Pype9 Documentation

Release 0.2.0

Thomas G. Close and Andrew P. Davison

Apr 12, 2020

Contents

“PYthon PipelinEs for 9ml (Pype9)” is a collection of Python pipelines to simulate neuron, and neuron network, models described in NineML using either Neuron or NEST as simulator backends.

Pype9 has a *Command Line Interface* (CLI), which allows experiments to be simulated directly from NineML descriptions (i.e. without scripting). Alternatively, simulations can be embedded within Python scripts with Pype9’s accessible *Public API*.

CHAPTER 1

User/Developer Guide

1.1 Installation

Pype9 itself is a pure Python application and can be installed from the Python Package Index (PyPI) with Pip:

```
$ pip install pype9
```

If you would like to use the *plot* command you will also need to install matplotlib, which can be done separately or by specifying the 'plot' extra:

```
$ pip install pype9[plot]
```

With just the Python packages installed you will be able to use the `convert` and `plot` pipelines but in order to run simulations with Pype9 you will need to install at least one of the supported simulator backends (see below).

1.1.1 Simulator Backends

Pype9 currently works with the following simulator backends

- Neuron >= 7.5
- NEST >= 2.14.0

There are various configurations in which to install them, with the best choice dependent on your operating system/development configuration and your own personal preference.

Manual Installation from Source (Linux/MacOS)

Detailed instructions on how to install NEST can be found on the official NEST docs.

Good instructions on how to install Neuron from source can be found in Andrew Davisons notes.

Homebrew/Linuxbrew (MacOS/Linux)

Homebrew is a package manager that was developed for MacOS, but which has proven so successful that it has been ported to Linux (Linuxbrew) to complement in-built package managers (Linuxbrew installs packages to a users home directory).

The Pype9 command line interface (CLI), Neuron and NEST can all be installed using Homebrew in one line with:

```
$ brew install tclose/pype9/pype9
```

The following options can be provided to the formula

- with-python3 - Install Pype9, Neuron and NEST using Python 3
- with-mpi - Install Pype9, Neuron and NEST with MPI support

Warning: As of 2.14.0 NEST will need to be reinstalled using `brew reinstall NEST --HEAD` in order to include commit that installs the required C++ header files to the install prefix (instead of leaving them in the build directory, which is deleted after the build). In future versions of NEST this step will not be necessary.

Note that this Homebrew formula installs the Pype9 package and all its Python dependencies in a virtual environment inside the Homebrew *Cellar*. Therefore, if you would like to access Pype9's Python API you should only install the Neuron and NEST dependencies via Homebrew and Pype9 and its Python dependencies via Pip:

```
$ brew install --only-dependencies tclose/pype9/pype9
$ pip install pype9
```

or for Python 3:

```
$ brew install --only-dependencies tclose/pype9/pype9 --with-python3
$ pip3 install pype9
```

Please see the notes on how Homebrew handles Python, to ensure that you use the same installation for Neuron, NEST and Pype9, taking special note of the sections on bottling if not passing options to the build (i.e. `--with-python3` or `--with-mpi`).

If you don't have a strong preference for which Python you use I would recommend using a Homebrew Python installation (either 2 or 3, but probably 3 is best since support for Python 2 ends in 2020) as the system Python on MacOS has been slightly altered and can break some packages.

Note: To set Hombrew's Python 2 to be the default Python used from your terminal add `/opt/brew/opt/python/libexec/bin` to your PATH variable.

Install scripts (Linux/MacOS)

To install Neuron and NEST from source you can use the scripts that Pype9 uses to set up its automated testing environment, which can be found in the `install` directory of the Pype9 repo. For example, to install NEST 2.14.0 with Python 3 bindings to the prefix `/opt/nest/2.14.0`:

```
$ wget https://raw.githubusercontent.com/tclose/pype9/develop/install/nest.sh
$ ./nest.sh 2.14.0 3 /opt/nest/2.14.0
```

or Neuron 7.5:

```
$ wget https://raw.githubusercontent.com/tclose/pype9/develop/install/neuron.sh
$ ./neuron.sh 7.5 3 /opt/neuron/7.5
```

These install scripts also work well within a virtualenv, where they will install NEST and Neuron to the virtualenv prefix by default. This allows you to maintain different versions of Neuron, NEST on your system, which is useful when upgrading.

When installing to a virtualenv, the Python version and install prefix don't need to be supplied to the install scripts:

```
$ wget https://raw.githubusercontent.com/tclose/pype9/develop/install/nest.sh
$ wget https://raw.githubusercontent.com/tclose/pype9/develop/install/neuron.sh
$ pip install virtualenvwrapper
$ mkvirtualenv -p python3 pype9
$ ./nest.sh 2.14.0
$ ./neuron.sh 7.5
```

On Ubuntu, the installation requires the following packages

- build-essential
- autoconf
- automake
- libtool
- libreadline6-dev
- libncurses5-dev
- libgsl0-dev
- python-dev
- python3-dev
- openmpi-bin
- libopenmpi-dev
- inkscape
- libhdf5-serial-dev
- libyaml-dev

Similar packages can be found in other package managers on other distributions/systems (e.g. Homebrew).

Docker (Windows/Linux/MacOS)

A complete installation of Neuron, NEST and Pype9 (with MPI and against Python 3) can be found on the Docker image, https://hub.docker.com/r/tclose/pype9.

1. Install Docker (see https://docs.docker.com/engine/installation/)
2. Pull the Pype9 Docker image:

   ```
   $ docker pull tclose/pype9
   ```

3. Create a Docker container from the downloaded image:

   ```
   $ docker run -v `pwd`/<your-local-output-dir>:/home/docker/output \
       -t -i tclose/pype9 /bin/bash
   ```

This will create a folder called <*your-local-output-dir*> in the directory you are running the docker container, which you can access from your host computer (i.e. outside of the container) and view the output figures from.

4. From inside the running container, you will be able to run pype9, e.g.:

```
(pype9)docker@b3eca79b5209:~$ pype9 simulate \
    ~/catalog/neuron/HodgkinHuxley.xml#PyNNHodgkinHuxleyProperties \
    nest 500.0 0.001 \
    --init_value v 65 mV \
    --init_value m 0.0 unitless \
    --init_value h 1.0 unitless \
    --init_value n 0.0 unitless \
    --record v ~/output/hh-v.neo.pkl

(pype9)docker@b3eca79b5209:~$ pype9 plot ~/output/hh-v.neo.pkl \
    --save ~/output/hh-v.png
```

Supply the *–help* option to see a full list of options for each example.

5. Edit the xml descriptions in the ~/catalog directory to alter the simulated models as desired.

1.2 Command Line Interface

The Pype9 command line interface will be installed on your system path when Pype9 is installed with Pip (see *Installation*), otherwise it can be found in the `bin` directory of the repository.

In a similar style to many popular command line tools (e.g. Git, Pip, Homebrew, etc..) there is a single command, `pype9`, which is used to switch between different pipelines, i.e.:

```
$ pype9 <cmd> <options> <args>
```

There are currently four pipeline switches:

- simulate
- plot
- convert
- help

1.2.1 Simulate

Simulates a single cell defined by a 9ML Dynamics or DynamicsProperties, or a complete 9ML network, using either Neuron or NEST as the simulator backend.

Send ports and state-variables of the simulation can be recorded and saved to file in Neo format using the '–record' option, e.g.:

```
$ pype9 simulate my_cell.xml nest 100.0 0.01 \
  --record my_event_port ~/my_even_port.neo.pkl
```

For single-cell simulations, analog and event inputs stored in Neo format can be "played" into ports of the Dynamics class using the '–play' option e.g.:

```
$ pype9 simulate my_cell.xml nest 100.0 0.01 \
  --record my_event_port data-dir/my_even_port.neo.pkl \
  --play my_analog_receive_port data-dir/my_input_current.neo.pkl
```

Properties, initial values and the initial regime (for single cells) can be overridden with the '–prop', '–initial_value' and '–initial_regime' respectively and must be provided for every parameter/state-variable if they are not in the model description file.

```
usage: pype9 simulate [-h] [--prop PARAM VALUE UNITS]
                      [--init_regime INIT_REGIME]
                      [--init_value STATE-VARIABLE VALUE UNITS]
                      [--record RECORD [RECORD ...]] [--play PORT FILENAME]
                      [--seed SEED] [--properties_seed PROPERTIES_SEED]
                      [--min_delay DELAY UNITS] [--device_delay DELAY UNITS]
                      [--build_mode BUILD_MODE] [--build_dir BUILD_DIR]
                      [--build_version BUILD_VERSION]
                      model {neuron,nest} time timestep
```

Positional Arguments

model	Path to nineml model file which to simulate. It can be a relative path, absolute path, URL or if the path starts with '//' it will be interpreted as a ninemlcatalog path. For files with multiple components, the name of component to simulated must be appended after a #, e.g. //neuron/izhikevich#izhikevich
simulator	Possible choices: neuron, nest Which simulator backend to use
time	Time to run the simulation for (ms)
timestep	Timestep used to solve the differential equations (ms)

Named Arguments

--prop	Set the property to the given value Default: []
--init_regime	Initial regime for dynamics
--init_value	Initial regime for dynamics Default: []
--record	Record the values from the send port or state variable and the filename to save it into. Each record option can have either 2 or 4 arguments: PORT/STATE-VARIABLE FILENAME [T_START T_START_UNITS] Default: []
--play	Name of receive port and filename with signal to play it into Default: []
--seed	Random seed used to create network and properties
--properties_seed	Random seed used to create network connections and properties. If not provided it is generated from the '–seed' option.

--min_delay	The minimum delay of the model (only applicable for single cell NEST simulations)
--device_delay	The delay applied to signals played into ports of the model (only applicable for NEST simulations)
--build_mode	The strategy used to build and compile the model. Can be one of 'lazy', 'force', 'require', 'build_only', 'generate_only', 'purge' (default "lazy") Default: "lazy"
--build_dir	Base build directory
--build_version	Version to append to name to use when building component classes

Note: To simulate network simulations on Neuron over multiple cores you need to use the MPI command `mpirun -n <ncores> pype9 simulate <options>` and have installed Neuron with the `--with-mpi` option (see *Installation*)

1.2.2 Plot

Simple tool for plotting the output of PyPe9 simulations using Matplotlib. Since Pype9 output is stored in Neo format, it can be used to plot generic Neo files but it also includes handling of Pype9-specific annotations, such as regime transitions.

```
usage: pype9 plot [-h] [--save SAVE] [--dims WIDTH HEIGHT] [--hide]
                  [--resolution RESOLUTION]
                  filename
```

Positional Arguments

filename	Neo file outputted from a PyPe9 simulation

Named Arguments

--save	Location to save the figure to
--dims	Dimensions of the plot Default: (10, 8)
--hide	Whether to show the plot or not Default: False
--resolution	Resolution of the figure when it is saved Default: 300.0

1.2.3 Convert

Tool to convert 9ML files between different supported formats (e.g. XML, JSON, YAML) and 9ML versions.

```
usage: pype9 convert [-h] [--nineml_version NINEML_VERSION] in_file out_file
```

Positional Arguments

in_file	9ML file to be converted
out_file	Converted filename

Named Arguments

--nineml_version, -v	The version of nineml to output

1.2.4 Help

Prints help information associated with a PyPe9 command

```
usage: pype9 help [-h] cmd
```

Positional Arguments

cmd	Name of the command to print help information

Examples

The available pipelines can be listed with:

```
$ pype9 help
usage: pype9 <cmd> <args>

available commands:
    convert
        Converts a 9ML file from one supported format to another
    help
        Prints help information associated with a PyPe9 command
    plot
        Convenient script for plotting the output of PyPe9 simulations (actually not
        9ML specific as the signals are stored in Neo format)
    simulate
        Runs a simulation described by an Experiment layer 9ML file
```

More detailed help messages for each available pipeline can be viewed by supplying its name to the help:

```
$ pype9 help plot
usage: pype9 plot [-h] [--save SAVE] [--dims WIDTH HEIGHT] [--hide]
               [--resolution RESOLUTION]
               filename

Convenient script for plotting the output of PyPe9 simulations (actually not
9ML specific as the signals are stored in Neo format)

positional arguments:
  filename              Neo file outputted from a PyPe9 simulation

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --save SAVE           Location to save the figure to
  --dims WIDTH HEIGHT   Dimensions of the plot
  --hide                Whether to show the plot or not
  --resolution RESOLUTION
                        Resolution of the figure when it is saved
```

1.3 Creating Simulations in Python

The Pype9 package is organised into sub-packages loosely corresponding to each pipeline (e.g. `simulate`, `plot`). The `simulate` package contains the sub-packages, `neuron` and `nest`, which provide the simulator-specific calls to their respective backends.

All classes required to design and run simulations in these packages derive from corresponding classes in the `common` package, which defines a consistent *Public API* across all backends. Therefore, code designed to run on with one backend can be switched to another by simply changing the package the simulator-specific classes are imported from (like in PyNN).

Note: The `neuron` and `nest` packages can be imported separately. Therefore, only the simulator you plan to use needs to be available on your system.

1.3.1 Simulation Control

Simulation parameters such as time step, delay limits and seeds for pseudo random number generators are set within an instance of the *Simulation* class. Simulator objects (e.g. cells and connections) can only be instantiated within the context of an active *Simulation* instance, and there can only be one active *Simulation* instance at any time.

A *Simulation* is activated with the `with` keyword

```
with Simulation(dt=0.1 * un.ms, seed=12345) as sim:
    # Design simulation here
```

The simulation is advanced using the `run` method of the *Simulation* instance

```
with Simulation(dt=0.1 * un.ms, seed=12345) as sim:
     # Create simulator objects here
     sim.run(100.0 * un.ms)
```

this can be done in stages if states or parameters need to be updated mid-simulation

```
with Simulation(dt=0.1 * un.ms, seed=12345) as sim:
     # Create simulator objects here
     sim.run(50.0 * un.ms)
     # Update simulator object parameters/state-variables
     sim.run(50.0 * un.ms)
```

After the simulation context exits all objects in the simulator backend are destroyed (unless an exception is thrown) and only recordings can be reliably accessed from the "dead" Pype9 objects.

There are pseudo-random number generator (RNG) seeds that can be passed to *Simulation* instances, `seed` which is used to seed the RNGs that generate random dynamic processes during the simulation (e.g. poisson processes), and `properties_seed` which is used to seed the RNG used to generate probabilistic connectivity rules and the random distribution of cell properties over populations. If `seed` is None (the default) then it is generated from the current timestamp, if `properties_seed` is None then it is derived from `seed`.

1.3.2 Cell Simulations

NineML Dynamics classes can be translated into simulator cell objects using the *CellMetaClass* class. A metaclass is class of classes, i.e. one whose instantiation is itself a class, such as the `type` class. *CellMetaClass* instantiations derive from the *Cell* class and can be used to represent different classes of neural models, such as Izhikevich or Hodgkin-Huxley for example. From these *Cell* classes as many cell instances (with their corresponding simulator objects) can be created as required e.g:

```
# Create Izhikevich cell class by instantiating the CellMetaClass with a
# ninml.Dynamics Izhikevich model
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
# Parameters and states of the cell class must be provided when the cells
# are instantiated.
# either as keyword args
izhi1 = Izhikevich(a=1, b=2, c=3, d=4, v=-65 * un.mV, u=14 * un.mV / un.ms)
# or from a nineml.DynamicsProperties object
izhi3 = Izhikevich('./izhikevich.xml#IzhikevichBurster')
```

If the specified Dynamics class has not been built before the *CellMetaClass* will automatically generate the required source code for the model, compile it, and load it into the simulator namespace. This can happen either inside or outside of an active *Simulation* instance. However, the cells objects themselves must be instantiated within a *Simulation* instance.

```
# The cell class can be created outside the simulation context
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
with Simulation(dt=0.1 * un.ms) as sim:
    # The cell object must be instantiated within the simulation context
    izhi = Izhikevich(a=1, b=2, c=3, d=4, v=-65 * un.mV,
                      u=14 * un.mV / un.ms)
    sim.run(1000.0 * un.ms)
```

The data can be recorded from every send port and state variable in the NineML Dynamics class using the `record` method of the *Cell* class. The recorded data can then be accessed with the `recording` method. The recordings will be Neo format.

```
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
with Simulation(dt=0.1 * un.ms) as sim:
    izhi = Izhikevich(a=1, b=2, c=3, d=4, v=-65 * un.mV,
                      u=14 * un.mV / un.ms)
    # Specify the variables to record
    izhi.record('v')
    sim.run(1000.0 * un.ms)
# Retrieve the recording
v = izhi.recording('v')
```

Transitions between regimes can be recorded using `record_regime` and retrieved using `regime_epochs`

```
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
with Simulation(dt=0.1 * un.ms) as sim:
    izhi = Izhikevich(a=1, b=2, c=3, d=4, v=-65 * un.mV,
                      u=14 * un.mV / un.ms)
    # Specify the variables to record
    izhi.record_regime()
```

(continues on next page)

(continued from previous page)

```
    sim.run(1000.0 * un.ms)
# Retrieve the regime changes
epochs = izhi.regime_epochs()
```

Data in Neo format can be "played" into receive ports of the *Cell*

```
neo_data = neo.PickleIO('./data/my_recording.neo.pkl').read()
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
with Simulation(dt=0.1 * un.ms) as sim:
    izhi = Izhikevich(a=1, b=2, c=3, d=4, v=-65 * un.mV,
                      u=14 * un.mV / un.ms)
    # Play analog signal (must be of current dimension) into 'i_syn'
    # analog-receive port.
    izhi.play('i_syn', neo_data.analogsignals[0])
    sim.run(1000.0 * un.ms)
```

States and parameters can be accessed and set using the attributes of the *Cell* objects

```
Izhikevich = CellMetaClass('./izhikevich.xml#Izhikevich')
with Simulation(dt=0.1 * un.ms) as sim:
    izhi = Izhikevich(a=1, b=2, c=3, d=4)
    sim.run(500.0 * un.ms)
    # Update the membrane voltage after 500 ms to 20 mV
    izhi.v = 20 * un.mV
    sim.run(500.0 * un.ms)
```

Event ports can be connected between individual cells

```
Poisson = CellMetaClass('./poisson.xml#Poisson')
LIFAlphSyn = CellMetaClass('./liaf_alpha_syn.xml#LIFAlphaSyn')
with Simulation(dt=0.1 * un.ms) as sim:
    poisson = Poisson(rate=10 * un.Hz, t_next=0.5 * un.ms)
    lif = LIFAlphaSyn('./liaf_alpha_syn.xml#LIFAlphaSynProps')
    # Connect 'spike_out' event-send port of the poisson cell to
    # the 'spike_in' event-receive port on the leaky-integrate-and-fire
    # cell
    lif.connect(poisson, 'spike_out', 'spike_in')
    sim.run(1000.0 * un.ms)
```

1.3.3 Network Simulations

Network simulations are specified in much the same way as *Cell Simulations*, with the exception that there is no metaclass for Networks (Network metaclasses will be added when the "Structure Layer" is introduced in NineML v2). Therefore, *Network* objects need to be instantiated within the simulation context.

```
with Simulation(dt=0.1 * un.ms) as sim:
    # Network objects need to be instantiated within the simulation context
    network = Network('./brunel/AI.xml#AI')
    sim.run(1000.0 * un.ms)
```

During construction of the network, the NineML Populations and Projections are flattened into Component Array and Connection Group objects such that the synapse dynamics in the projection are included in the dynamics of the Component Array and each port connection is converted into a separate Connection Group of static connections.

To record data, the relevant component array needs to be accessed using the `component_array` or `component_arrays` accessors of the network class. Then as in the *Cell Simulations* case the `record` method is used to specify which variables to record and the `recording` method is used to access the recording after the simulation.

```
with Simulation(dt=0.1 * un.ms) as sim:
    network = Network('./brunel/AI.xml#AI')
    # 'spike_out' is explicitly connected in the connection so it is
    # mapped to the global namespace of the flattened cell + synapses model
    network.component_array('Exc').record('spike_out')
    # State-variables of the cell dynamics are suffixed with '__cell'
    network.component_array('Inh').record('v__cell')
    # State-variables of synapses, in this case synapses from the
    # 'Inhibition' projection, are prefixed with '__<projection-name>'
    network.component_array('Exc').record('a__Inhibition')
    sim.run(1000.0 * un.ms)
exc_spikes = network.component_array('Exc').recording('spike_out')
inh_v = network.component_array('Inh').recording('v__cell')
exc_inh_a = network.component_array('Exc').recording('a__Inhibition')
```

Note: During the cell and synapse flattening process the names of state variables and unconnected ports will be suffixed with `__cell` if they belong to the population dynamics or __`<my-projection>` if they belong to the synapse of the a projection

Network models are simulated via integration with PyNN and therefore will run on multiple processes using Open MPI (and *Open MP_* for NEST) if the calling Python script is run with `mpirun/mpiexec`.

1.4 Public API

The Pype9 public API consists of seven classes required to create simulations of individual neurons or neural networks described in NineML. All classes in the public API have an abstract base class in the `pype9.simulate.common` module and matching derived simulator-specific classes in the `pype9.simulate.neuron` and `pype9.simulate.nest` modules.

As the simulator-specific classes have the same signatures as those in the base module only the base module classes are described here.

1.4.1 Simulation

class `pype9.simulate.common.simulation.`**`Simulation`**(*dt*, *t_start=0.0 * s*, *seed=None*, *properties_seed=None*, *min_delay=1.0 * ms*, *max_delay=10.0 * ms*, *code_generator=None*, *build_base_dir=None*, ***options*)

Base class of all simulation classes that prepares and runs the simulator kernel. All simulator objects must be created within the context of a Simulation instance.

```
with Simulation(dt=0.1 * un.ms, seed=12345) as sim:
    # Design simulation here
```

The simulation is advanced using the `run` method

```
with Simulation(dt=0.1 * un.ms, seed=12345) as sim:
    # Create simulator objects here
    sim.run(100.0 * un.ms)
```

After the simulation context exits all objects in the simulator backend are destroyed (unless an exception is thrown) and only recordings can be reliably accessed from the "dead" Pype9 objects.

Parameters

- **`dt`** (*`nineml.Quantity (time)`*) – The resolution of the simulation
- **`t_start`** (*`nineml.Quantity (time)`*) – The time to start the simulation from
- **`seed`** (*`int | None`*) – The seed with which to construct the cell/network properties. NB: This seed will only reproduce constant results if the number of MPI nodes is constant
- **`properties_seed`** (*`int | None`*) – The seed used for random number generator used to set properties and generate connectivity. If not provided it will be derived from the 'seed' argument. NB: This seed will only reproduce constant results if the number of MPI nodes is constant
- **`min_delay`** (*`nineml.Quantity (time) | None`*) – The minimum delay in the network. If None the min delay will be calculated from the first network to be created (if a single cell then it will be the same as the timestep)
- **`max_delay`** (*`nineml.Quantity (time) | None`*) – The maximum delay in the network. If None the max delay will be calculated from the first network to be created (if a single cell then it will be the same as the timestep)
- **`options`** (*`dict(str, object)`*) – Options passed to the simulator-specific methods

`run`(*self*, *t_stop*, ***kwargs*)

Run the simulation until time `t_stop`.

Parameters **`t_stop`** (*`nineml.Quantity (time)`*) – The time to run the simulation until

1.4.2 CellMetaClass

class `pype9.simulate.common.cells.`**`CellMetaClass`**(*component_class*, ***kwargs*)

Metaclass for creating simulator-specific cell classes from 9ML Dynamics classes. Instantiating a CellMetaClass with a `nineml.Dynamics` instance will generate, compile and load the required simulator-specific code and create a class that can be used to instantiate dynamics objects.

Parameters

- **`component_class`** (*`nineml.Dynamics`*) – The 9ML component class to create the Cell class for
- **`name`** (*`str`*) – The name of the cell class, which is used for the generated simulator code. If None, the name of the component_class is used. Note, names must be unique among classes loaded within the same simulation script.

1.4.3 Cell

class `pype9.simulate.common.cells.`**`Cell`**(**args*, ***kwargs*)

Base class for all cell classes created from the CellMetaClass. It defines all methods that can be called on cell model objects.

Parameters

- **prototype** (*DynamicsProperties*) – A dynamics properties object used as the "prototype" for the cell
- **regime** (*str*) – Name of regime the cell will be initiated in
- **kwargs** (*dict(str, nineml.Quantity)*) – Properties and initial state variables to initiate the cell with. These will override properties/initial-values in the prototype

connect (*self*, *sender*, *send_port_name*, *receive_port_name*, *delay*, *properties=[]*)
Connects an event send port from other into an event receive port in the cell

Parameters

- **sender** (*pype9.simulator.base.cells.Cell*) – The sending cell to connect the from
- **send_port_name** (*str*) – Name of the port in the sending cell to connect to
- **receive_port_name** (*str*) – Name of the receive port in the current cell to connect from
- **delay** (*nineml.Quantity (time)*) – The delay of the connection
- **properties** (*list(nineml.Property)*) – The connection properties of the event port

play (*self*, *port_name*, *signal*, *properties=[]*)
Plays an analog signal or train of events into a port of the cell

Parameters

- **port_name** (*str*) – The name of the port to play the signal into
- **signal** (*neo.AnalogSignal | neo.SpikeTrain*) – The signal to play into the cell
- **properties** (*dict(str, nineml.Quantity)*) – Connection properties when playing into a event receive port with static connection properties

record (*self*, *port_name*, *t_start=None*)
Specify the recording of a send port or state-variable before the simulation.

record_regime (*self*)
Returns the current regime at each timestep. Periods spent in each regimes can be retrieved with the `regime_epochs` method.

recording (*self*, *port_name*, *t_start=None*)
Return recorded data as a dictionary containing one numpy array for each neuron, ids as keys.

Parameters **port_name** (*str*) – Name of the port to retrieve the recording for

regime_epochs (*self*)
Retrieves the periods spent in each regime during the simulation in a neo.core.EpochArray

1.4.4 Network

class `pype9.simulate.common.network.`**Network** (*nineml_model*, *build_mode='lazy'*, ***kwargs*)
Constructs a network simulation, generating and compiling dynamics classes as required (depending on the 'build_mode' option). The populations and projections of the network are flattened so that the synapse projections are included in the cell dynamics and the connection groups are just static connections.

Parameters **`nineml_model`** (*`nineml.Network | nineml.Document | URL`*) – A 9ML-Python model of a network (or Document containing populations and projections for 9MLv1) or a URL referring to a 9ML model.

`component_array`(*self*, *name*)
Returns the component array matching the given name

Parameters **`name`** (*`str`*) – Name of the component array

`component_arrays`
Iterate through component arrays

`connection_group`(*self*, *name*)
Returns the connection group matching the given name

Parameters **`name`** (*`str`*) – Name of the component array

`connection_groups`
Iterate through connection_groups

`selection`(*self*, *name*)
Returns the selection matching the given name

Parameters **`name`** (*`str`*) – Name of the selection

`selections`
Iterate through selections

1.4.5 ComponentArray

class `pype9.simulate.common.network.`**`ComponentArray`**(*nineml_model*, *build_mode='lazy'*, ***kwargs*)
Component array object corresponds to a NineML type to be introduced in NineMLv2 (see https://github.com/INCF/nineml/issues/46), which consists of a dynamics class and a size. Populations and the synapses on incoming projections.

Parameters

- **`nineml_model`** (*`nineml.ComponentArray`*) – Component array nineml
- **`build_mode`** (*`str`*) – The build/compilation strategy for rebuilding the generated code, can be one of 'lazy', 'force', 'build_only', 'require'.

`play`(*self*, *port_name*, *signal*, *properties=[]*)
Plays an analog signal or train of events into a port of the dynamics array.

Parameters

- **`port_name`** (*`str`*) – The name of the port to play the signal into
- **`signal`** (*`neo.AnalogSignal | neo.SpikeTrain`*) – The signal to play into the cell
- **`properties`** (*`dict(str, nineml.Quantity)`*) – Connection properties when playing into a event receive port with static connection properties

`record`(*self*, *port_name*, *t_start=None*)
Records the port or state variable

Parameters **`port_name`** (*`str`*) – Name of the port to record

`recording`(*self*, *port_name*, *t_start=None*)
Returns the recorded data for the given port name

Parameters **port_name** (*str*) – The name of the port (or state-variable) to retrieve the recorded data for

Returns **recording** – The recorded data in a neo.Segment

Return type neo.Segment

1.4.6 Selection

class pype9.simulate.common.network.**Selection**(*nineml_model*, **component_arrays*)

A selection of cells from one or multiple component arrays. Used to connect ConnectionGroup.

Parameters

- **nineml_model** (*nineml.Selection*) – The NineML Selection object
- **component_arrays** (*list(nineml.ComponentArray)*) – List of component arrays included in the selection.

1.4.7 ConnectionGroup

class pype9.simulate.common.network.**ConnectionGroup**(*nineml_model*, *source*, *destination*)

ConnectionGroup object corresponds to a NineML type to be introduced in NineMLv2 (see https://github.com/INCF/nineml/issues/46), which consists of a dynamics class and a size. Only consists of references to ports on the source and destination ComponentArrays|Selections and connectivity.

Parameters

- **nineml_model** (*nineml.ConnectionGroup*) – Component array nineml
- **source** (ComponentArray) – Source component array
- **destination** (ComponentArray) – Destination component array

1.5 Developer Documentation

Contributions to Pype9 are most welcome! To contribute simply fork Pype9 on Github and create a pull request when you are ready to merge your code.

Contributions are required to: * Strictly adhere to Pep8 (PyLint is useful for checking this) * Not significantly decrease the code coverage. Meaning that you will probably need to add a unittest or two.

1.5.1 Additional Simulator Backends

An area in which Pype9 can be extended is by adding support for additional simulator backends (e.g. Brian2, GeNN), including any that you may be developing.

One of the key benefits of clearly separating the model description language (i.e. NineML), and the associated NineML Python library, from simulator-specific code is that provides a clearly defined interface for adding support for additional simulators. Therefore, extending Pype9 is definitely not a requirement for adding NineML support for your simulator (and from a language perspective a diversity of independent tools is desirable). However, since Pype9 has already implemented many of the menial tasks involved in working with NineML models, such as managing the build process and file IO, you may find it convenient to extend Pype9's base classes.

In order to extend Pype9's base classes you will need to override the following abstract methods.

Single Cell Simulations

CodeGenerator

- generate_source_files
- configure_build_files
- compile_source_files
- clean_src_dir
- clean_compile_dir
- load_libraries

Cell

- _get
- _set
- _set_regime
- record
- record_regime
- recording
- _regime_recording
- reset_recordings
- play
- connect

Network Simulations

At this stage Pype9's network simulations are handled by PyNN so it is probably only practical to add support for simulators that are supported by PyNN. Future versions of Pype9 will likely introduce base classes for network objects, which can be extended though.

1.5.2 Additional Pipelines

Pype9 has been designed to be a general umbrella containing a wide range of pipelines for manipulating and working with NineML models in addition to running simulations.

Import Simulator-Specific Models

High on the list of useful additional pipelines are importer pipelines that take models, or part thereof, written in simulator specific code and translate them into NineML.

For example, a prototype **NMODL_** and Neuron "model view" importers exists in the branch *neuron_import* at http://github.com/tclose/pype9. It would be great to extend this to other popular simulators such as NEST and Brian.

Graphical User Interface

Another key benefit of separating the model description from the simulation code is that it greatly simplifies the creation of graphical user interfaces with which to create models with. With the NineML Python Library able to read JSON files, a promising approach would be to create a javascript GUI that can read, modify and write NineML files to JSON that could then be simulated with different backends.

1.6 Unsupported 9ML

NineML aims to be a comprehensive description language for neural simulation. This means that it allows the expression of some uncommon configurations that are difficult to implement in Neuron and NEST. Work is planned to make the Neuron and NEST pipelines in Pype9 fully support NineML, however until then the following restrictions apply to models that can be used with Pype9.

- synapses must be linear (to be relaxed in v0.2)
- synapses can only have one variable that varies over a projection (e.g. weight) (to be relaxed in v0.2)
- no analog connections between populations (i.e. gap junctions) (gap junctions to be implemented in v0.2)
- only one event send port per cell (current limitation of Neuron/NEST)
- names given to NineML elements are not escaped and therefore can clash with built-in keywords and some PyPe9 method names (e.g. 'lambda' is a reserved keyword in Python). Please avoid using names that clash with C++ or Python keywords (all 9ML names will be escaped in PyPe9 v0.2).

1.7 Getting help

Mailing list: nineml-users@incf.org

NeuralEnsemble Google group

If you find a bug or would like to add a new feature to Pype9 package, please go to https://github.com/NeuralEnsemble/Pype9/issues/. First check that there is not an existing ticket for your bug or request, then click on "New issue" to create a new ticket (you will need a GitHub account, but creating one is simple and painless). Please add the label "Python" to the ticket.

Index