Nengo SpiNNaker Documentation

Release 0.0a1

Andrew Mundy, Terry Stewart

October 15, 2014

Contents

Nengo is a suite of software used to build and simulate large-scale brain models using the methods of the Neural Engineering Framework. SpiNNaker is a neuromorphic hardware platform designed to run large-scale spiking neural models in real-time. Using SpiNNaker to simulate Nengo models allows you to run models in real-time and interface with external hardware devices such as robots.

CHAPTER 1

Installation

1.1 Requirements

Nengo SpiNNaker requires that you have installed appropriate versions of Nengo and the SpiNNaker tools package.

1.2 Basic Installation

We're working towards providing the Nengo SpiNNaker package on PyPi at which point you will be able to:

```
pip install nengo_spinnaker
```

For now, like Nengo itself, do a developer installation.

1.3 Developer Installation

If you plan to make changes to Nengo SpiNNaker you should clone its git repository and install from it:

```
git clone https://github.com/ctn-waterloo/nengo_spinnaker
cd nengo_spinnaker
python setup.py develop --user
```

If you're in a virtualenv you can omit the `--user` flag.

1.3.1 Building the SpiNNaker binaries

If you installed the Nengo SpiNNaker package from source you will need to go through a few additional steps prior to running Nengo models. These steps build the executable binaries which are loaded to the SpiNNaker machine.

```
# Change to the root directory of the SpiNNaker package
# Edit `spinnaker_tools/setup` to point at your ARM cross-compilers
source ./setup
make

# Now move to the root directory of the Nengo SpiNNaker package
cd spinnaker_components
make
```

If you're new to Nengo we recommend reading through the Nengo documentation and trying a few examples before progressing on to running examples on SpiNNaker.

CHAPTER 2

Running Nengo models on SpiNNaker

If this is how your Nengo model currently works:

```
import nengo

model = nengo.Network()
with model:
    # ... Build a model
    a = nengo.Ensemble(100, dimensions=1)

sim = nengo.Simulator(model)
sim.run(10.)
```

Then porting it to Nengo SpiNNaker requires very few changes:

```
import nengo
import nengo_spinnaker

model = nengo.Network()
with model:
  # ... Build a model
  a = nengo.Ensemble(100, dimensions=1)

sim = nengo_spinnaker.Simulator(model)
sim.run(10.)
```

2.1 Nengo SpiNNaker Simulator

Some hardware is already supported by Nengo SpiNNaker and more will be added over time.

- "PushBot" - Neuroscientific System Theory (NST)

While Ensembles and various other components are simulated directly on the SpiNNaker board this is, in general, not possible for Nodes, which may be any arbitrary function.

CHAPTER 3

Running Nodes directly on SpiNNaker

By default Nodes are executed on the host computer and communicate with the SpiNNaker board to transmit and receive values. The result can be undesirable sampling of Node input and output.

For example:

```
import nengo
import nengo_spinnaker

model = nengo.Network()
with model:
    n = nengo.Node(np.sin)
    e = nengo.Ensemble(nengo.LIF(100), 1)
    p = nengo.Probe(e)

    nengo.Connection(n, e)

sim = nengo_spinnaker.Simulator(model)
sim.run(10.)
```

Results in:

For Nodes which are solely functions of time it is possible to precompute the output of the Node and play this back. Nodes with a constant output value and no input are automatically added to the bias current of Ensemble which they feed. Finally, more complex Nodes may be implemented as SpiNNaker executables and directly executed on the SpiNNaker hardware.

3.1 Function-of-time Nodes

Nodes which are purely functions of time may be precomputed for the duration of the simulation (or the period of the function if appropriate) and played back during the simulation. Nodes you wish to be executed in this way must be marked with an appropriate directive:

```
# As before...

# Create the configuration and configure `n` as being f(t)
config = nengo_spinnaker.Config()
config[n].f_of_t = True  # Mark Node as being a function of time
config[n].f_period = 2*np.pi  # Mark the period of the function

# Pass the configuration to the simulator
sim = nengo_spinnaker.Simulator(model, config=config)
```

Results in:

The two directives are:

- *f_of_t* marks a Node as being precomputable. This is not checked - be careful!
- *f_period* marks the period of the function in seconds. If this is *None* then the Node will be precomputed for the entire duration of the simulation - it is possible to run out of memory. Again, this cannot be trivially validated.

3.2 Writing a SpiNNaker executable for a Node

3.3 Using the binary

3.4 Configuring the executable instances

CHAPTER 4

Writing new Input/Output Handlers

Input/Output Handlers manage the communication between the host and the SpiNNaker machine running the simulation. This entails two tasks:

1. Modifying the SpiNNaker model to include appropriate executables and connections for handling Node input/output.
2. Providing functions for getting input for Nodes and setting Node output.

The first of these tasks is handled by "Node Builders", the second by "Node Communicators".

4.1 Node Builders

When building a model for simulation a `nengo_spinnaker.builder.Builder` delegates the tasks of building Nodes and the connections into or out of Nodes to a Node Builder.

Additionally, the `nengo_spinnaker.Simulator` will expect the Node Builder to provide a context manager for the Node Communicator.

A Node Builder is expected to look like the following:

class `GenericNodeBuilder`

`get_node_in_vertex`(*self*, *builder*, *connection*)
Get the PACMAN vertex where input to the Node should be sent.

Parameters

- **builder** – A `nengo_spinnaker.builder.Builder` instance providing *add_vertex* and *add_edge* methods.
- **connection** – A `nengo.Connection` object which specifies the connection being built. The Node will be referred to by *connection.post*.

Returns The PACMAN vertex where input for the Node at the end of the given connection should be sent.

It is expected that this function will need to create new PACMAN vertices and add them to the graph using the builder object.

`get_node_out_vertex`(*self*, *builder*, *connection*)
Get the PACMAN vertex where output from the Node can be expected to arrive in the SpiNNaker network.

Parameters

- **builder** – A `nengo_spinnaker.builder.Builder` instance providing *add_vertex* and *add_edge* methods.
- **connection** – A `nengo.Connection` object which specifies the connection being built. The Node will be referred to by *connection.pre*.

Returns The PACMAN vertex where output from the Node will appear.

It is expected that this function will need to create new PACMAN vertices and add them to the graph using the builder object.

build_node(*self*, *builder*, *node*)
Perform any tasks necessary to build a Node which is neither constant nor a function of time.

Parameters

- **builder** – A `nengo_spinnaker.builder.Builder` instance providing *add_vertex* and *add_edge* methods.
- **node** – The `nengo.Node` object for which to provide IO.

Note: In all current implementations this method does nothing, it is generally more useful to instantiate any vertices or edges when connecting to or from a Node.

---

io
A reference to the Communicator object.

__enter__(*self*)
Create and return a Communicator to handle input/output for Nodes.

Returns A Communicator of the appropriate type.

__exit__(*self*, *exception_type*, *exception_value*, *traceback*)
Perform any tasks necessary to stop the Communicator from running.

4.2 Node Communicators

The `nengo_spinnaker.Simulator` delegates the task of getting Node input and setting Node output to a communicator which is generated by the Node Builder.

A Node Communicator is required to look like the following:

class GenericNodeCommunicator

Warning: It is required that the Communicator be thread safe. Each Node is independently responsible for getting its input and setting its output and each Node will be executed within its own thread.

start(*self*)
Start execution of the communicator thread.

get_node_input(*self*, *node*)
Return the latest received input for the given Node.

Parameters **node** – A `nengo.Node` for which input is desired.

Returns The latest received value as a Numpy array, or *None* if no data has yet been received from the Node.

Raises `KeyError` if the Node is not recognised by the Communicator.

set_node_output(*self*, *node*, *output*)
Transmit the output of the Node to the SpiNNaker board.

Parameters

- **node** – A `nengo.Node` for which output is being provided.
- **output** – The latest output from the Node.

Raises `KeyError` if the Node is not recognised by the Communicator.

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

Symbols

B

G

I

S