NineML Python library

Release 0.3dev

Andrew P. Davison Thomas G. Close, Mike Hull, and Eilif Muller,

Jan 22, 2018

Contents

NineML is a language for describing the dynamics and connectivity of neuronal network simulations; in particular for large-scale simulations of many point neurons.

The language is defined as an object model, described in the NineML specification, with standardized serializations to XML, JSON, YAML and HDF5.

This documentation describes the `nineml` Python package, which implements the NineML object model using Python classes, allowing models to be created, edited, introspected, etc. using Python, and then written to/read from the NineML XML format.

CHAPTER 1

Users' guide

1.1 Motivation

1.1.1 Why NineML?

NineML (or "9ML") is a language for describing the dynamics and connectivity of neuronal network simulations; in particular for large-scale simulations of many point neurons (where the neuron model does not explicitly represent dendrites).

At present, networks of point-neurons are typically simulated by writing either a custom simulation program in a general-purpose programming language (such as Python, MATLAB) or by writing a model for a particular simulator (NEURON, NEST, Brian, etc.) As models of neuronal dynamics and connectivity become more and more complex, writing a simulation from scratch in Python or Matlab can become more and more complex, taking time to debug and producing hard to find bugs. Writing simulator-specific models can reduce some of this duplication, but this means the model will only run on a single simulator and is hence difficult to share.

Programmatic model description APIs such as PyNN provide simulator independence at the expense of (i) having to choose from a limited library of neuron models (note however that PyNN now works with neuron/synapse models defined in NineML, for certain simulators), (ii) being tied to a particular programming language. Having access to a full programming language is also a temptation to writing over-complex, difficult to maintain model descriptions when compared to a declarative language such as NineML.

NineML tries to mitigate some of these problems by providing an language for defining smaller components of a simulation in a declarative, language-independent way. Various tools are then available for generating code for various simulators from this description (see http://nineml.net/software).

Note: NineML and NeuroML version 2 are both languages for mathematically-explicit descriptions of biological neuronal network models. NineML currently works only for point-neuron/single-compartment neuron models, while NeuroML also supports multi-compartment, morphologically-detailed models. The two languages evolved in parallel, although with considerable cross-influence in both directions. It is possible they will merge in future; tools are under development to allow conversion between the formats where possible. Which one you should choose depends largely on what you want to do, and what tools are available for working with the two languages.

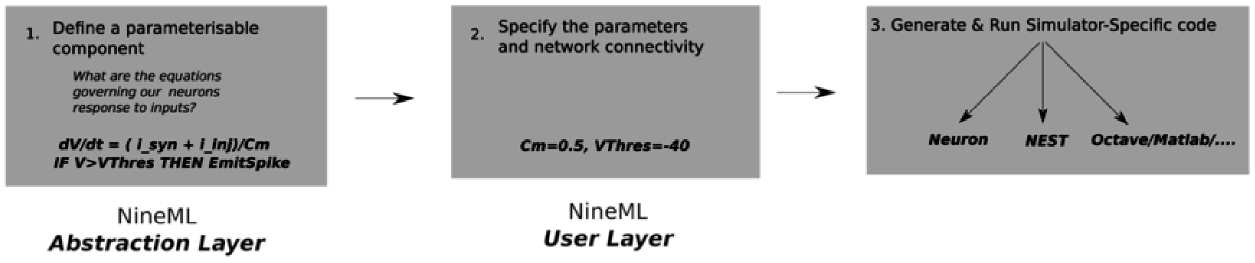
1.1.2 *Abstraction* and *User* Layers

In NineML, the definition of a component is split into two parts;

Abstraction Layer Components on this layer can be thought of as parameterised models. For example, we could specify a general integrate-and-fire neuron, with a firing threshold, `V_Threshold` and a reset voltage `V_Reset`. We are able to define the dynamics of the neuron in terms of these parameters.

User Layer In order to simulate a network, we need to take the *parameterised* models from the *Abstraction Layer*, fill in the parameters, and specify the number of each type of component we wish to simulate and how they should be connected. For example, we might specify for our neurons that `V_Threshold` was -45 mV and `V_Reset` was -60 mV.

The flow for a simulation using NineML would look like:


1. Define a parameterisable component
What are the equations governing our neurons response to inputs?
dV/dt = (I_syn + I_inj)/Cm
IF V>VThres THEN EmitSpike
NineML
Abstraction Layer
2. Specify the parameters and network connectivity
Cm=0.5, VThres=-40
NineML
User Layer
3. Generate & Run Simulator-Specific code
Neuron
NEST
Octave/Matlab/....

An obvious question is *"Why do this?!"*

For a single, relatively simple simulation, it may not be worth the effort! But imagine we are modelling a (relatively simple) network of neurons, which contains five different types of neurons. The neurons synapse onto each other, and there are three different classes of synapses, with different models for their dynamics. If we were to implement this naively, we could potentially copy and paste the same code 15 times, *for each simulator*. By factoring out basic functionality, we make our workflow much more manageable.

1.1.3 The `nineml` Python library

NineML is defined by an object model (the specification can be found at nineml.net), with standardized serializations to XML, JSON, YAML and HDF5. The Python `nineml` library provides tools for reading and writing NineML models to and from the supported serialization formats and an API for building/introspecting/manipulating/validating NineML models in Python (including a shorthand notation for building NineML models). The library is intended as a base for other Python tools working with NineML, for example tools for code generation.

1.2 Installation

Use of the Python 9ML API requires that you have Python (version 2.7 or >=3.4) with the `sympy` package installed. To serialize NineML to XML, YAML and HDF5 formats the `lxml`, `pyyaml` and `h5py` packages are also required respectively.

1.2.1 Depdendencies

macOS

If you are not already using another Python installation (e.g. Enthought, Python(x,y), etc...) it can be a good idea to install Python using the Homebrew package manager rather than using the system version as Apple has modified some package versions (e.g. `six`), which can cause difficulties down the track.

```
$ brew install python
```

While other Python installations should work fine, it is not recommended to use the system Python installation at */usr/bin/python* for scientific computing as some of the standard pacakges (e.g. *six*) have been modified and this can cause problems with other packages down the track.

Before installing *h5py* you will also need to install a development version of HDF5. With Homebrew this can be done with:

```
$ brew install hdf5
```

Linux

On Linux, development packages for HDF5 (i.e. with headers). For Ubuntu/Debian the following packages can be used

- libhdf5-serial-dev (serial)
- libhdf5-openmpi-dev (parallel with Open MPI)
- libhdf5-mpich-dev (parallel with MPICH)

Please consult the relevant documentation to find the appropriate package for other distributions.

Windows

On Windows, you can download the Python installer from http://www.python.org. To use HDF5 serialisation you will need to install HDF5 from source, see http://docs.h5py.org/en/latest/build.html.

1.2.2 Install Python packages

To install the Python package it is recommeded to install from PyPI using *pip*:

```
$ pip install nineml
```

Otherwise for the latest version you can clone the repository at http://github.com/INCF/nineml-python or install directly with:

```
$ pip install git+http://github.com/INCF/nineml-python
```

1.3 Getting started

1.3.1 Reading model descriptions from XML files

NineML documents can contain abstraction layer models, user layer models (with references to abstraction layer models defined in other documents) or both.

To read a file containing only abstraction layer elements:

```
>>> import nineml, pprint
>>> doc = nineml.read("./BrunelIaF.xml")
>>> pprint(doc.items())
[('BrunelIaF', Dynamics(name='BrunelIaF')),
 ('current', Dimension(name='current', i=1)),
 ('resistance', Dimension(name='resistance', i=-2, m=1, t=-3, l=2)),
 ('time', Dimension(name='time', t=1)),
 ('voltage', Dimension(name='voltage', i=-1, m=1, t=-3, l=2)]
```

This gives us a `Document` instance, a dictionary-like object containing a `Dynamics` definition of an integrate-and-fire neuron model, together with the definitions of the physical dimensions of parameters and state variables used in the model.

Now for a file containing an entire user layer model (with references to other NineML documents containing the abstraction layer definitions):

```
>>> doc = nineml.read("./network/Brunel2000/AI.xml")
>>> pprint(doc.items())
[('All': Selection(name='All')),
 ('Exc': Population(name='Exc', number=4000, cell=nrn)),
 ('Excitation': Projection(name="Excitation", source=Population(name='Exc',␣
↪number=4000, cell=nrn), destination=Selection(name='All'),␣
↪connectivity=BaseComponent(name="RandomExc", componentclass="RandomFanIn"),␣
↪response=BaseComponent(name="syn", componentclass="AlphaPSR
↪")plasticity=BaseComponent(name="ExcitatoryPlasticity", componentclass=
↪"StaticConnection"), delay=Delay(value=1.5, unit=ms), with 2 port-connections)),
 ('Ext': Population(name='Ext', number=5000, cell=stim)),
 ('External': Projection(name="External", source=Population(name='Ext', number=5000,␣
↪cell=stim), destination=Selection(name='All'), connectivity=BaseComponent(name=
↪"OneToOne", componentclass="OneToOne"), response=BaseComponent(name="syn",␣
↪componentclass="AlphaPSR")plasticity=BaseComponent(name="ExternalPlasticity",␣
↪componentclass="StaticConnection"), delay=Delay(value=1.5, unit=ms), with 2 port-
↪connections)),
 ('Hz': Unit(name='Hz', dimension='per_time', power=0)),
 ('Inh': Population(name='Inh', number=1000, cell=nrn)),
 ('Inhibition': Projection(name="Inhibition", source=Population(name='Inh',␣
↪number=1000, cell=nrn), destination=Selection(name='All'),␣
↪connectivity=BaseComponent(name="RandomInh", componentclass="RandomFanIn"),␣
↪response=BaseComponent(name="syn", componentclass="AlphaPSR
↪")plasticity=BaseComponent(name="InhibitoryPlasticity", componentclass=
↪"StaticConnection"), delay=Delay(value=1.5, unit=ms), with 2 port-connections)),
 ('Mohm': Unit(name='Mohm', dimension='resistance', power=6)),
 ('current': Dimension(name='current', i=1)),
 ('mV': Unit(name='mV', dimension='voltage', power=-3)),
 ('ms': Unit(name='ms', dimension='time', power=-3)),
 ('nA': Unit(name='nA', dimension='current', power=-9)),
 ('per_time': Dimension(name='per_time', t=-1)),
 ('resistance': Dimension(name='resistance', i=-2, m=1, t=-3, l=2)),
 ('time': Dimension(name='time', t=1)),
 ('voltage': Dimension(name='voltage', i=-1, m=1, t=-3, l=2))]
```

Again we get a `Document` instance object containing all the NineML objects in the document. An alternative representation can be obtained by reading the file as a `Network` object:

```
>>> from nineml.user import Network
>>> net = doc.read("./network/Brunel2000/AI.xml").as_network('BrunelAI')
>>> print(net)
Network(name='BrunelAI')
```

This gives a much more structured representation. For example, all the `Populations` within the model are available through the `populations` attribute:

```
>>> pprint(list(net.populations))
[Population(name='Exc', number=4000, cell=nrn),
 Population(name='Ext', number=5000, cell=stim),
 Population(name='Inh', number=1000, cell=nrn)]
```

1.3.2 Introspecting NineML models

Introspecting abstraction layer models

Once we have loaded a model from an XML file we can begin to examine its structure.

```
>>> model = doc['BrunelIaF']
>>> model
Dynamics(name='BrunelIaF')
```

We can see a list of model parameters:

```
>>> pprint(list(model.parameters))
[Parameter(theta, dimension=voltage),
 Parameter(Vreset, dimension=voltage),
 Parameter(R, dimension=resistance),
 Parameter(tau_rp, dimension=time),
 Parameter(tau, dimension=time)]
```

a list of state variables:

```
>>> pprint(list(model.state_variables))
[StateVariable(V, dimension=voltage),
 StateVariable(t_rpend, dimension=time)]
```

and a list of the variables that are imported from/exposed to the outside world:

```
>>> pprint(list(model.ports))
[AnalogSendPort('V', dimension='Dimension(name='voltage', i=-1, m=1, t=-3, l=2)'),
 AnalogSendPort('t_rpend', dimension='Dimension(name='time', t=1)'),
 AnalogReducePort('Isyn', dimension='Dimension(name='current', i=1)', op='+'),
 EventSendPort('spikeOutput')]
```

Delving more deeply, we can examine the model's regimes more closely:

```
>>> pprint(list(model.regimes))
[Regime(refractoryRegime),
 Regime(subthresholdRegime)]
>>> r_ref, r_sth = model.regimes
```

Looking first at the subthreshold regime, we can see the differential equations:

```
>>> list(r_sth.time_derivatives)
[TimeDerivative( dV/dt = (-V + R*Isyn)/tau )]
```

and the conditions under which the model will transition to the refractory regime:

```
>>> list(r_sth.transitions)
[OnCondition( V > theta )]
>>> tr_spike = next(r_sth.transitions)
```

The trigger for this transition is for the variable `V` to pass a threshold (parameter `theta`):

```
>>> tr_spike.trigger
Trigger('V > theta')
```

When the transition is initiated, the model will emit an output event (i.e. a spike) and discontinusouly change the values of some of the state variables:

```
>>> tr_spike.output_events
[OutputEvent('spikeOutput')]
>>> tr_spike.state_assignments
[StateAssignment('t_rpend', 't + tau_rp'), StateAssignment('V', 'Vreset')]
```

Then it will move to the refractory regime:

```
>>> tr_spike.target_regime
Regime(refractoryRegime)
```

The refractory regime can be introspected in a similar way.

Introspecting user layer models

As shown above, once a complete network model has been loaded as a `Network` object, we can look at its neuron populations and the connections between these populations ("projections"):

```
>>> pprint(list(net.populations))
[Population(name='Exc', number=4000, cell=nrn),
 Population(name='Ext', number=5000, cell=stim),
 Population(name='Inh', number=1000, cell=nrn)]

>>> pprint(list(net.projections))
[Projection(name="Inhibition", pre=Population(name='Inh', size=2500, cell=nrn),␣
↪post=Selection(name='All', Concatenate(Item(name='0'), Item(name='1'))),␣
↪connectivity=Connectivity(rule=RandomFanIn, src_size=2500, dest_size=12500),␣
↪response=DynamicsProperties(name="syn", component_class="Alpha
↪")plasticity=DynamicsProperties(name="InhibitoryPlasticity", component_class="Static
↪"), delay=1.5 * ms,event_port_connections=[EventPortConnection(sender=role:pre->
↪spike_output, receiver=role:response->input_spike)], analog_port_
↪connections=[AnalogPortConnection(sender=role:response->i_synaptic,␣
↪receiver=role:post->i_synaptic), AnalogPortConnection(sender=role:plasticity->fixed_
↪weight, receiver=role:response->weight)]),
 Projection(name="External", pre=Population(name='Ext', size=12500, cell=stim),␣
↪post=Selection(name='All', Concatenate(Item(name='0'), Item(name='1'))),␣
↪connectivity=Connectivity(rule=OneToOne, src_size=12500, dest_size=12500),␣
↪response=DynamicsProperties(name="syn", component_class="Alpha
↪")plasticity=DynamicsProperties(name="ExternalPlasticity", component_class="Static
↪"), delay=1.5 * ms,event_port_connections=[EventPortConnection(sender=role:pre->
↪spike_output, receiver=role:response->input_spike)], analog_port_
↪connections=[AnalogPortConnection(sender=role:response->i_synaptic,␣
↪receiver=role:post->i_synaptic), AnalogPortConnection(sender=role:plasticity->fixed_
↪weight, receiver=role:response->weight)]),
 Projection(name="Excitation", pre=Population(name='Exc', size=10000, cell=nrn),␣
↪post=Selection(name='All', Concatenate(Item(name='0'), Item(name='1'))),␣
↪connectivity=Connectivity(rule=RandomFanIn, src_size=10000, dest_size=12500),␣
↪response=DynamicsProperties(name="syn", component_class="Alpha
↪")plasticity=DynamicsProperties(name="ExcitatoryPlasticity", component_class="Static
↪"), delay=1.5 * ms,event_port_connections=[EventPortConnection(sender=role:pre->
↪spike_output, receiver=role:response->input_spike)], analog_port_
↪connections=[AnalogPortConnection(sender=role:response->i_synaptic,␣
↪receiver=role:post->i_synaptic), AnalogPortConnection(sender=role:plasticity->fixed_
```

NineML also supports "selections", groupings of neurons which span populations:

```
>>> pprint(list(net.selections))
[Selection(name='All', Concatenate(Item(name='0'), Item(name='1')))]
```

Note: in NineML version 1, the only type of selection is a concatenation of two or more populations. In future versions it will be possible to select and combine sub-populations.

Looking more closely at a population, we can see its name, the number of neurons it contains and the neuron model used (`Component`):

```
>>> p_exc = net.population('Exc')
>>> p_exc
Population(name='Exc', size=4000, cell=nrn)
>>> p_exc.size
4000
>>> p_exc.cell
DynamicsProperties(name="nrn", componentclass="BrunelIaF")
```

In the neuron model component we can see its abstraction layer definition (`ComponentClass`), it's properties (parameter values), and the initial values of its state variables.

Note: the handling of initial values is likely to change in future versions of NineML.

```
>>> p_exc.cell.component_class
Dynamics(name='BrunelIaF')
>>> pprint(list(p_exc.cell.properties))
[Property(name=Vreset, value=10.0, unit=mV),
 Property(name=tau, value=20.0, unit=ms),
 Property(name=R, value=1.5, unit=Mohm),
 Property(name=tau_rp, value=2.0, unit=ms),
 Property(name=theta, value=20.0, unit=mV)]
>>> pprint(list(p_exc.cell.initial_values))
[Initial(name='t_rpend', value=0.0, unit=ms),
 Initial(name='V', value=0.0, unit=mV)]
```

Turning from a population to a projection:

```
>>> prj_inh = net.projection('Inhibition')
>>> prj_inh.pre
Population(name='Inh', number=1000, cell=nrn)
>>> prj_inh.post
Selection(name='All', Concatenate(Item(name='0'), Item(name='1')))
>>> prj_inh.response
DynamicsProperties(name="syn", componentclass="AlphaPSR")
>>> prj_inh.connectivity
DynamicsProperties(name="RandomInh", componentclass="RandomFanIn")
>>> prj_inh.plasticity
DynamicsProperties(name="InhibitoryPlasticity", componentclass="StaticConnection")
>>> prj_inh.delay
1.5 * ms
>>> pprint(list(prj_inh.port_connections))
[AnalogPortConnection(sender=role:response->i_synaptic, receiver=role:post->i_
↪synaptic),
AnalogPortConnection(sender=role:plasticity->fixed_weight, receiver=role:response->
↪weight),
EventPortConnection(sender=role:pre->spike_output, receiver=role:response->input_
↪spike)]
```

Note that the `pre` and `post` attributes point to `Populations` or `Projections`, the `connectivity` rule, the post-synaptic `response` model and the synaptic `plasticity` model are all `Components`. The `port_connections` attribute indicates which ports in the different components should be connected together.

1.3.3 Writing model descriptions in Python

Writing abstraction layer models

```
subthreshold_regime = Regime(
    name="subthreshold_regime",
    time_derivatives=[
        "dV/dt = alpha*V*V + beta*V + zeta - U + Isyn / C_m",
        "dU/dt = a*(b*V - U)", ],

    transitions=[On("V > theta",
                    do=["V = c",
                        "U =  U+ d",
                        OutputEvent('spike')],
                    to='subthreshold_regime')]
)

ports = [AnalogSendPort("V", un.voltage),
         AnalogReducePort("Isyn", un.current, operator="+")]

parameters = [
    Parameter('theta', un.voltage),
    Parameter('a', un.per_time),
    Parameter('b', un.per_time),
    Parameter('c', un.voltage),
    Parameter('d', un.voltage / un.time),
    Parameter('C_m', un.capacitance),
    Parameter('alpha', un.dimensionless / (un.voltage * un.time)),
    Parameter('beta', un.per_time),
    Parameter('zeta', un.voltage / un.time)]

state_variables = [
    StateVariable('V', un.voltage),
    StateVariable('U', un.voltage / un.time)]

izhi = Dynamics(
    name="Izhikevich",
    parameters=parameters,
    state_variables=state_variables,
    regimes=[subthreshold_regime],
    analog_ports=ports)
```

Writing user layer models

```
# Meta-parameters
order = 1000          # scales the size of the network
Ne = 4 * order        # number of excitatory neurons
Ni = 1 * order        # number of inhibitory neurons
epsilon = 0.1         # connection probability
Ce = int(epsilon * Ne)   # number of excitatory synapses per neuron
Ci = int(epsilon * Ni)   # number of inhibitory synapses per neuron
Cext = Ce             # effective number of external synapses per neuron
delay = 1.5           # (ms) global delay for all neurons in the group
J = 0.1               # (mV) EPSP size
Jeff = 24.0 * J         # (nA) synaptic weight
Je = Jeff             # excitatory weights
Ji = -g * Je          # inhibitory weights
Jext = Je             # external weights
theta = 20.0          # firing thresholds
tau = 20.0            # membrane time constant
tau_syn = 0.1         # synapse time constant
# nu_thresh = theta / (Je * Ce * tau * exp(1.0) * tau_syn) # threshold rate
nu_thresh = theta / (J * Ce * tau)
nu_ext = eta * nu_thresh      # external rate per synapse
input_rate = 1000.0 * nu_ext * Cext    # mean input spiking rate

# Parameters
neuron_parameters = dict(tau=tau * ms,
                         v_threshold=theta * mV,
                         refractory_period=2.0 * ms,
                         v_reset=10.0 * mV,
                         R=1.5 * Mohm)   # units??
psr_parameters = dict(tau=tau_syn * ms)

# Initial Values
v_init = RandomDistributionProperties(
    "uniform_rest_to_threshold",
    ninemlcatalog.load("randomdistribution/Uniform",
                       'UniformDistribution'),
    {'minimum': (0.0, unitless),
     'maximum': (theta, unitless)})
neuron_initial_values = {"v": (v_init * mV),
                         "refractory_end": (0.0 * ms)}
synapse_initial_values = {"a": (0.0 * nA), "b": (0.0 * nA)}
tpoisson_init = RandomDistributionProperties(
    "exponential_beta",
    ninemlcatalog.load('randomdistribution/Exponential',
                       'ExponentialDistribution'),
    {"rate": (1000.0 / input_rate * unitless)})

# Dynamics components
celltype = DynamicsProperties(
    "nrn",
    ninemlcatalog.load('neuron/LeakyIntegrateAndFire',
                       'LeakyIntegrateAndFire'),
    neuron_parameters, initial_values=neuron_initial_values)
ext_stim = DynamicsProperties(
    "stim",
    ninemlcatalog.load('input/Poisson', 'Poisson'),
    dict(rate=(input_rate, Hz)),
    initial_values={"t_next": (tpoisson_init, ms)})
psr = DynamicsProperties(
    "syn",
    ninemlcatalog.load('postsynapticresponse/Alpha', 'Alpha'),
    psr_parameters,
    initial_values=synapse_initial_values)

# Connecion rules
one_to_one_class = ninemlcatalog.load(
    '/connectionrule/OneToOne', 'OneToOne')
random_fan_in_class = ninemlcatalog.load(
    '/connectionrule/RandomFanIn', 'RandomFanIn')

# Populations
exc_cells = Population("Exc", Ne, celltype, positions=None)
inh_cells = Population("Inh", Ni, celltype, positions=None)
external = Population("Ext", Ne + Ni, ext_stim, positions=None)

# Selections
all_cells = Selection(
    "All", Concatenate(exc_cells, inh_cells))

# Projections
input_prj = Projection(
    "External", external, all_cells,
    connectivity=ConnectionRuleProperties(
        "OneToOne", one_to_one_class),
    response=psr,
    plasticity=DynamicsProperties(
        "ExternalPlasticity",
        ninemlcatalog.load("plasticity/Static", 'Static'),
        properties={"weight": (Jext, nA)}),
    port_connections=[
        EventPortConnection(
            'pre', 'response', 'spike_output', 'spike'),
        AnalogPortConnection(
            "plasticity", "response", "fixed_weight", "weight"),
        AnalogPortConnection(
            "response", "destination", "i_synaptic", "i_synaptic")],
    delay=(delay, ms))

exc_prj = Projection(
    "Excitation", exc_cells, all_cells,
    connectivity=ConnectionRuleProperties(
        "RandomExc", random_fan_in_class, {"number": (Ce * unitless)}),
    response=psr,
    plasticity=DynamicsProperties(
        "ExcitatoryPlasticity",
        ninemlcatalog.load("plasticity/Static", 'Static'),
        properties={"weight": (Je, nA)}),
    port_connections=[
        EventPortConnection(
            'pre', 'response', 'spike_output', 'spike'),
        AnalogPortConnection(
            "plasticity", "response", "fixed_weight", "weight"),
        AnalogPortConnection(
            "response", "destination", "i_synaptic", "i_synaptic")],
    delay=(delay, ms))
```

```
inh_prj = Projection(
    "Inhibition", inh_cells, all_cells,
    connectivity=ConnectionRuleProperties(
        "RandomInh", random_fan_in_class, {"number": (Ci * unitless)}),
    response=psr,
    plasticity=DynamicsProperties(
        "InhibitoryPlasticity",
        ninemlcatalog.load("plasticity/Static", 'Static'),
        properties={"weight": (Ji, nA)}),
    port_connections=[
        EventPortConnection(
            'pre', 'response', 'spike_output', 'spike'),
        AnalogPortConnection(
            "plasticity", "response", "fixed_weight", "weight"),
        AnalogPortConnection(
            "response", "destination", "i_synaptic", "i_synaptic")],
    delay=(delay, ms))

# Save to document in NineML Catalog
network = Network(name if name else "BrunelNetwork")
network.add(exc_cells, inh_cells, external, all_cells, input_prj, exc_prj,
            inh_prj)
```

1.4 NineML Types

1.4.1 Relationship to specification

There is a near one-to-one mapping between NineML types as defined in the NineML specification and classes in the `nineml` Python package.

The most significant exceptions are classes in the `nineml` package that are modelled on proposed changes to the NineML specification (see http://github.com/INCF/nineml-spec/issues), e.g. ComponentClass->:ref:*Dynamics*/ConnectionRule, Projection, Quantity.

There are also cases where a type in the specification is just a thin wrapper around a body element (e.g. Delay, Size), which are "flattened" to be attributes in the NineML Python Library.

Mathematical expressions

All expressions in the NineML Python Library are represented using Sympy objects. Whereas in the NineML Specification mathematical expressions are specified to be enclosed within *MathInline* elements (with a subset of *MathML* planned as an alternative in future versions), in the NineML Python Library the Sympy object representing is accessed via the `rhs` property of the relevant objects.

1.4.2 Common properties/methods

All types

All NineML types in the NineML Python Library derive from `BaseNineMLObject`, which adds some common methods.

Document-level types

There are 12 types that are permitted in the root of a NineML document

- Dynamics
- DynamicsProperties
- ConnectionRule
- ConnectionRuleProperties
- RandomDistribution
- RandomDistribution
- Population
- Projection
- Selection
- Network
- Unit
- Dimension

Instances of these types has a `document` property to access the document it belongs to and a `url` property to access the url of the document. If the instance has not been added to a document then they will return `None`.

Container types

NineML types that can have multiple child elements of one or more types, i.e.:

- Dynamics
- ConnectionRule
- RandomDistribution
- DynamicsProperties
- ConnectionRuleProperties
- RandomDistributionProperties
- Regime
- OnEvent
- OnCondition
- Network
- Selection

derive from the `ContainerObject` class, which defines several methods to accessing, adding and removing their children. Internally, each child is stored in a dictionary according to its type. However, access to children is provided through four standardised accessor methods for each child type the container can hold:

`<child-type-plural>`: Property that returns an iterator over child elements of the given type (e.g. `aliases`, `parameters`, `on_conditions`)

<child-type>_names/keys: Property that returns an iterator over the keys of child elements that are used to store the child in the internal dictionary. If the child type has a name, then the access will be `<child-type>_names`, otherwise it will be `<child-type>_keys` (e.g. `alias_names`, `parameter_names`, `on_condition_keys`)

num_<child-type-plural>: Property that returns the number child elements in the container

<child-type>: Accessor method that takes the name/key of the child type and returns the corresponding element in the container.

There are a number of standard methods for container types

Annotations

All NineML elements can be annotated (except Annotations themselves) via their `annotations` property. The `annotations` property returns an `Annotations` element, with several convenient methods for setting attributes of nested elements.

1.5 Serialization

All NineML Python objects can be written to file via their `write` method, which simply wraps the object in a Document and passes it to the `nineml.write` function (alternatively the `nineml.write` function can be called directly). NineML documents can be read from files into Document objects using the `nineml.read` method, e.g.:

```
>>> dynA =  nineml.Dynamics('A', ...)
>>> dynA.write('example.xml')  # Alternatively nineml.write('example.xml', dynA, ...)
>>> doc = nineml.read('example.xml')
>>> dynA = doc['dynA']
```

Documents that are read or written to/from files will be cached in the Document class unless the `register` keyword argument is set to `False`.

NineML objects can also be serialized to string and/or basic Python objects and back again using the `serialize` and `unserialize` methods depending on the data format chosen (see *Formats*).

1.5.1 Formats

There are currently five supported formats for serialization with the NineML Python library: XML, YAML, JSON, HDF5, and Python dictionary (the JSON and YAML formats are derived from the Python dictionary serializer). Noting that the serialization module is written in a modular way that can support additional hierarchical formats if required by deriving the `BaseSerializer` and `BaseUnserializer` classes.

Depending on the format used, NineML can be serialized to file, string or standard Python objects (i.e dictionary).

Format	File	String	Object
XML	X	X	X
JSON	X	X	
YAML	X	X	
HDF5	X		
Python dictionary			X

Note: Although the set of hierarchical object models that can be represented by XML, JSON/YAML and HDF5 are very similar, there are slight differences that prevent general one-to-one mappings between them. These issues, and how they are overcome are explained in the Serialization Section of the NineML Specification.

1.5.2 Versions

The NineML Python Library is fully interoperable with the NineML v1 syntax the v2 syntax currently under development. While this will not be feasible as non-compatible features are added to v2, the aim is to maintain full backwards compatibility with v1.

1.5.3 Referencing style

References from one serialized NineML object to another can either be "local", where both objects are contained in the same document, or "remote", where the referenced object is in a different document to the object that references it.

Remote references enable large and complex models to be split across a number of files, or to reference standardized models from the NineML catalog for example. However, in some circumstances it may be desirable to copy all references to the local document, for ease-of-portability or to reduce the complexity of the read methods required by supporting tools.

The `ref_style` keyword argument can be used to control the referencing style used when serializing NineML documents. Valid options are

local: All references are written locally to the serialized document.

prefer: Objects are written as references where possible

inline: Objects are written inline where possible

None: Whether an object is written as a reference or inline is preserved from when the document was read

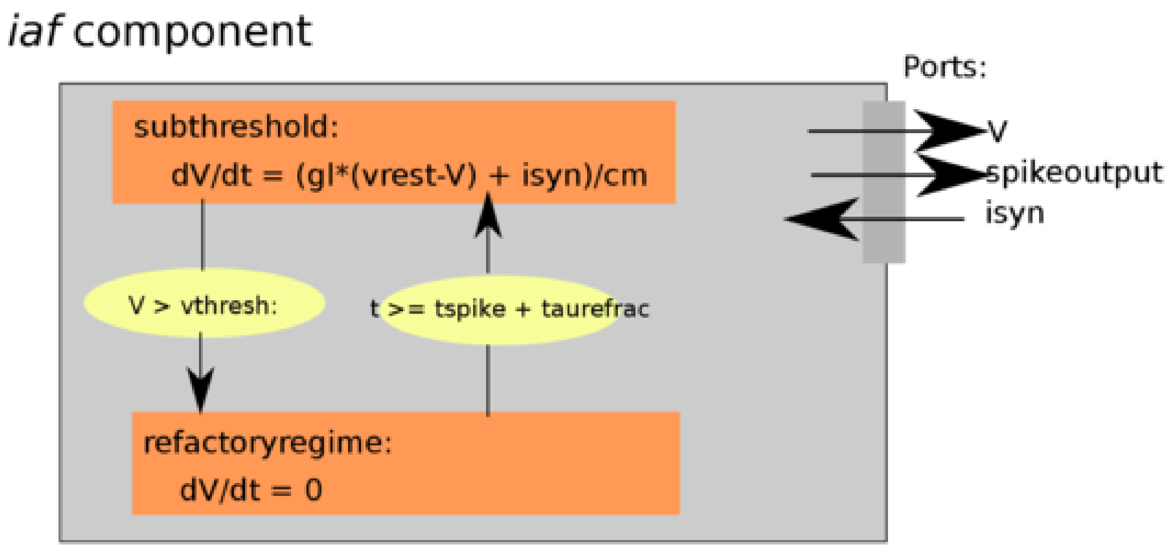
1.6 Hierarchical dynamics

Hierarchical components allow us to build a single component, out of several smaller components. For example, imagine we could build a component that represented an integrate-and-fire neuron (IAF) with 2 input synapses. We could do this by either by creating a single component, as we have been doing previously, or by creating 3 components; the IAF component and 2 synapses, and then creating a larger component out of them by specifying internal connectivity.

Building larger components out of smaller components has several advantages:

- **We can define components in a reusable way. I.e., we can write the IAF** subcomponent once, then reuse it across multiple components.
- **We can isolated unrelated variables; reducing the chance of a typo** producing a bug or variable collisions.

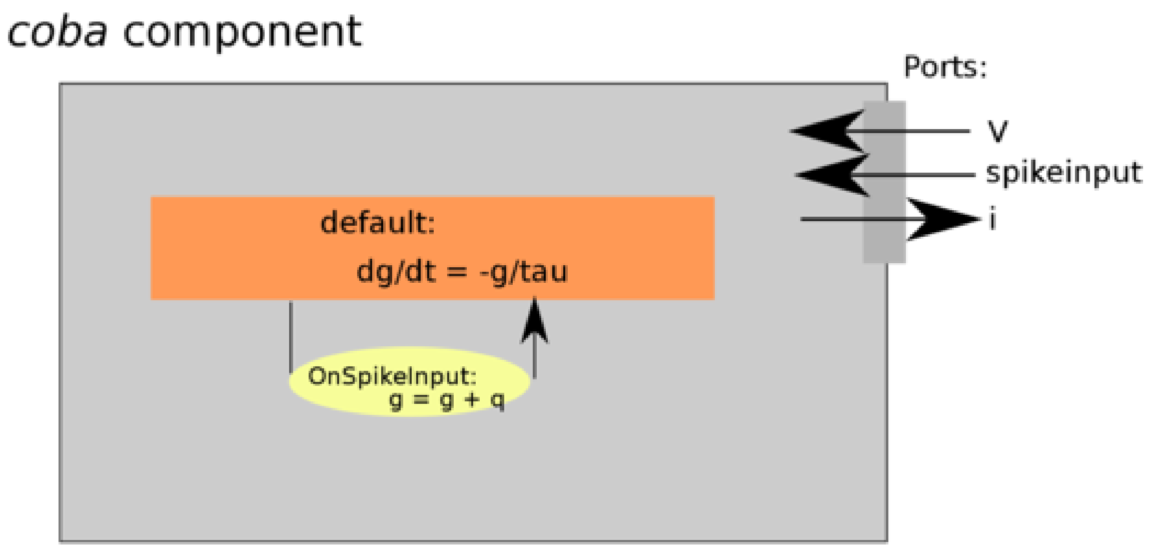
We look at the IAF with two synapse example in more detail. The following figure shows a cartoon of an iaf neuron with a refractory period. Orange boxes denote regimes, yellow ovals denote transitions and the ports are shown on the right-hand-side. Parameters have been omitted.



The corresponding code to generate this component is:

```
r1 = al.Regime(name = "subthresholdregime",
    time_derivatives = ["dV/dt = ( gl*( vrest - V ) + ISyn)/(cm)"],
    transitions = [al.On("V > vthresh",
                         do=["tspike = t",
                             "V = vreset",
                             al.OutputEvent('spikeoutput')],
                         to="refractoryregime")])
r2 = al.Regime(name="refractoryregime",
               time_derivatives=["dV/dt = 0"],
iaf = al.Dynamics(
    name = "iaf",
    dynamics = al.Dynamics( regimes = [r1,r2] ),
    analog_ports=[al.SendPort("V"), al.ReducePort("ISyn", reduce_op="+")],
    event_ports=[al.SendEventPort('spikeoutput')])
```

Similarly, we can define a synapse component:



with corresponding code:

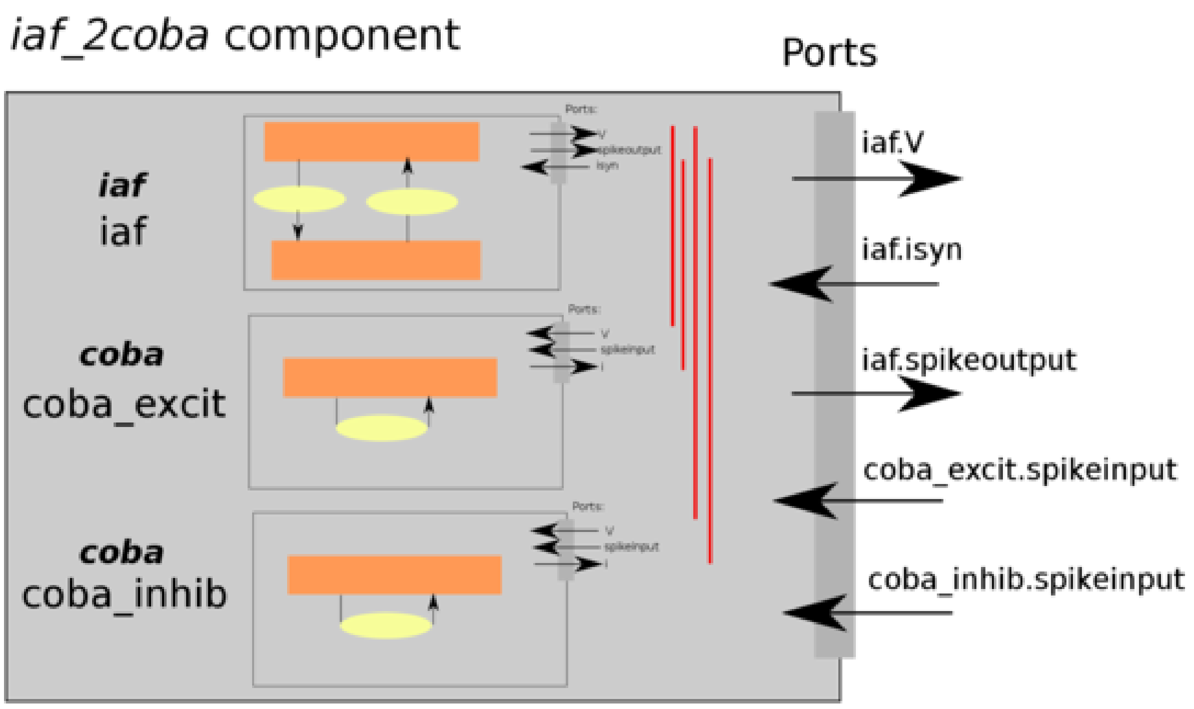
```
coba = al.Dynamics(
     name = "CobaSyn",
     dynamics =
        al.Dynamics(
            aliases = ["I:=g*(vrev-V)", ],
            regimes = [
              al.Regime(
                 name = "cobadefaultregime",
                 time_derivatives = ["dg/dt = -g/tau",],
                 transitions = [
                     al.On(al.InputEvent('spikeinput'), do=["g=g+q"]),
                     ],
                 )
                ],
            state_variables = [ al.StateVariable('g') ]
            ),

     analog_ports = [ al.RecvPort("V"), al.SendPort("I"), ],
     event_ports = [al.RecvEventPort('spikeinput') ],
     parameters = [ al.Parameter(p) for p in ['tau','q','vrev']  ]
     )
```

1.6.1 Multi-Dynamics

We now define a larger component, which will contain these sub_dynamics. When we create the component, we specify the name of each subcomponent, which allows us to reference them in the future.

We also need to specify that the voltage send port from the iaf needs to be connected to the voltage receive ports of the synapse. Similarly we need to connect the current port from the synapses into the current reduce port on the IAF neuron. These connections are shown in red on the diagram, and correspond to the arguments corresponding to the *port_connections* argument.

In a diagram:



In code:

```
# Create a model, composed of an iaf neuron, and
iaf_2coba_comp = al.MultiDynamics(name="iaf_2coba",
                                  sub_dynamics={"iaf" : get_iaf(),
                                                "coba_excit" : get_coba(),
                                                "coba_inhib" : get_coba()},
                                  port_connections=[
                                      ("iaf", "V", "coba_excit", "V"),
                                      ("iaf", "V", "coba_inhib", "V"),
                                      ("coba_excit", "I", "iaf", "ISyn"),
                                      ("coba_inhib", "I", "iaf", "ISyn")]
```

1.7 Examples

1.7.1 Neuron Models

Example - Adaptive Exponential Integrate and Fire

```
from __future__ import division
from nineml import units as un
from nineml import abstraction as al, user as ul
```

```python
def create_adaptive_exponential():
    """
    Adaptive exponential integrate-and-fire neuron as described in
    A. Destexhe, J COmput Neurosci 27: 493--506 (2009)

    Author B. Kriener (Jan 2011)

    ## neuron model: aeIF

    ## variables:
    ## V: membrane potential
    ## w: adaptation variable

    ## parameters:
    ## C_m      # specific membrane capacitance [muF/cm**2]
    ## g_L      # leak conductance [mS/cm**2]
    ## E_L      # resting potential [mV]
    ## Delta    # steepness of exponential approach to threshold [mV]
    ## V_T      # spike threshold [mV]
    ## S        # membrane area [mum**2]
    ## trefractory # refractory time [ms]
    ## tspike   # spike time [ms]
    ## tau_w    # adaptation time constant
    ## a, b     # adaptation parameters [muS, nA]
    """
    aeIF = al.Dynamics(
        name="AdaptiveExpIntegrateAndFire",
        parameters=[
            al.Parameter('C_m', un.capacitance),
            al.Parameter('g_L', un.conductance),
            al.Parameter('E_L', un.voltage),
            al.Parameter('Delta', un.voltage),
            al.Parameter('V_T', un.voltage),
            al.Parameter('S'),
            al.Parameter('trefractory', un.time),
            al.Parameter('tspike', un.time),
            al.Parameter('tau_w', un.time),
            al.Parameter('a', un.dimensionless / un.voltage),
            al.Parameter('b')],
        state_variables=[
            al.StateVariable('V', un.voltage),
            al.StateVariable('w')],
        regimes=[
            al.Regime(
                name="subthresholdregime",
                time_derivatives=[
                    "dV/dt = -g_L*(V-E_L)/C_m + Isyn/C_m + g_L*Delta*exp((V-V_T)/
↪Delta-w/S)/C_m",   # @IgnorePep8
                    "dw/dt = (a*(V-E_L)-w)/tau_w", ],
                transitions=al.On("V > V_T",
                                  do=["V = E_L", "w = w + b",
                                      al.OutputEvent('spikeoutput')],
                                  to="refractoryregime")),
            al.Regime(
                name="refractoryregime",
                transitions=al.On("t>=tspike+trefractory",
                                  to="subthresholdregime"))],
        analog_ports=[al.AnalogReducePort("Isyn", un.current, operator="+")])
    return aeIF


def parameterise_adaptive_exponential(definition=None):
    if definition is None:
        definition = create_adaptive_exponential()
    comp = ul.DynamicsProperties(
        name='SampleAdaptiveExpIntegrateAndFire',
        definition=definition,
        properties=[ul.Property('C_m', 1 * un.pF),
                    ul.Property('g_L', 0.1 * un.nS),
                    ul.Property('E_L', -65 * un.mV),
                    ul.Property('Delta', 1 * un.mV),
                    ul.Property('V_T', -58 * un.mV),
                    ul.Property('S', 0.1),
                    ul.Property('tspike', 0.5 * un.ms),
                    ul.Property('trefractory', 0.25 * un.ms),
                    ul.Property('tau_w', 4 * un.ms),
                    ul.Property('a', 1 * un.per_mV),
                    ul.Property('b', 2)],
        initial_values=[ul.Initial('V', -70 * un.mV),
                        ul.Initial('w', 0.1 * un.mV)])
    return comp
```

Example - Hodgkin-Huxley

```
from __future__ import division
from past.utils import old_div
from nineml import abstraction as al, user as ul, Document
from nineml import units as un
from nineml.xml import E, etree


def create_hodgkin_huxley():
    """A Hodgkin-Huxley single neuron model.
    Written by Andrew Davison.
    See http://phobos.incf.ki.se/src_rst/
              examples/examples_al_python.html#example-hh
    """
    aliases = [
        "q10 := 3.0**((celsius - qfactor)/tendegrees)",  # temperature correction␣
↪factor @IgnorePep8
        "m_alpha := m_alpha_A*(V-m_alpha_V0)/(exp(-(V-m_alpha_V0)/m_alpha_K) - 1.0)",␣
↪ # @IgnorePep8
        "m_beta := m_beta_A*exp(-(V-m_beta_V0)/m_beta_K)",
        "mtau := 1.0/(q10*(m_alpha + m_beta))",
        "minf := m_alpha/(m_alpha + m_beta)",
        "h_alpha := h_alpha_A*exp(-(V-h_alpha_V0)/h_alpha_K)",
        "h_beta := h_beta_A/(exp(-(V-h_beta_V0)/h_beta_K) + 1.0)",
        "htau := 1.0/(q10*(h_alpha + h_beta))",
        "hinf := h_alpha/(h_alpha + h_beta)",
        "n_alpha := n_alpha_A*(V-n_alpha_V0)/(exp(-(V-n_alpha_V0)/n_alpha_K) - 1.0)",␣
↪ # @IgnorePep8
        "n_beta := n_beta_A*exp(-(V-n_beta_V0)/n_beta_K)",
        "ntau := 1.0/(q10*(n_alpha + n_beta))",
        "ninf := n_alpha/(n_alpha + n_beta)",
        "gna := gnabar*m*m*m*h",
        "gk := gkbar*n*n*n*n",
        "ina := gna*(ena - V)",
        "ik := gk*(ek - V)",
        "il := gl*(el - V )"]

    hh_regime = al.Regime(
        "dn/dt = (ninf-n)/ntau",
        "dm/dt = (minf-m)/mtau",
        "dh/dt = (hinf-h)/htau",
        "dV/dt = (ina + ik + il + isyn)/C",
        transitions=al.On("V > v_threshold", do=al.SpikeOutputEvent())
    )

    state_variables = [
        al.StateVariable('V', un.voltage),
        al.StateVariable('m', un.dimensionless),
        al.StateVariable('n', un.dimensionless),
        al.StateVariable('h', un.dimensionless)]

    # the rest are not "parameters" but aliases, assigned vars, state vars,
    # indep vars, analog_analog_ports, etc.
    parameters = [
        al.Parameter('el', un.voltage),
        al.Parameter('C', un.capacitance),
        al.Parameter('ek', un.voltage),
        al.Parameter('ena', un.voltage),
        al.Parameter('gkbar', un.conductance),
        al.Parameter('gnabar', un.conductance),
        al.Parameter('v_threshold', un.voltage),
        al.Parameter('gl', un.conductance),
        al.Parameter('celsius', un.temperature),
        al.Parameter('qfactor', un.temperature),
        al.Parameter('tendegrees', un.temperature),
        al.Parameter('m_alpha_A', old_div(un.dimensionless, (un.time * un.voltage))),
        al.Parameter('m_alpha_V0', un.voltage),
        al.Parameter('m_alpha_K', un.voltage),
        al.Parameter('m_beta_A', old_div(un.dimensionless, un.time)),
        al.Parameter('m_beta_V0', un.voltage),
        al.Parameter('m_beta_K', un.voltage),
        al.Parameter('h_alpha_A', old_div(un.dimensionless, un.time)),
        al.Parameter('h_alpha_V0', un.voltage),
        al.Parameter('h_alpha_K', un.voltage),
        al.Parameter('h_beta_A', old_div(un.dimensionless, un.time)),
        al.Parameter('h_beta_V0', un.voltage),
        al.Parameter('h_beta_K', un.voltage),
        al.Parameter('n_alpha_A', old_div(un.dimensionless, (un.time * un.voltage))),
        al.Parameter('n_alpha_V0', un.voltage),
        al.Parameter('n_alpha_K', un.voltage),
        al.Parameter('n_beta_A', old_div(un.dimensionless, un.time)),
        al.Parameter('n_beta_V0', un.voltage),
        al.Parameter('n_beta_K', un.voltage)]

    analog_ports = [al.AnalogSendPort("V", un.voltage),
                    al.AnalogReducePort("isyn", un.current, operator="+")]

    dyn = al.Dynamics("HodgkinHuxley",
                      parameters=parameters,
                      state_variables=state_variables,
                      regimes=(hh_regime,),
                      aliases=aliases,
                      analog_ports=analog_ports)
    return dyn


def parameterise_hodgkin_huxley(definition=None):
    if definition is None:
        definition = create_hodgkin_huxley()
    comp = ul.DynamicsProperties(
        name='SampleHodgkinHuxley',
        definition=create_hodgkin_huxley(),
        properties=[ul.Property('C', 1.0 * un.pF),
                    ul.Property('celsius', 20.0 * un.degC),
                    ul.Property('ek', -90 * un.mV),
                    ul.Property('el', -65 * un.mV),
                    ul.Property('ena', 80 * un.mV),
                    ul.Property('gkbar', 30.0 * un.nS),
                    ul.Property('gl', 0.3 * un.nS),
                    ul.Property('gnabar', 130.0 * un.nS),
                    ul.Property('v_threshold', -40.0 * un.mV),
                    ul.Property('qfactor', 6.3 * un.degC),
                    ul.Property('tendegrees', 10.0 * un.degC),
                    ul.Property('m_alpha_A', -0.1,
                                old_div(un.unitless, (un.ms * un.mV))),
                    ul.Property('m_alpha_V0', -40.0 * un.mV),
                    ul.Property('m_alpha_K', 10.0 * un.mV),
                    ul.Property('m_beta_A', 4.0 * un.per_ms),
                    ul.Property('m_beta_V0', -65.0 * un.mV),
                    ul.Property('m_beta_K', 18.0 * un.mV),
                    ul.Property('h_alpha_A', 0.07 * un.per_ms),
                    ul.Property('h_alpha_V0', -65.0 * un.mV),
                    ul.Property('h_alpha_K', 20.0 * un.mV),
                    ul.Property('h_beta_A', 1.0 * un.per_ms),
                    ul.Property('h_beta_V0', -35.0 * un.mV),
                    ul.Property('h_beta_K', 10.0 * un.mV),
                    ul.Property('n_alpha_A', -0.01,
                                old_div(un.unitless, (un.ms * un.mV))),
                    ul.Property('n_alpha_V0', -55.0 * un.mV),
                    ul.Property('n_alpha_K', 10.0 * un.mV),
                    ul.Property('n_beta_A', 0.125 * un.per_ms),
                    ul.Property('n_beta_V0', -65.0 * un.mV),
                    ul.Property('n_beta_K', 80.0 * un.mV)],
        initial_values=[ul.Initial('V', -70 * un.mV),
                        ul.Initial('m', 0.1),
                        ul.Initial('n', 0),
                        ul.Initial('h', 0.9)])
    return comp
```

Example - Leaky Integrate and Fire

Example - Izhikevich

```
from __future__ import division
from past.utils import old_div
from nineml import units as un
from nineml import abstraction as al, user as ul, Document
from nineml.xml import etree, E


def create_izhikevich():
    subthreshold_regime = al.Regime(
        name="subthreshold_regime",
        time_derivatives=[
            "dV/dt = alpha*V*V + beta*V + zeta - U + Isyn / C_m",
            "dU/dt = a*(b*V - U)", ],

        transitions=[al.On("V > theta",
                           do=["V = c",
                               "U =  U+ d",
                               al.OutputEvent('spike')],
                           to='subthreshold_regime')]
    )

    ports = [al.AnalogSendPort("V", un.voltage),
             al.AnalogReducePort("Isyn", un.current, operator="+")]

    parameters = [
        al.Parameter('theta', un.voltage),
        al.Parameter('a', un.per_time),
        al.Parameter('b', un.per_time),
        al.Parameter('c', un.voltage),
        al.Parameter('d', old_div(un.voltage, un.time)),
        al.Parameter('C_m', un.capacitance),
        al.Parameter('alpha', old_div(un.dimensionless, (un.voltage * un.time))),
        al.Parameter('beta', un.per_time),
        al.Parameter('zeta', old_div(un.voltage, un.time))]

    state_variables = [
        al.StateVariable('V', un.voltage),
        al.StateVariable('U', old_div(un.voltage, un.time))]

    c1 = al.Dynamics(
        name="Izhikevich",
        parameters=parameters,
        state_variables=state_variables,
        regimes=[subthreshold_regime],
        analog_ports=ports

    )
    return c1


def create_izhikevich_fast_spiking():
    """
    Load Fast spiking Izhikevich XML definition from file and parse into
    Abstraction Layer of Python API.
    """
    izhi_fs = al.Dynamics(
        name='IzhikevichFastSpiking',
        parameters=[
            al.Parameter('a', un.per_time),
            al.Parameter('b', old_div(un.conductance, (un.voltage ** 2))),
            al.Parameter('c', un.voltage),
            al.Parameter('k', old_div(un.conductance, un.voltage)),
            al.Parameter('Vr', un.voltage),
            al.Parameter('Vt', un.voltage),
            al.Parameter('Vb', un.voltage),
            al.Parameter('Vpeak', un.voltage),
            al.Parameter('Cm', un.capacitance)],
        analog_ports=[
            al.AnalogReducePort('iSyn', un.current, operator="+"),
            al.AnalogSendPort('U', un.current),
            al.AnalogSendPort('V', un.voltage)],
        event_ports=[
            al.EventSendPort("spikeOutput")],
        state_variables=[
            al.StateVariable('V', un.voltage),
            al.StateVariable('U', un.current)],
        regimes=[
            al.Regime(
                'dU/dt = a * (b * pow(V - Vb, 3) - U)',
                'dV/dt = V_deriv',
                transitions=[
                    al.On('V > Vpeak',
                          do=['V = c', al.OutputEvent('spikeOutput')],
                          to='subthreshold')],
                name="subthreshold"),
            al.Regime(
                'dU/dt = - U * a',
                'dV/dt = V_deriv',
                transitions=[al.On('V > Vb', to="subthreshold")],
                name="subVb")],
        aliases=["V_deriv := (k * (V - Vr) * (V - Vt) - U + iSyn) / Cm"])  #␣
↪@IgnorePep8
    return izhi_fs


def parameterise_izhikevich(definition=None):
    if definition is None:
        definition = create_izhikevich()
    comp = ul.DynamicsProperties(
        name='SampleIzhikevich',
        definition=create_izhikevich(),
        properties=[ul.Property('a', 0.2 * un.per_ms),
                    ul.Property('b', 0.025 * un.per_ms),
                    ul.Property('c', -75 * un.mV),
                    ul.Property('d', 0.2 * un.mV / un.ms),
                    ul.Property('theta', -50 * un.mV),
                    ul.Property('alpha', 0.04 * un.unitless / (un.mV * un.ms)),
                    ul.Property('beta', 5 * un.per_ms),
                    ul.Property('zeta', 140.0 * un.mV / un.ms),
                    ul.Property('C_m', 1.0 * un.pF)],
        initial_values=[ul.Initial('V', -70 * un.mV),
                        ul.Initial('U', -1.625 * un.mV / un.ms)])
    return comp


def parameterise_izhikevich_fast_spiking(definition=None):
    if definition is None:
        definition = create_izhikevich_fast_spiking()
    comp = ul.DynamicsProperties(
        name='SampleIzhikevichFastSpiking',
        definition=create_izhikevich_fast_spiking(),
        properties=[ul.Property('a', 0.2 * un.per_ms),
                    ul.Property('b', 0.025 * un.nS / un.mV ** 2),
                    ul.Property('c', -45 * un.mV),
                    ul.Property('k', 1 * un.nS / un.mV),
                    ul.Property('Vpeak', 25 * un.mV),
                    ul.Property('Vb', -55 * un.mV),
                    ul.Property('Cm', 20 * un.pF),
                    ul.Property('Vr', -55 * un.mV),
                    ul.Property('Vt', -40 * un.mV)],
        initial_values=[ul.Initial('V', -70 * un.mV),
                        ul.Initial('U', -1.625 * un.mV / un.ms)])
    return comp
```

1.7.2 Post-synaptic Response Models

Example - Alpha

1.7.3 Plasticity Models

Example - Static

```
from __future__ import print_function
from nineml import units as un, user as ul, abstraction as al, Document
from nineml.xml import etree, E


def create_static():
    dyn = al.Dynamics(
        name="Static",
        aliases=["fixed_weight := weight"],
        regimes=[
            al.Regime(name="default")],
        analog_ports=[al.AnalogSendPort("fixed_weight", dimension=un.current)],
        parameters=[al.Parameter('weight', dimension=un.current)])
    return dyn


def parameterise_static():

    comp = ul.DynamicsProperties(
        name='SampleAlpha',
        definition=create_static(),
        properties=[ul.Property('weight', 10.0 * un.nA)])
    return comp


if __name__ == '__main__':
    import argparse
    try:
        import ninemlcatalog
        catalog_path = 'plasticity/Static'
    except ImportError:
        ninemlcatalog = None
    parser = argparse.ArgumentParser()
    parser.add_argument('--mode', type=str, default='print',
                        help=("The mode to run this script, can be 'print', "
                              "'compare' or 'save', which correspond to "
                              "printing the models, comparing the models with "
                              "the version in the catalog, or overwriting the "
                              "version in the catalog with this version "
                              "respectively"))
    args = parser.parse_args()

    if args.mode == 'print':
        document = Document()
        print(etree.tostring(
            E.NineML(
                create_static().to_xml(document),
                parameterise_static().to_xml(document)),
            encoding="UTF-8", pretty_print=True, xml_declaration=True))
    elif args.mode == 'compare':
        if ninemlcatalog is None:
            raise Exception(
                "NineML catalog is not installed")
        local_version = create_static()
        catalog_version = ninemlcatalog.load(catalog_path,
                                             local_version.name)
        mismatch = local_version.find_mismatch(catalog_version)
        if mismatch:
            print ("Local version differs from catalog version:\n{}"
                   .format(mismatch))
        else:
            print("Local version matches catalog version")
    elif args.mode == 'save':
        if ninemlcatalog is None:
            raise Exception(
                "NineML catalog is not installed")
        dynamics = create_static()
        ninemlcatalog.save(dynamics, catalog_path, dynamics.name)
        params = parameterise_static(
            ninemlcatalog.load(catalog_path, dynamics.name))
        ninemlcatalog.save(params, catalog_path, params.name)
        print("Saved '{}' and '{}' to catalog".format(dynamics.name,
                                                      params.name))
```

Example - Guetig Spike-timing Dependent Plasticity (STDP)

```
from nineml import units as un, user as ul, abstraction as al


def create_stdp_guetig():
    dyn = al.Dynamics(
        name="StdpGuetig",
        parameters=[
            al.Parameter(name='tauLTP', dimension=un.time),
            al.Parameter(name='aLTD', dimension=un.dimensionless),
            al.Parameter(name='wmax', dimension=un.dimensionless),
            al.Parameter(name='muLTP', dimension=un.dimensionless),
            al.Parameter(name='tauLTD', dimension=un.time),
            al.Parameter(name='aLTP', dimension=un.dimensionless)],
        analog_ports=[
            al.AnalogReceivePort(dimension=un.dimensionless, name="w"),
            al.AnalogSendPort(dimension=un.dimensionless, name="wsyn")],
        event_ports=[
            al.EventReceivePort(name="incoming_spike")],
        state_variables=[
            al.StateVariable(name='tlast_post', dimension=un.time),
            al.StateVariable(name='tlast_pre', dimension=un.time),
            al.StateVariable(name='deltaw', dimension=un.dimensionless),
            al.StateVariable(name='interval', dimension=un.time),
            al.StateVariable(name='M', dimension=un.dimensionless),
            al.StateVariable(name='P', dimension=un.dimensionless),
            al.StateVariable(name='wsyn', dimension=un.dimensionless)],
        regimes=[
            al.Regime(
                name="sole",
                al.On('incoming_spike',
                      target_regime="sole",
                      do=[
                          al.StateAssignment(
                              'tlast_post',
                              '((w >= 0) ? ( tlast_post ) : ( t ))'),
                          al.StateAssignment(
                              'tlast_pre',
                              '((w >= 0) ? ( t ) : ( tlast_pre ))'),
                          al.StateAssignment(
                              'deltaw',
                              '((w >= 0) ? '
                              '( 0.0 ) : '
                              '( P*pow(wmax - wsyn, muLTP) * '
                              'exp(-interval/tauLTP) + deltaw ))'),
                          al.StateAssignment(
                              'interval',
                              '((w >= 0) ? ( -t + tlast_post ) : '
                              '( t - tlast_pre ))'),
                          al.StateAssignment(
                              'M',
                              '((w >= 0) ? ( M ) : '
                              '( M*exp((-t + tlast_post)/tauLTD) - aLTD ))'),
                          al.StateAssignment(
                              'P',
                              '((w >= 0) ? '
                              '( P*exp((-t + tlast_pre)/tauLTP) + aLTP ) : '
                              '( P ))'),
                          al.StateAssignment(
                              'wsyn', '((w >= 0) ? ( deltaw + wsyn ) : '
                              '( wsyn ))')]))])
    return dyn


def parameterise_stdp_guetig():

    comp = ul.DynamicsProperties(
        name='SampleAlpha',
        definition=create_stdp_guetig(),
        properties=[])
    return comp
```

1.7.4 Network Models

Example - Brunel

```
# encoding: utf-8
"""
Network model from

    Brunel, N. (2000) J. Comput. Neurosci. 8: 183-208

expressed in NineML using the Python API

Author: Andrew P. Davison, UNIC, CNRS
June 2014
    Edited by Thomas G. Close, October 2015
"""

from __future__ import division
from nineml.user import (
    DynamicsProperties, Population, RandomDistributionProperties,
    Projection, ConnectionRuleProperties, AnalogPortConnection,
    EventPortConnection, Network, Selection, Concatenate)
from nineml.units import ms, mV, nA, unitless, Hz, Mohm
import ninemlcatalog


def create_brunel(g, eta, name=None):
    """
    Build a NineML representation of the Brunel (2000) network model.

    Arguments:
        g: relative strength of inhibitory synapses
        eta: nu_ext / nu_thresh

    Returns:
        a nineml user layer Model object
    """
    # Meta-parameters
    order = 1000       # scales the size of the network
    Ne = 4 * order     # number of excitatory neurons
    Ni = 1 * order     # number of inhibitory neurons
    epsilon = 0.1      # connection probability
    Ce = int(epsilon * Ne)  # number of excitatory synapses per neuron
    Ci = int(epsilon * Ni)  # number of inhibitory synapses per neuron
    Cext = Ce          # effective number of external synapses per neuron
    delay = 1.5        # (ms) global delay for all neurons in the group
    J = 0.1            # (mV) EPSP size
    Jeff = 24.0 * J      # (nA) synaptic weight
    Je = Jeff          # excitatory weights
    Ji = -g * Je       # inhibitory weights
    Jext = Je          # external weights
    theta = 20.0       # firing thresholds
    tau = 20.0         # membrane time constant
    tau_syn = 0.1      # synapse time constant
    # nu_thresh = theta / (Je * Ce * tau * exp(1.0) * tau_syn) # threshold rate
    nu_thresh = theta / (J * Ce * tau)
    nu_ext = eta * nu_thresh      # external rate per synapse
    input_rate = 1000.0 * nu_ext * Cext    # mean input spiking rate

    # Parameters
    neuron_parameters = dict(tau=tau * ms,
                             v_threshold=theta * mV,
                             refractory_period=2.0 * ms,
                             v_reset=10.0 * mV,
                             R=1.5 * Mohm)  # units??
    psr_parameters = dict(tau=tau_syn * ms)

    # Initial Values
    v_init = RandomDistributionProperties(
        "uniform_rest_to_threshold",
        ninemlcatalog.load("randomdistribution/Uniform",
                           'UniformDistribution'),
        {'minimum': (0.0, unitless),
         'maximum': (theta, unitless)})
#     v_init = 0.0
    neuron_initial_values = {"v": (v_init * mV),
                             "refractory_end": (0.0 * ms)}
    synapse_initial_values = {"a": (0.0 * nA), "b": (0.0 * nA)}
    tpoisson_init = RandomDistributionProperties(
        "exponential_beta",
        ninemlcatalog.load('randomdistribution/Exponential',
                           'ExponentialDistribution'),
        {"rate": (1000.0 / input_rate * unitless)})
#     tpoisson_init = 5.0

    # Dynamics components
    celltype = DynamicsProperties(
        "nrn",
        ninemlcatalog.load('neuron/LeakyIntegrateAndFire',
                           'LeakyIntegrateAndFire'),
        neuron_parameters, initial_values=neuron_initial_values)
    ext_stim = DynamicsProperties(
        "stim",
        ninemlcatalog.load('input/Poisson', 'Poisson'),
        dict(rate=(input_rate, Hz)),
        initial_values={"t_next": (tpoisson_init, ms)})
    psr = DynamicsProperties(
        "syn",
        ninemlcatalog.load('postsynapticresponse/Alpha', 'Alpha'),
        psr_parameters,
        initial_values=synapse_initial_values)

    # Connecion rules
    one_to_one_class = ninemlcatalog.load(
        '/connectionrule/OneToOne', 'OneToOne')
    random_fan_in_class = ninemlcatalog.load(
        '/connectionrule/RandomFanIn', 'RandomFanIn')

    # Populations
    exc_cells = Population("Exc", Ne, celltype, positions=None)
    inh_cells = Population("Inh", Ni, celltype, positions=None)
    external = Population("Ext", Ne + Ni, ext_stim, positions=None)
    # Selections
    all_cells = Selection(
        "All", Concatenate((exc_cells, inh_cells)))

    # Projections
    input_prj = Projection(
        "External", external, all_cells,
        connection_rule_properties=ConnectionRuleProperties(
            "OneToOne", one_to_one_class),
        response=psr,
        plasticity=DynamicsProperties(
            "ExternalPlasticity",
            ninemlcatalog.load("plasticity/Static", 'Static'),
            properties={"weight": (Jext, nA)}),
        port_connections=[
            EventPortConnection(
                'pre', 'response', 'spike_output', 'spike'),
            AnalogPortConnection(
                "plasticity", "response", "fixed_weight", "weight"),
            AnalogPortConnection(
                "response", "destination", "i_synaptic", "i_synaptic")],
        delay=(delay, ms))

    exc_prj = Projection(
        "Excitation", exc_cells, all_cells,
        connection_rule_properties=ConnectionRuleProperties(
            "RandomExc", random_fan_in_class, {"number": (Ce * unitless)}),
        response=psr,
        plasticity=DynamicsProperties(
            "ExcitatoryPlasticity",
            ninemlcatalog.load("plasticity/Static", 'Static'),
            properties={"weight": (Je, nA)}),
        port_connections=[
            EventPortConnection(
                'pre', 'response', 'spike_output', 'spike'),
            AnalogPortConnection(
                "plasticity", "response", "fixed_weight", "weight"),
            AnalogPortConnection(
                "response", "destination", "i_synaptic", "i_synaptic")],
        delay=(delay, ms))

    inh_prj = Projection(
        "Inhibition", inh_cells, all_cells,
        connection_rule_properties=ConnectionRuleProperties(
            "RandomInh", random_fan_in_class, {"number": (Ci * unitless)}),
        response=psr,
        plasticity=DynamicsProperties(
            "InhibitoryPlasticity",
            ninemlcatalog.load("plasticity/Static", 'Static'),
            properties={"weight": (Ji, nA)}),
        port_connections=[
            EventPortConnection(
                'pre', 'response', 'spike_output', 'spike'),
            AnalogPortConnection(
                "plasticity", "response", "fixed_weight", "weight"),
            AnalogPortConnection(
                "response", "destination", "i_synaptic", "i_synaptic")],
        delay=(delay, ms))

    # Save to document in NineML Catalog
    network = Network(name if name else "BrunelNetwork")
    network.add(exc_cells, inh_cells, external, all_cells, input_prj, exc_prj,
                inh_prj)
    return network
```

1.8 API reference

Following the layer structure of the NineML specification, the `nineml` package is split into a *Abstraction* and *User Layers*, with a small intersection that are common to both layers.

1.8.1 Common Types API

There a few NineML types are common across all layers

Document

Dimensions and units

A number of `Dimensions` and `Unit`have been pre-defined, in the `nineml.units` module, for example:

```
>>> from nineml.units import time, voltage, capacitance, nA, mol_per_cm3, Mohm
>>> voltage
Dimension(name='voltage', i=-1, m=1, t=-3, l=2)
>>> nA
Unit(name='nA', dimension='current', power=-9)
```

Dimension and units implement multiplication/division operators to allow the quick creation of compound units and dimensions

```
>>> from nineml.units import mV, ms
>>> mV / ms
Unit(name='mV_per_ms', dimension='voltage_per_time', power=0)
```

1.8.2 Abstraction layer API

The abstraction layer is intended to provide explicit mathematical descriptions of any components used in a neuronal network model, where such components may be neuron models, synapse models, synaptic plasticity algorithms, connectivity rules, etc.

The abstraction layer therefore has a modular structure, to support different types of components, and allow extensions to the language. The current modules are:

dynamics: for describing hybrid dynamical systems, whose behaviour is governed both by differential equations and by discontinuous events. Such systems are often used to model point neurons, synapses and synaptic plasticity mechanisms.

connectionrule: a module containing several “built-in” connectivity rules (‘all-to-all’, etc.).

randomdistribution: a module for specifying random distributions.

Common types

Mathematics

Mathematical expressions are stored in Sympy objects throughout the Python NineML library. However, they are typically constructed by passing a string representation to a derived class of the `Expression` class (e.g. `Trigger`, `Alias`). The Sympy string parsing has been slightly extended to handle the ANSI-C-based format in the NineML specification, such as using the caret symbol to signify raising to the power of (Sympy uses the Python syntax of '**' to signify raising to the power of), e.g:

```
(3 * B + 1) * V ^ 2
```

Note: Currently, trigonometric functions are parsed as generic functions but this is planned to change in later versions of the library to use in-built Sympy functions. For the most part this will not have much effect on the represented expressions but in some cases it may prevent Sympy's solving and simplifying algorithms from making use of additional assumptions.

`dynamics` module

Ports

Time derivatives

Transitions

`connectionrule` module

`randomdistribution` module

1.8.3 User layer API

A NineML model is made up of populations of cells, connected via synapses, which may exhibit plasticity. The models for the cells, synapses and plasticity mechanisms are all instances of subclasses of `Component`. Populations of cells are represented by `Population`, the set of connections between two populations by `Projection`. Finally, the entire model is encapsulated in `Network`.

Components

References

NineML has three closely-related objects used to refer to other NineML objects. `Definition` is used inside `Components` to refer to abstraction layer `ComponentClass` definitions. `Prototype` is used inside `Components` to refer to previously-defined `Components`. `Reference` is used inside `Selections` to refer to `Population` objects, and inside `Projections` to refer to `Populations` and `Selections`.

Values and Physical Quantities

Properties

Populations

Projections

Networks

1.9 Release notes

All released NineML Python versions:

1.10 Getting help

For help using the NineML Python Library please contact the NeuralEnsemble Google group.

If you find a bug or would like to add a new feature to the Python `nineml` package, please go to https://github.com/INCF/nineml-python/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).

If you would like to propose a change to the specification, please see the issue tracker at https://github.com/INCF/nineml-spec/issues/.

CHAPTER 2

Developers' guide

2.1 Contributing to NineML

2.1.1 Mailing list

Discussions about Python `nineml` take place in the NeuralEnsemble Google Group.

2.1.2 Setting up a development environment

Requirements

In addition to the requirements listed in *Installation*, you will need to install:

- nose
- coverage

to run tests, and:

- Sphinx
- numpydoc

to build the documentation.

Code checkout

NineML development is based around GitHub. Once you have a GitHub account, you should fork the official NineML repository, and then clone your fork to your local machine:

```
$ git clone https://github.com/<username>/nineml-python.git nineml_dev
$ cd nineml_dev
```

To work on the development version:

```
$ git checkout master
```

To keep your NineML repository up-to-date with respect to the official repository, add it as a remote:

```
$ git remote add upstream https://github.com/INCF/nineml-python.git
```

and then you can pull in any upstream changes:

```
$ git pull upstream master
```

We suggest developing in a virtualenv, and installing `nineml` using:

```
$ pip install -e .
```

2.1.3 Coding style

We follow the PEP8 coding style. Please note in particular:

- indentation of four spaces, no tabs
- single space around most operators, but no space around the '=' sign when used to indicate a keyword argument or a default parameter value.
- we currently only Python version 2.7 but Python 3 support is planned.

2.1.4 Testing

Running the PyNN test suite requires the *nose_* packages, and optionally the *coverage_* package. To run the entire test suite, in the `lib9ml/python/test` subdirectory of the source tree:

```
$ nosetests unit
```

To see how well the codebase is covered by the tests, run:

```
$ nosetests --with-coverage --cover-package=nineml --cover-erase --cover-html test/
↪unittests
```

If you add a new feature to `nineml`, or fix a bug, you should write a unit test to cover the situation it arose.

Unit tests should where necessary make use of mock/fake/stub/dummy objects to isolate the component under test as well as possible.

2.1.5 Submitting code

The best way to get started with contributing code to NineML is to fix a small bug (bugs marked "minor" in the bug tracker) in your checkout of the code. Once you are happy with your changes, **run the test suite again to check that you have not introduced any new bugs**. If this is your first contribution to the project, please add your name and affiliation/employer to `lib9ml/python/AUTHORS`.

After committing the changes to your local repository:

```
$ git commit -m 'informative commit message'
```

first pull in any changes from the upstream repository:

```
$ git pull upstream master
```

then push to your own account on GitHub:

```
$ git push
```

Now, via the GitHub web interface, open a pull request.

2.1.6 Documentation

Python NineML documentation is generated using Sphinx.

To build the documentation in HTML format, run:

```
$ make html
```

in the `doc` subdirectory of the source tree. Some of the files contain examples of interactive Python sessions. The validity of this code can be tested by running:

```
$ make doctest
```

NineML documentation is hosted at http://readthedocs.org/nineml

2.1.7 Making a release

To make a release of NineML requires you to have permissions to upload Python NineML packages to the Python Package Index and the INCF Software Center. If you are interested in becoming release manager for Python NineML, please contact us via the mailing list.

When you think a release is ready, run through the following checklist one last time:

- do all the tests pass? This means running **`nosetests`** and **`make doctest`** as described above. You should do this on at least two Linux systems – one a very recent version and one at least a year old, and on at least one version of macOS.
- does the documentation build without errors? You should then at least skim the generated HTML pages to check for obvious problems.
- have you updated the version numbers in `setup.py`, `__init__.py`, `doc/source/conf.py` and `doc/source/installation.rst`?
- have you written release notes and added them to the documentation?

Once you've confirmed all the above, create a source package using:

```
$ python setup.py sdist
```

and check that it installs properly (you will find it in the `dist` subdirectory.

Now you should commit any changes, then tag with the release number as follows:

```
$ git tag x.y.z
```

where `x.y.z` is the release number.

If this is a development release (i.e. an *alpha* or *beta*), the final step is to upload the source package to the INCF Software Center. Do **not** upload development releases to PyPI.

To upload a package to the INCF Software Center, log-in, and then go to the Contents tab. Click on "Add new…" then "File", then fill in the form and upload the source package.

If this is a final release, there are a few more steps:

- if it is a major release (i.e. an `x.y.0` release), create a new bug-fix branch:

  ```
  $ git branch x.y
  ```

- upload the source package to PyPI:

  ```
  $ python setup.py sdist upload
  ```

- make an announcement on the mailing list
- if it is a major release, write a blog post about it with a focus on the new features and major changes.

2.2 Developer reference

The structure NineML Python library aims to closely match the NineML specification, with each NineML "layer" represented by a sub-package (i.e. `nineml.abstraction` and `nineml.user`) and each NineML type mapping to a separate Python class, with the exception of some simple types that just contain a single element (e.g. Size) or are used just to provide a name to a singleton child class (e.g. Pre, Post, etc…).

2.2.1 Base classes

There are number of base classes that should be derived from when designing NineML classes, which one(s) depend on the structure of the type, e.g. whether the contain annotations, child elements, or can be placed at the top-level of a NineML document.

BaseNineMLObject

All classes that represent objects in the "NineML object model" should derive from `BaseNineMLObject`.

`BaseNineMLObject` defines a number of common methods such as `clone`, `equals`, `write`, etc… (see NineML Types). As well as default values for class attributes that are required for all NineML classes, `nineml_type`, `nineml_attr`, `nineml_child`, `nineml_children`. These class attributes match the structure of the NineML specification and are used extensively within the visitor architecture (including serialization).

nineml_type

`nineml_type` should be a string containing the name of the corresponding NineML type in the NineML specification.

nineml_type_v1

If the nineml_type differs between v1 and v2 of the specification, `nineml_type_v1` should also be defined to hold the name of the type in the v1 syntax.

nineml_attr

`nineml_attr` should be a tuple of strings, listing the attributes of the given NineML class that are part of the NineML specification and are not NineML types themselves, such as `str`, `int` and `float` fields.

nineml_child

`nineml_child` should be a dictionary, which lists the names of singleton NineML child attributes in the class along with a mapping to their expected class. If the the child attribute can be one of several NineML classes then the attribute should map to None.

nineml_children

`nineml_children` should be a tuple listing the NineML classes that are contained within the object as children sets (e.g. `(Property, Initial)` for the `DynamicsProperties` class). Note that if a class has non-empty `nineml_children` it should derive from `ContainerObject`.

temporary

"Temporary" NineML objects are created when calling iterator properties and accessor methods of the `MultiDynamics` class that override corresponding in the `Dynamics` class, allowing `MultiDynamics` objects to duck-type (i.e. pretend to be) `Dynamics` objects. Such classes should override the `temporary` class attribute and set it to `True`. This prevents their address in memory being used to identify the object (e.g. in the cloner "memo" dictionary) as it since they are generated on the fly, this address will change between accesses.

Note: The `id` property in BaseNineMLObject should always be used to check whether two Python objects are the representing the same NineML object for this reason.

AnnotatedObject

The NineML specification states that all NineML objects can be annotated except `Annotations` objects themselves. Therefore, all bar `Annotations` NineML classes should derive from `AnnotatedObject`, which itself derives from `BaseNineMLObject`. This provides the `annotations` attribute, which can provides access to any annotations associated with the object.

ContainerObject

"Container classes" are classes that contain sets of children, such as Dynamics' (contains parameters, regimes, state-variables) or OnCondition (contains state assignments and output events), as opposed to classes that have nested singleton objects such as Dimension objects in Parameter objects. Such classes should derive from `ContainerObject`.

`ContainerObject` adds a number of convenient methods, including `add`, `remove`, and general iterators used to traverse the object hierarchy.

The `ContainerObject.__init__` method creates an `OrderedDict` for each child set with the name supplied by the child class' `_children_dict_name` method (which is `_<pluralized-lowercase-child-type>` by default).

Iterators and accessors

Container classes need to define three iterator properties and one accessor method for each children-set, corresponding to the method names supplied by the class methods in the child class, `_children_iter_name`, `_num_children_name`, `_children_keys_name` and `_child_accessor_name`. By default the method names returned by these class methods will be *<pluralized-lowercase-nineml_type>*, *num_<pluralized-lowercase-nineml_type>*, *<pluralized-lowercase-nineml_type>_names*, and *<lowercase-nineml_type>* respectively. These properties/method should return:

***children_iter*:** A property that returns an iterator over all children in the dictionary

***num_children* :** A property that returns the number of children in the dictionary:

***children_keys*:** A property that returns an iterator over the keys of the dictionary. If the child type doesn't have a `name` attribute then the iterator should be named <pluralized-lowercase-nineml-type>_keys instead.

***child_accessor*:** An accessor that takes the name (or key) of a child and returns the child.

Note: It would be possible to implement these properties/methods in the `ContainerObject` base class using `__getattr__` but since they are part of the public API that could be confusing to the user.

DocumentLevelObject

All NineML classes that are permitted at the top level in NineML documents (see Document-level types) need to derive from `DocumentLevelObject`, this provides `document` and `url` attribute properties and is also used in checks at various points in the code.

2.2.2 Visitors

Visitor patterns are used extensively within the NineML Python to find, validate, modify and analyze NineML structures, including their serialization.

Base Visitors

Visitor base classes are found in the `nineml.visitors.base` module, which search the object hierarchy and perform an "action" each object. These visitors use the `nineml_*` class attributes (see *BaseNineMLObject*) to navigate the object hierarchy and therefore can be used search to any NineML object.

If not overridden, the action method applied to each object will first check whether a specialized method for that type of object called `action_<lowercase-nineml_type>` has been implemented and call it if it has, otherwise call `default_action` method. Note that if specialized methods are not required then the visitor can just override the `action` method directly.

There are a number of different base visitor classes to derive from depending on the requirements of the visitor pattern in question.

BaseVisitor

If no contextual information or results of child objects are required then a visitor can derive directly from the `BaseVisitor` class. The action method will be called before child objects are actioned.

BaseVisitorWithContext

If contextual information is required, such as the parent container (and its parent, etc...) then the `BaseVisitorWithContext` can be derived instead. The immediate context is available via the `context` property and the context of all parent containers via the `contexts` attribute.

BaseChildResultsVisitor

For visitors that require the results of child objects (e.g. `Cloner`) to in their action methods. The child/children results can be accessed via the `child_result` and `children_result` dictionaries. If context information is also required use the `BaseChildResultsVisitorWithContext` visitor.

BasePreAndPostVisitor

For visitors the need to perform and action before and after the child results are actioned. The "pre" action methods are the same as in the `BaseVisitor` class and the "post" action method is called `post_action`, which by default will call the `post_action_<lowercase-nineml_type>` or `default_post_action` methods. If context information is also required use the `BasePreAndPostVisitorWithContext` visitor.

BaseDualVisitor

This visitor visits two objects side by side, raising exceptions if their structure doesn't match. As such it is probably only useful for equality checking (and is derived by the `EqualityChecker` and `MismatchFinder` visitors). A `BaseDualVisitorWithContext` visitor is also available.

Validation

Validation is currently only performed on component classes (i.e. `Dynamics`, `ConnectionRule`, and `RandomDistribution`). A separate visitor is implemented for every aspect of the component classes that need to be validated (e.g. name-conflicts, mismatching-dimensions).

Base validators are implemented in the `nineml.abstraction.componentclassvisitors.validators` package with specializations for each component class type in the corresponding `nineml.abstraction.<componentclass-type>.visitors.validators` packages (at this stage only the `Dynamics` component class has specialised validators).

Serialization

For serialization visitors to be able to serialize a NineML object it needs to define both `serialize_node` and `unserialize_node` methods.

`serialize_node`/`unserialize_node`

Both `serialize_node` and `unserialize_node` take a single argument, a `NodeToSerialize` or `NodeToUnSerialize` instance respectively. These node objects wrap a serial element of the given serialization format (e.g. `lxml.etree._Element` for the `XMLSerializer`) and provide convenient methods for adding, or accessing, children, attributes and body elements to the node.

The node method calls then call format-specific method of the serialization visitor to un/serialize the NineML objects. However, in some cases (particularly in some awkward v1.0 syntax), the serialization visitor may need to be accessed directly, which is available at `node.visitor`.

Both `serialize_node` and `unserialize_node` should accept arbitrary keyword arguments and pass them on to all calls made to methods of the nodes and the visitor directly. However, these arguments are not currently used by any of the current serializers.

`has_serial_body`

NineML classes that contain "body" text when serialized (to a supporting serial format) should override the class attribute `has_serial_body` to set it to `True`. If the class has a body only in NineML v1.0 syntax but not v2.0 then it should be set to `'v1'`.

NineML classes that just contain a single body element (e.g. `SingleValue`) should set has_serial_body to `'only'`, to allow them to be collapsed into an attribute in formats that don't support body text (i.e. YAML, JSON).