```
1  import numpy
```

import a simulator

```
5  import pyNN.nest as sim
6  # import pyNN.neuron as sim
7  # import pyNN.pcsim as sim
8  # import pyNN.brian as sim
9  # import pyNN.facetshardware1 as sim
```

Small networks

Before using any other functions or classes from PyNN, the user must call the setup() function

```
15 sim.setup(timestep=0.1, min_delay=0.1,max_delay=0.5, debug=False)
```

Some simulators accept additional arguments, e.g.

```
18 sim.setup(rngseeds=[12345,67890],threads=2) # nest
```

Creating neurons

Neurons are created with the create() function:

```
24 sim.create(sim.IF_curr_alpha)
```

Here, IF_curr_alpha is a so-called “standard cell”, which will work with any PyNN backend, whether NEURON, NEST or .. You don’t have ro use the standard cells. You can also use neuron models which are only available to a specific simulator:

```
28 sim.create("iaf_neuron")
```

To create many neurons at once, add the n argument:

```
32 sim.create(sim.IF_curr_alpha ,n=10)
```

The neurons we have created so far have all had default paramter values, stored in the default_values of the standard cell class:

```
35 print sim.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}
```

To use different paramter values, use the paramter_dict argument:

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

If you wish to do something with the cell after creating it: record from it, change a parameter, connect it to another cell, you should assign the return value of the function to a variable, e.g.:

```
40 cell = sim.create(sim.IF_curr_alpha)
41 cell_list = sim.create(sim.IF_curr_alpha,n=10)
```

Connecting neurons

Any neuron that emits spikes can be connected to any neuron with at least one synapse using the connect() function:

```
47 times = [10.,20.,30.]
48 spike_source = sim.create(sim.SpikeSourceArray,{'spike_times':times})
49
50 cell_list2 = sim.create(sim.IF_cond_exp,n=10)
51 sim.connect(spike_source,cell_list2)
```

In case we connect a spike-generating mechanism to each cell in the list, we create 10 connection at once. For clarity, we could also have specified the arguments names:

```
56 sim.connect(source=spike_source, target=cell_list2)
```

Either source or target or both may be individual cell ids or lists of ids. In the latter case, each source (presynaptic) cell is connected to every target (postsynaptic) cell with probability given by the optional argument p, which defaults to 1, e.g.:

```
58 source_list = cell_list
59 target_list = cell_list2
60 sim.connect(source_list, target_list, p=0.5)
```

Weights and delays can be specified:

```
63 sim.connect(source_list, target_list, weight=1.5, delay=1.0)
```

(Weights are in nA for 'current-based' synapses or muS for 'conductance-based' synapses. Delays are in ms. For current-based synapses, weights should be negative for inhibitory synapses. For conductance-based synapses, weights should always be positive, since the effect of a synapse is determined by its reversal potential.)

If the neuron model has more than one synapse mechanism, or more than one synaptic location, the particular synapse to which the connection should be made is specified with the synapse_type argument, e.g.:

```
68 sim.connect(source_list, target_list, weight=1.5, delay=1.0,synapse_type='inhibitory')
```

Setting neuron parameters

To change a single parameter of a single neuron, set the relevant attribute of the neuron ID object, e.g.:

```
73 cells = sim.create(sim.IF_curr_exp, n=10)
74 print cells[0].tau_m
```

```
20.0
```

```
75 cells[0].tau_m = 15
76 print cells[0].tau_m
```

```
15.0
```

To change several parameters at once for a single neuron, use the verb|set_parameters()| method of the neuron ID, e.g.:

```
79 cells[1].set_parameters(tau_m=10.0, cm=0.5)
80 print cells[1].tau_m
```

```
10.0
```

```
82 print cells[1].cm
```

```
0.5
```

To change parameters for several cells at once, use the set() function, e.g.:

```
84 sim.set(cells[0:5], param='v_init', val=-65.0)
85 print cells[0].v_init
```

```
-65.0
```

```
print cells[5].v_init
```

```
-65.0
```

Individual parameters can be set using the param and val arguments, as above, or multiple parameters can be set at once by passing a dictionary of name:value pairs as the param argument, with val empty, e.g.:

```
sim.set(cells, param={'tau_refrac': 2.0,'tau_syn_E': 5.0})
```

Setting position in space

In some cases it is important to know the position of a neuron in space. This information can be set and retrieved using the *position* attribute of the neuron ID:

```
cells[0].position = (75, 456, 56)
```

print cells[0].position

Positions must always be in 3D, and may be given as integers or floating-point values, and as tuples or as numpy arrays. No specific scale of units is assumed, although many parts of PyNN do assume a Euclidean coordinate system.

Recording spikes and membrane potential

To record action potentials use the record() function and to record membrane potential use the record_v() function. The arguments for both functions are a cell id or list of ids, and a filename, e.g.: sim.record(cell, “spikes.dat”) sim.record_v(cell_list, “v.dat”)

By default, all simulators write data files in the same format. In some cases it is more efficient to write files in the simulator’s native format, rather than the standard PyNN format.

Running a simulation and Finishing up

The run() function runs the simulation for a given number of milliseconds, e.g.:

```
113 sim.run(1000.0)
```

The end() function is called at the end of a simulation to remove temporarly folders etc.

```
117 sim.end()
```

Standard cell types

Standard models are neuron models that are available in at least two of the simulation engines supported by PyNN. PyNN performs automatic translation of parameter names, types and units.

```
123 from pyNN import nest
```

print nest.list_standard_models()

Larger networks

Problems with creating very large networks using create() and connect() involves writing a lot of repetitive code, which is the same or similar for every model: iterating over lists of cells and connections, creating common projection patterns, recording from all or a subset of neurons...

For these reasons, PyNN provides: Populations and Projections

```
137 p1 = sim.Population((10,10), sim.IF_curr_exp)
138 p2 = sim.Population(100, sim.SpikeSourceArray,label="Input Population")
139 p3 = sim.Population(dims=(3,4,5), cellclass=sim.IF_cond_alpha,cellparams={'v_thresh': -55, 'tau_m'
    : 10},label="Column 1")
```

The population dimensions can be retrieved using the dim attribute, e.g.:

```
143 print p1.dim, p2.dim, p3.dim
```

```
(10, 10) (100,) (3, 4, 5)
```

while the total number of neurons in a population can be obtained with the Python len() function:

```
146 print len(p1), len(p2), len(p3)
```

```
100 100 60
```

The previous examples all use PyNN standard cell models. It is also possible to use simulator-specific models, but in this case the cellclass should be given as a string, e.g.:

```
149 p4 = sim.Population(20, "iaf_neuron")
```

This example will work with NEST but not with NEURON or PCSIM.
Addressing individual neurons

```
153 print p1[0,0]
```

```
55
```

```
155 print p1[9,9]
```

```
154
```

To obtain an address given the id, use locate(), e.g.:

```
157 print p3[2,2,0]
```

```
305
```

```
159 print p3.locate(305)
```

```
(2, 2, 0)
```

The positions of individual neurons in a population can be accessed using their position attribute, e.g.:

```
161 p1[1,0].position = (0.0, 0.1, 0.2)
162 print p1[1,0].position
```

To obtain the positions of all neurons at once (as a numpy array), use the positions attribute of the Population object, e.g.: p1.positions

Recording

Recording spike times: record() Recording membrane potential: record_v() Record from all neurons in the population

```
175 p1.record()
```

```
[ 0.   0.1  0.2]
```

Record from 10 neurons chosen at random

```
178 p1.record(10)
```

Record from specific neurons

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

Writing the recorded values to file is done with a second pair of methods, printSpikes() and print_v(), e.g.: p1.printSpikes("spikefile.dat")

Connecting two Populations with a Projection

A Projection object is a container for all the synaptic connections of a given type between neurons in two Populations, together with methods for setting synaptic weights and delays, e.g.:

```
190 prj = sim.Projection(p2, p1, sim.AllToAllConnector())
```

Use of the OneToOneConnector requires that the pre- and post-synaptic populations have the same dimensions, e.g.:

```
193 prj = sim.Projection(p1, p1, sim.OneToOneConnector())
```

With the FixedProbabilityConnector method, each possible connection between all pre-synaptic neurons and all post-synaptic neurons is created with probability p_connect:

```
196 prj = sim.Projection(p2, p3, sim.FixedProbabilityConnector(0.2))
```

Setting weights and delays

To set all weights to the same value:

```
201 connector = sim.AllToAllConnector(weights=0.7)
202 prj = sim.Projection(p1, p3, connector)
```

To set delays to random values taken from a specific distribution:

```
205 distr = sim.RandomDistribution('gamma', [2.,0.1])
206 conn = sim.FixedNumberPostConnector(n=20, delays=distr)
```

```
207 prj = sim.Projection(p2, p1, conn)
```

To set individual weights and delays to specific values:

```
210 weights = numpy.arange(1.1, 2.0, 0.9/len(p1))
211 delays = 2*weights
212 connector = sim.OneToOneConnector(weights=weights,delays=delays)
213 prj = sim.Projection(p1, p1, connector)
```

Synaptic plasticity

A Projection with facilitating/depressing synapses, but no long-term plasticity:

```
219 depressing_syn = sim.TsodyksMarkramMechanism()
220 syn_dyn = sim.SynapseDynamics(fast=depressing_syn)
221 prj = sim.Projection(p4, p4, sim.AllToAllConnector(),synapse_dynamics=syn_dyn)
```