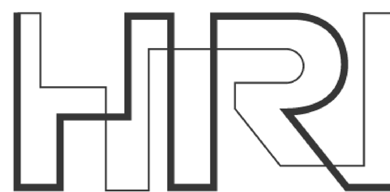
NEST 2: A Parallel Simulator for Large Neuronal Networks

Markus Diesmann[1,2], Jochen Martin Eppler[2,3], Marc-Oliver Gewaltig[2,3], Moritz Helias[2]
Abigail Morrison[1], Hans Ekkehard Plesser[4],

1
RIKEN Brain Science Institute
Wako City
351-0198 Saitama, Japan
http://www.brain.riken.jp

2
Bernstein Center for Computational Neuroscience
Hansastraße 9a
79104 Freiburg, Germany
http://www.bccn.uni-freiburg.de



3
Honda Research Institute Europe
Carl-Legien-Str. 30
63073 Offenbach/Main, Germany
http://www.honda-ri.de

4
Norwegian University of Life Sciences (UMB)
Drøbakveien 31
1432 Ås, Norway
http://www.umb.no

- NEST is a simulator for large networks of spiking neurons that runs on workstation computers and clusters.
- We present PyNEST, the Python interface to NEST. It allows
  - convenient simulation setup and control from Python
  - stimulus construction using other Python modules
  - simulation, analysis and display of simulation results in one script

The PyNEST API

NEST [1,2] sees the neural world as a directed, weighted graph of nodes and connections. Nodes are either neurons, or devices to stimulate or observe the network. Connections are characterized by a configurable, but fixed delay, and a weight that can be static or change according to a plasticity rule (e.g. STDP). Several models for nodes and connections are built into NEST. Their parameters are accessed via dictionaries.

Models

`Models(mtype="all", sel=None)`: Return a list of all available models (nodes and synapses). Use *mtype="nodes"* to only see node models, *mtype="synapses"* to only see synapse models. *sel* can be a string, used to filter the result list and only return model names containing it.

`GetDefaults(model)`: Return a dictionary with the default parameters of *model*.

`SetDefaults(model, params)`: Set the default parameters of *model* to *params*.

`CopyModel(existing, new, params=None)`: Create a *new* model by copying an *existing* one. Default parameters can be given as *params*, or else are taken from *existing*.

Nodes

`Create(model, n=1, params=None)`: Create *n* instances of type *model*. Parameters for the new nodes can are given as *params* (a single dictionary or a list of dictionaries with size *n*). If omitted, the *model*'s defaults are used.

`GetStatus(nodes, keys=None)`: Return the parameter dictionaries of the given list of *nodes*. If *keys* is given, a list of values is returned instead. *keys* may also be a list, in which case the returned list contains lists of values.

`SetStatus(nodes, params, val=None)`: Set the parameters of *nodes* to *params*, which may be a single dictionary or a list of dictionaries. If *val* is given, *params* has to be the name of an attribute, which is set to *val* on the *nodes*. *val* can be a single value or a list of the same size as *nodes*.

Connections

`Connect(pre, post, weight=None, delay=None, model="static_synapse")`: Make one-to-one connections of type *model* between the nodes in *pre* and *post*. *pre* and *post* have to be lists of the same length. If *weight* is given (as dictionary or list of dictionaries), they are used as parameters for the connections. If *weight* is given as a single float or as list of floats, it is used as weight(s), in which case *delay* also has to be given as float or as list of floats.

`ConvergentConnect(pre, post, weight=None, delay=None, model="static_synapse")`: Connect all neurons in *pre* to each neuron in *post*. The other arguments are used as in `Connect()`.

`DivergentConnect(pre, post, weight=None, delay=None, model="static_synapse")`: Connect each neuron in *pre* to all neurons in *post*. The other arguments are used as in `Connect()`.

Both, `ConvergentConnect()` and `DivergentConnect()` have variants to establish a given number of random connections from *pre* to *post*.

Structured Networks

Nodes can be arranged in sub-networks to create hierarchical structures, very similar to how files are organized in directories. After importing PyNEST, a single empty sub-network exists. New sub-networks are created from the model *subnet*. New nodes are always created in the current sub-network.

`CurrentSubnet()`: Return the id of the current sub-network.

`ChangeSubnet(subnet)`: Make *subnet* the current sub-network.

`GetLeaves(subnet)`: Return all nodes under *subnet* that are not sub-networks.

`GetNodes(subnet)`: Return the complete list of nodes (including sub-networks) under *subnet*.

`GetNetwork(subnet, depth)`: Return a nested list of children of *subnet* up to *depth*.

`LayoutNetwork(model, dim, label=None, customdict=None)`: Create a *dim*-dimensional network of nodes of type *model*. *label* is an optional name for the network. If present, *customdict* is set as custom dictionary of the sub-network, which can be used to store custom information.

`BeginSubnet(label=None, customdict=None)`: Create a new sub-network and change into it. Optionally set *label* on the sub-network and *customdict* as custom dictionary.

`EndSubnet()`: Change to the parent subnet and return the id of the previously current one.

Simulation control

`Simulate(t)`: Simulate the network for *t* milliseconds.

`ResetKernel()`: Reset the simulation kernel. This will destroy the network as well as all custom models created with `CopyModel()`. This is equivalent to restarting NEST.

`ResetNetwork()`: Reset all nodes and connections to their original state.

`SetKernelStatus(params)`: Set kernel parameters like the temporal resolution of the simulation, the number of threads, data directory, etc.

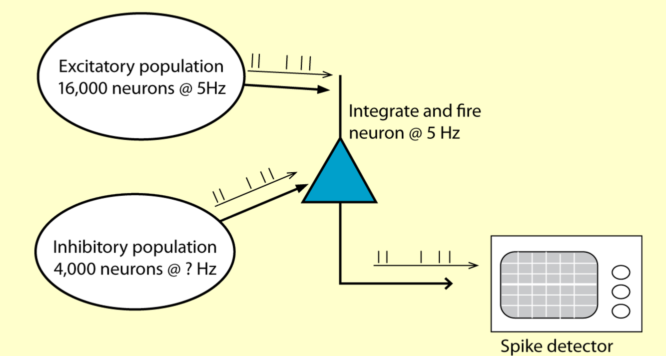
`GetKernelStatus()`: Returns a dictionary with the parameters of the simulation kernel.

`PrintNetwork(depth=1, subnet=None)`: Print the network tree up to *depth*, starting at *subnet*. If *subnet* is omitted, the current subnet is used instead.

A full journal publication describing the implementation of PyNEST is already submitted [3]

An example simulation

We illustrate the key features of PyNEST with a simulation of an integrate-and-fire neuron that receives input from an excitatory and an inhibitory neuron population. Both populations are modeled by Poisson spike generators. The simulation tries to find an appropriate firing rate for the neurons of the inhibitory population such that the target neuron fires at the same rate as the neurons of the excitatory population. (see figure.)



First, we import all necessary modules and functions into the namespace of our simulation script.

```
from nest import *
import nest.voltage_trace as voltage_trace
from scipy.optimize import bisect
```

Then we set the simulation parameters.

```
t_sim = 10000.0     # how long we simulate
n_ex  = 16000       # size of the excitatory population
n_in  =  4000       # size of the inhibitory population
r_ex  =     5.0     # mean rate of the excitatory population
r_in  =    12.5     # initial rate of the inhibitory population
epsc  =    45.0     # peak amplitude of excitatory synaptic currents
ipsc  =   -45.0     # peak amplitude of inhibitory synaptic currents
```

Now, we create the elements of the simulation and store the returned handles for later reference.

```
neuron        = Create("iaf_neuron")               # neuron        == [1]
noise         = Create("poisson_generator", 2)     # noise         == [2,3]
voltmeter     = Create("voltmeter")                # voltmeter     == [4]
spikedetector = Create("spike_detector")           # spikedetector == [5]
```

We adjust the parameters for the spike generators and for the voltmeter.

```
SetStatus(noise, [{"rate": n_ex*r_ex}, {"rate": n_in*r_in}])
SetStatus(voltmeter, {"interval": 10.0, "withgid": True})
```

Next, we connect the noise generators with the neuron, as well as the neuron with the spike detector and with the voltmeter.

```
ConvergentConnect(noise, neuron, [epsc, ipsc], 1.0)    # delay is 1.0 ms
Connect(voltmeter, neuron)
Connect(neuron, spikedetector)
```

To determine the optimal rate of the neurons in the inhibitory population, the network is simulated several times for different values of the inhibitory rate, while measuring the rate of the target neuron. This is done until the rate of the target neuron matches the rate of the neurons in the excitatory population with a certain precision. The algorithm is implemented in two steps:

First, a function `output_rate()` is defined to measure the firing rate of the target neuron for a given rate of the inhibitory population.

```
def output_rate(guess):
    rate = float(abs(n_in*guess))
    SetStatus([noise[1]], "rate", rate)
    SetStatus(spikedetector, "n_events", 0)
    Simulate(t_sim)
    r_target = GetStatus(spikedetector, "n_events")[0]*1000.0/t_sim
    print "  r_in=%.4f Hz, r_target=%.3f Hz" % (guess, r_target)
    return r_target
```

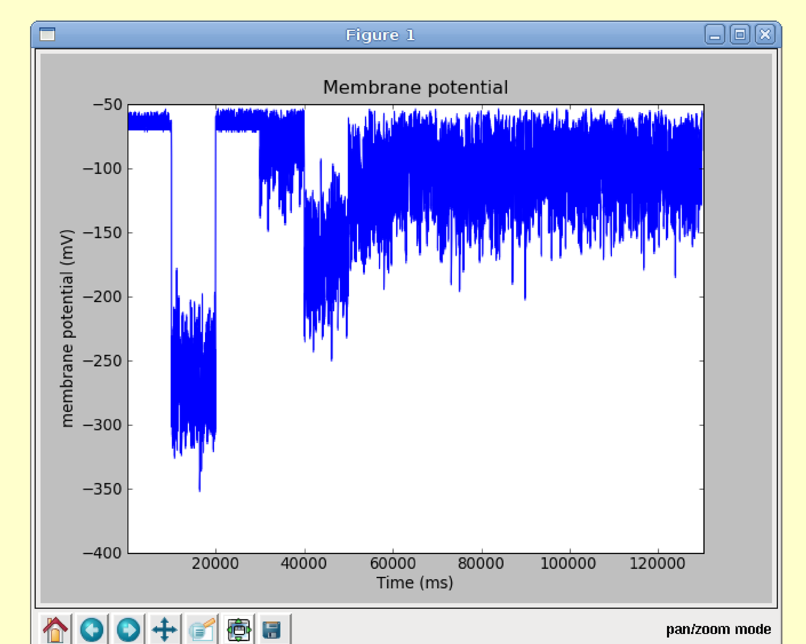
Second, the SciPy function `bisect()` is used to determine the optimal firing rate of the neurons of the inhibitory population. `bisect()` takes four arguments: a function whose zero crossing has to be determined, the lower and upper bound of the interval in which to search for the zero crossing, and the desired precision of the zero crossing.

```
print "Desired target rate: %.2f Hz" % r_ex
in_rate = bisect(lambda x: output_rate(x)-r_ex, 5.0, 25.0, xtol=0.01)
print "Resulting inhibitory rate: %.4f Hz" % in_rate
```

Finally, we plot the target neuron's membrane potential against time (see figure below).

```
voltage_trace.from_device(voltmeter)
```

```
NEST v1.9.svn (C) 1995-2008 NEST Initiative
Desired target rate: 5.00 Hz
  r_in=5.0000 Hz, r_target=434.400 Hz
  r_in=25.0000 Hz, r_target=0.300 Hz
  r_in=15.0000 Hz, r_target=347.500 Hz
  r_in=20.0000 Hz, r_target=36.000 Hz
  r_in=22.5000 Hz, r_target=0.000 Hz
  r_in=21.2500 Hz, r_target=1.300 Hz
     ⋮
  r_in=20.7422 Hz, r_target=6.800 Hz
  r_in=20.7617 Hz, r_target=6.900 Hz
  r_in=20.7715 Hz, r_target=4.800 Hz
Resulting inhibitory rate: 20.7715 Hz
```



The figure shows the output and the membrane potential during the different iterations of `bisect()`.

References

[1] M.-O. Gewaltig and M. Diesmann (2007). NEST (Neural Simulation Tool). Scholarpedia 2(4), 1430.

[2] H.E. Plesser et al. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. Proceedings of the EuroPar 2007.

[3] J.M. Eppler et al. PyNEST: A convenient interface to the NEST simulator. Frontiers in Neuroinformatics. (submitted)

The source code of NEST and its user manual is available on the homepage of the NEST Initiative at
`http://www.nest-initiative.org`