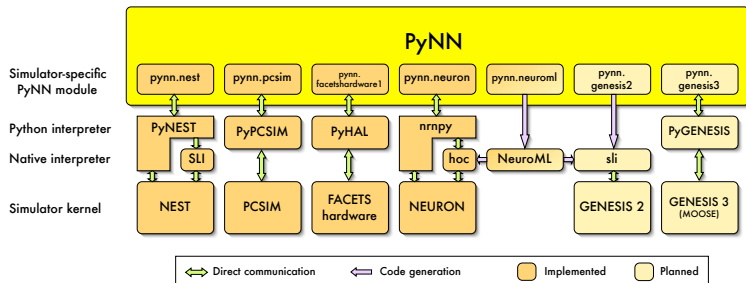
Goal

Write the code for a model simulation *once*, run it on any supported simulator* *without modification*.

* *or hardware device*



Architecture



How to get PyNN

Latest stable version

`http://neuralensemble.org/PyNN/wiki/Download`

Latest development version

```
svn co https://neuralensemble.org/svn/PyNN/trunk pyNN
```

Full documentation

`http://neuralensemble.org/PyNN`

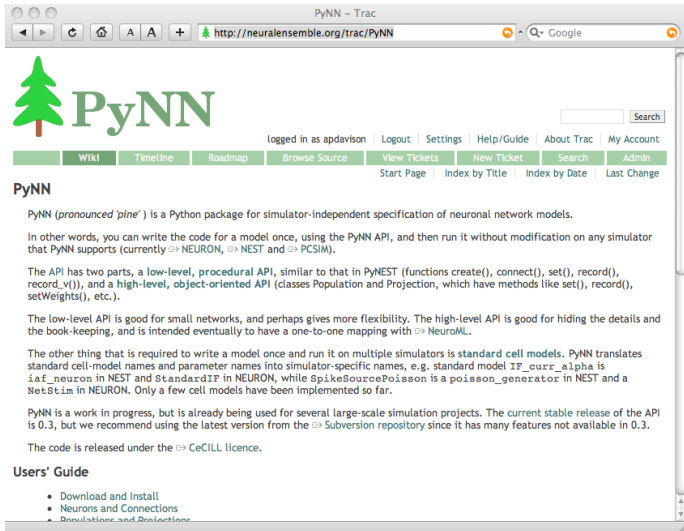


Installing PyNN

via svn or distutils



How to participate in PyNN development



The screenshot shows a web browser window displaying the PyNN Trac page. The browser's address bar shows the URL `http://neuralensemble.org/trac/PyNN`. The page features the PyNN logo (a green pine tree) and the text "PyNN". Below the logo is a navigation menu with links for "Wiki", "Timeline", "Roadmap", "Browse Source", "View Tickets", "New Ticket", "Search", and "Admin". A search box is located to the right of the navigation menu. The main content area is titled "PyNN" and contains several paragraphs of text describing the package. The text includes information about the package's purpose, supported simulators, API types, and current status. A "Users' Guide" section is also visible, with a list of links for "Download and Install", "Neurons and Connections", and "Population and Projection".

PyNN (pronounced 'pine') is a Python package for simulator-independent specification of neuronal network models.

In other words, you can write the code for a model once, using the PyNN API, and then run it without modification on any simulator that PyNN supports (currently [NEURON](#), [NEST](#) and [PCSIM](#)).

The API has two parts, a **low-level, procedural API**, similar to that in PyNEST (functions `create()`, `connect()`, `set()`, `record()`, `record_v()`), and a **high-level, object-oriented API** (classes `Population` and `Projection`, which have methods like `set()`, `record()`, `setWeights()`, etc.).

The low-level API is good for small networks, and perhaps gives more flexibility. The high-level API is good for hiding the details and the book-keeping, and is intended eventually to have a one-to-one mapping with [NeuroML](#).

The other thing that is required to write a model once and run it on multiple simulators is **standard cell models**. PyNN translates standard cell-model names and parameter names into simulator-specific names, e.g. `standard model IF_curr_alpha` is `iaf_neuron` in NEST and `StandardIF` in NEURON, while `SpikeSourcePoisson` is a `poisson_generator` in NEST and a `NetStim` in NEURON. Only a few cell models have been implemented so far.

PyNN is a work in progress, but is already being used for several large-scale simulation projects. The current stable release of the API is 0.3, but we recommend using the latest version from the [Subversion repository](#) since it has many features not available in 0.3.

The code is released under the [CeCILL licence](#).

Users' Guide

- [Download and Install](#)
- [Neurons and Connections](#)
- [Population and Projection](#)

`http://neuralensemble.org/PyNN`



How to participate in PyNN development

Google groups screenshot

Google groups URL



- 1 A brief introduction to PyNN
- 2 A tour of the API**
- 3 Parallel simulations
- 4 Use cases
- 5 Future directions



Selecting the simulator

```
from pyNN.neuron import *  
from pyNN.nest1 import *  
from pyNN.nest2 import *  
from pyNN.pcsim import *  
from pyNN.moose import *  
from pyNN.brian import *  
  
import pyNN.neuron as sim
```



setup() and end()

```
setup(timestep=0.1, min_delay=0.1, debug=False)
```

```
setup(timestep=0.1, min_delay=0.1, debug='pyNN.log',  
      threads=2, shark_teeth=999)
```

```
end()
```



create()

```
create(IF_curr_alpha)
```

```
create(IF_curr_alpha, n=10)
```

```
create(IF_curr_alpha, {'tau_m': 15.0, 'cm': 0.9}, n=10)
```

```
>>> IF_curr_alpha.default_parameters
```

```
{'tau_refrac': 0.0, 'tau_m': 20.0, 'i_offset': 0.0,  
  'cm': 1.0, 'v_init': -65.0, 'v_thresh': -50.0,  
  'tau_syn_E': 0.5, 'v_rest': -65.0, 'tau_syn_I': 0.5,  
  'v_reset': -65.0}
```



create()

```
>>> create(IF_curr_alpha, param_dict='foo': 15.0)
```

```
Traceback (most recent call last):
```

```
.  
. .  
. .
```

```
NonExistentParameterError: foo
```

```
>>> create(IF_curr_alpha, param_dict='tau_m': 'bar')
```

```
Traceback (most recent call last):
```

```
.  
. .  
. .
```

```
InvalidParameterValueError:
```

```
(<type 'str'>, should be <type 'float'>)
```



create()

```
create(IF_curr_alpha, 'v_thresh': -50, 'cm': 0.9)
```

```
create('iaf_neuron', 'V_th': -50, 'C_m': 900.0)
```



Standard cell models

IF_curr_alpha
IF_curr_exp
IF_cond_alpha
IF_cond_exp,
IF_cond_exp_gsfa_grr
IF_facets_hardware1
HH_cond_exp,
EIF_cond_alpha_isfa_ista
SpikeSourcePoisson
SpikeSourceInhGamma
SpikeSourceArray



Standard cell models

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

Name	Units	NEST	NEURON
v_rest	mV	U0	v_rest
v_reset	mV	Vreset	v_reset
cm	nF	C†	CM
tau_m	ms	Tau	tau_m
tau_refrac	ms	TauR	t_refrac
tau_syn	ms	TauSyn	tau_syn
v_thresh	mV	Theta	v_thresh
i_offset	nA	I0†	i_offset

†Unit differences: C is in pF, I0 in pA.



ID objects

```
>>> my_cell = create(IF_cond_exp)
>>> print my_cell
1
>>> type(my_cell)
<class 'pyNN.nest2.ID'>
>>> my_cell.tau_m
20.0
>>> my_cell.position
(1.0, 0.0, 0.0)
>>> my_cell.position = (0.76, 0.54, 0.32)
```



connect()

```
spike_source = create(SpikeSourceArray,  
                      {'spike_times': [10.0, 20.0, 30.0]})  
cell_list = create(IF_curr_exp, n=10)  
  
connect(spike_source, cell_list)  
  
connect(sources, targets, weight=1.5, delay=0.5,  
       p=0.2, synapse_type='inhibitory')
```



record()

```
record(cell, "spikes.dat")
```

```
record_v(cell_list, "Vm.dat")
```

Writing occurs on end()



```
run()
```

```
run(100.0)
```



`get_current_time()`

`get_time_step()`

`get_min_delay()`

`num_processes()`

`rank()`



Random numbers

```
>>> from pyNN.random import NumpyRNG, GSLRNG, NativeRNG
```

```
>>> rng = NumpyRNG(seed=12345)
```

```
>>> rng.next()
```

```
0.6754034
```

```
>>> rng.next(3, 'uniform', (-70,-65))
```

```
[-67.4326, -69.9223, -65.4566]
```

- Use `NativeRNG` or `GSLRNG` to ensure different simulators get the same random numbers
- Use `NativeRNG` to use a simulator's built-in RNG



Random numbers

```
>>> from pyNN.random import RandomDistribution

>>> distr = RandomDistribution('uniform', (-70, -65),
...                             rng=rng)
>>> distr.next(3)
[-67.4326, -69.9223, -65.4566]
```



Populations

```
p1 = Population((10,10), IF_curr_exp)
```

```
p2 = Population(100, SpikeSourceArray,  
               label="Input Population")
```

```
p3 = Population(dims=(3,4,5), cellclass=IF_cond_alpha,  
               cellparams={'v_thresh': -55.0},  
               label="Column 1")
```

```
p4 = Population(20, 'iaf_neuron', {'Tau': 15.0,  
                                   'C': 100.0})
```



Populations

Accessing individual members

```
>>> p1[0,0]
1
>>> p1[9,9]
100
>>> p3[2,1,0]
246

>>> p3.locate(246)
(2, 1, 0)

>>> p1.index(99)
100

>>> p1[0,0].tau_m = 12.3
```



Populations

Iterators

```
>>> for id in p1:
...     print id, id.tau_m
...
0 12.3
1 20.0
2 20.0
...

>>> for addr in p1.addresses():
...     print addr
...
(0, 0)
(0, 1)
(0, 2)
...
(0, 9)
(1, 0)
```



set(), tset(), rset()

```
>>> p1.set("tau_m", 20.0)

>>> p1.set('tau_m':20, 'v_rest':-65)

>>> distr = RandomDistribution('uniform', [-70,-55])
>>> p1.rset('v_init', distr)

>>> import numpy
>>> current_input = numpy.zeros(p1.dim)
>>> current_input[:,0] = 0.1
>>> p1.tset('i_offset', current_input)
```



Recording

```
# record from all neurons in the population
```

```
>>> p1.record()
```

```
# record from 10 neurons chosen at random
```

```
>>> p1.record(10)
```

```
# record from specific neurons
```

```
>>> p1.record([p1[0,0], p1[0,1], p1[0,2]])
```

```
>>> p1.printSpikes("spikefile.dat")
```

```
>>> p1.getSpikes()
```

```
array([])
```



Position in space

```
>>> p1[1,0].position = (0.0, 0.1, 0.2)
```

```
>>> p1[1,0].position
```

```
array([ 0. ,  0.1,  0.2])
```

```
>>> p1.positions
```

```
array([[...]])
```

```
>>> p1.nearest((4.5, 7.8, 3.3))
```

```
48
```

```
>>> p1[p1.locate(48)].position
```

```
array([ 4.,  8.,  0.])
```



Projections

```
prj2_1 = Projection(p2, p1, AllToAllConnector())
```

```
prj1_2 = Projection(p1, p2, FixedProbabilityConnector(0.02),  
                    target='inhibitory', label='foo',  
                    rng=NumpyRNG())
```



Connectors

AllToAllConnector

OneToOneConnector

FixedProbabilityConnector

DistanceDependentProbabilityConnector

FixedNumberPostConnector

FixedNumberPostConnector

FromFileConnector*

FromListConnector

(* cf Projection.saveConnections(filename))



```
c = DistanceDependentProbabilityConnector(  
    "exp(-abs(d))",  
    axes='xy',  
    periodic_boundaries=(500, 500, 0),  
    weights=0.7,  
    delays=RandomDistribution('gamma', [1,0.1])  
)
```



Weights and delays

```
>>> prj1_1.setWeights(0.2)

>>> weight_list = 0.1*numpy.ones(len(prj2_1))
>>> weight_list[0:5] = 0.2
>>> prj2_1.setWeights(weight_list)

>>> prj1_1.randomizeWeights(weight_distr)

>>> prj1_2.setDelays('exp(-d/50.0)+0.1')
```

[Note: synaptic weights are in nA for current-based synapses and μ S for conductance-based synapses]



Weights and delays

```
w_array = prj.getWeights()
```

```
prj.printWeights(filename)
```



Synaptic plasticity

```
# Facilitating/depressing synapses
depressing_syn = SynapseDynamics(
    fast=TsodyksMarkramMechanism(**params))
prj = Projection(pre, post, AllToAllConnector(),
    synapse_dynamics=depressing_syn)

# STDP
stdp_model = STDPMechanism(
    timing_dependence=SpikePairRule(
        tau_plus=20.0,
        tau_minus=20.0),
    weight_dependence=AdditiveWeightDependence(
        w_min=0, w_max=0.02,
        A_plus=0.01, A_minus=0.012)
)
prj2 = Projection(pre, post, FixedProbabilityConnector(p=0.1),
    synapse_dynamics=SynapseDynamics(slow=stdp_model))
```



Outline

- 1 A brief introduction to PyNN
- 2 A tour of the API
- 3 Parallel simulations**
- 4 Use cases
- 5 Future directions



png from NEURON on different numbers of processors?



Outline

- 1 A brief introduction to PyNN
- 2 A tour of the API
- 3 Parallel simulations
- 4 Use cases**
- 5 Future directions



Testing a model on multiple simulators

Cross-checking gives greater confidence in correctness of results.

Porting a model between simulators

Gradually replace simulator-specific code with Python code, checking results are unchanged at each step.

Hardware interface

Neuromorphic VLSI hardware can also use PyNN, allowing direct comparison of numerical simulations and emulations in silicon.

Collaborating between different groups

Each group can use their preferred simulator, while working on a common code base.



Outline

- 1 A brief introduction to PyNN
- 2 A tour of the API
- 3 Parallel simulations
- 4 Use cases
- 5 Future directions**



Future directions

- Extend range of simulators supported (full NeuroML support, MOOSE, Brian,...)
- Support non-spiking (firing-rate based) neuron models?
- Support explicit units (cf Brian)
- Optimisation, so PyNN is only a little slower than native code
- Improved parallelisation
- Extensions of the API:
 - current highest level of organisation is Population, Projection.
 - extend to Column, Layer, Meta-column, Map, ...
 - extend stimuli, e.g., DriftingGrating, DenseNoise,...
 - extend recording, e.g. `recordActivityMap()`, ...

