
nengo*mpi0.1.0.dev0docs*

Release 0.1.0.dev0

Eric Crawford

February 21, 2017

1	Getting Started	3
1.1	Installation	3
1.2	Adapting Existing Nengo Scripts	3
1.3	Running scripts	5
2	User Guide	7
2.1	Installing nengo_mpi	7
2.2	How It Works	9
2.3	Workflows	10
2.4	Modules	11
2.5	Benchmarks	13
2.6	FAQ	14
3	Developer Guide	17
3.1	Building the documentation	17
3.2	Development workflow	17

`nengo_mpi` is a C++/MPI backend for `nengo`, a python library for building and simulating biologically realistic neural networks. `nengo_mpi` makes it possible to run `nengo` simulations in parallel on thousands of processors, and existing `nengo` scripts can be adapted to make use of `nengo_mpi` with minimal effort.

With an MPI implementation installed on the system, `nengo_mpi` can be used to run neural simulations in parallel using just a few lines of code:

```
import nengo
import nengo_mpi
import numpy as np
import matplotlib.pyplot as plt

with nengo.Network() as net:
    sin_input = nengo.Node(output=np.sin)

    # A population of 100 neurons representing a sine wave
    sin_ens = nengo.Ensemble(n_neurons=100, dimensions=1)
    nengo.Connection(sin_input, sin_ens)

    # A population of 100 neurons representing the square of the sine wave
    sin_squared = nengo.Ensemble(n_neurons=100, dimensions=1)
    nengo.Connection(sin_ens, sin_squared, function=np.square)

    # View the decoded output of sin_squared
    squared_probe = nengo.Probe(sin_squared, synapse=0.01)

partitioner = nengo_mpi.Partitioner(2)
sim = nengo_mpi.Simulator(net, partitioner=partitioner)
sim.run(5.0)

plt.plot(sim.trange(), sim.data[squared_probe])
plt.show()
```

There are 3 differences between this script and a script using the reference implementation of `nengo`. First, we need to import `nengo_mpi`. Then we need to create a `Partitioner` object, which specifies how many components the `nengo` network should be split up into (this corresponds to the maximum number of distinct processors that can be used to run the simulation). Finally, when creating the simulator we need to use `nengo_mpi`'s `Simulator` class, and we need to pass in the `Partitioner` instance.

The script can then be run in parallel using:

```
mpirun -np 2 python -m nengo_mpi <script_name>
```

`sin_ens` and `sin_squared` will be simulated on separate processors, with the output of `sin_ens` being passed to the input of `sin_squared` every time-step using MPI. After the simulation has completed, the probed results from all processors are passed back to the main processor, so all probed data can be accessed in the usual way (e.g. `sim.data[squared_probe]`).

`nengo_mpi` is fully featured, supporting all aspects of Nengo Release 2.0.2.

Getting Started

Installation

As of this writing, `nengo_mpi` is only known to be usable on Linux. Installation is straightforward. On an Ubuntu workstation (as opposed to a cluster) an OpenMPI based installation can be obtained by first installing dependencies:

```
sudo apt-get install openmpi-bin libopenmpi-dev libhdf5-openmpi-dev
sudo apt-get install libboost-dev libatlas-base-dev
```

Then download the code:

```
git clone https://github.com/nengo/nengo_mpi.git
```

And install `nengo_mpi` and all its python dependencies (including `nengo`):

```
cd nengo_mpi
pip install --user .
```

or if inside a `virtualenv`:

```
cd nengo_mpi
pip install .
```

And run the test to make sure it works:

```
py.test
```

See *Installing `nengo_mpi`* for more detailed installation instructions, especially for installing on high-performance clusters.

Adapting Existing Nengo Scripts

Existing `nengo` scripts can be adapted to make use of `nengo_mpi` by making just a few small modifications. The most basic change that needs to be made is importing `nengo_mpi` in addition to `nengo`, and then using the `nengo_mpi.Simulator` class in place of the `Simulator` class provided by `nengo`

```
import nengo_mpi
import nengo

... Code to build network ...

sim = nengo_mpi.Simulator(network)
sim.run(1.0)

plt.plot(sim.trange(), sim.data[probe])
```

This will run a simulation using the `nengo_mpi` backend, but does not yet take advantage of parallelization. However, even without parallelization, the `nengo_mpi` backend can often be quite a bit faster than the reference implementation (see our [Benchmarks](#)) since it is a C++ library wrapped by a thin python layer, whereas the reference implementation is pure python.

Partitioning

In order to have simulations run in parallel, we need a way of specifying which nengo objects are going to be simulated on which processors. A `Partitioner` is the abstraction we use to do this specification. The most basic information that a partitioner requires is the number of components to split the network into. We can supply this information when creating the partitioner, and then pass the partitioner to the `Simulator` object:

```
partitioner = nengo_mpi.Partitioner(n_components=8)
sim = nengo_mpi.Simulator(network, partitioner=partitioner)
sim.run(1.0)
```

The number of components we specify here acts as an upper bound on the effective number of processors that can be used to run the simulation.

We can also specify a partitioning function, which accepts a graph (corresponding to a nengo network) and a number of components, and returns a python dictionary which gives, for each nengo object, the component it has been assigned to. If no partitioning function is supplied, then a default is used which simply assigns each component a roughly equal number of neurons. A more sophisticated partitioning function (which has additional dependencies) uses the `metis` package to assign objects to components in a way that minimizes the number of nengo `Connections` that straddle component boundaries. For example:

```
partitioner = nengo_mpi.Partitioner(n_components=8, func=nengo_mpi.metis_partitioner)
sim = nengo_mpi.Simulator(network, partitioner=partitioner)
sim.run(1.0)
```

For small networks, we can also supply a dict mapping from nengo objects to component indices:

```
model = nengo.Network()
with model:
    A = nengo.Ensemble(n_neurons=50, dimensions=1)
    B = nengo.Ensemble(n_neurons=50, dimensions=1)
    nengo.Connection(A, B)

assignments = {A: 0, B: 1}
```



```
sim = nengo_mpi.Simulator(model, assignments=assignments)
sim.run(1.0)
```

Note, though, that this does not scale well and should be reserved for toy networks/demos.

Running scripts

To use the nengo_mpi backend without parallelization, scripts modified as above can be run in the usual way

```
python nengo_script.py
```

This will run serially, even if we have used a partitioner to specify that the network be split up into multiple components. When a script is run, nengo_mpi automatically detects how many MPI processes are active, and assigns components to each process. In this case only one process (the master process) is active, and all components will be assigned to it.

In order to get parallelization we need a slightly more complex invocation:

```
mpirun -np NP python -m nengo_mpi nengo_script.py
```

where NP is the number of MPI processes to launch. Its fine if NP is not equal to the number of components that the network is split into; if NP is larger, then some MPI processes will not be assigned any component to simulate, and if NP is smaller, some MPI processes will be assigned multiple components to simulate.

Installing nengo_mpi

Basic Installation (Ubuntu)

This section outlines how to install nengo_mpi on a typical workstation running Ubuntu. Other Linux distributions should be able to use these instructions as well, appropriately adapted. If installing on a high-performance cluster, skip to *Installing On High-Performance Clusters*.

Dependencies

nengo_mpi requires working installations of:

- MPI
- HDF5 (a version compatible with the installed MPI implementation)
- BLAS
- Boost

To satisfy the Boost and BLAS requirements, the following should work on any Ubuntu version after 14.04:

```
sudo apt-get install libboost-dev libatlas-base-dev
```

Installing MPI depends on which implementation you want to use:

1. **OpenMPI** - On Ubuntu 14.04 and later the following should be sufficient for getting obtaining OpenMPI and the remaining requirements:

```
sudo apt-get install openmpi-bin libopenmpi-dev libhdf5-openmpi-dev
```

2. **MPICH** - On Ubuntu 16.04 (Xenial) the requirements can be satisfied using MPICH with the following invocation:

```
sudo apt-get install mpich libmpich-dev libhdf5-mpich-dev
```

and on Ubuntu 14.04 (Trusty) the following will do the trick:

```
sudo apt-get install mpich libmpich-dev libhdf5-mpich2-dev
```

3. Other MPI Implementations - May work but have not been tested.

Installation

nengo_mpi is not currently available on PyPI, but can nevertheless be installed via pip once you've downloaded the code. First obtain a copy of the code from github:

```
git clone https://github.com/nengo/nengo_mpi.git
cd nengo_mpi
```

If using the MPICH MPI implementation, we also have to tell the compiler where to find various header files and libraries. Such information is read from the file `mpi.cfg`. The default `mpi.cfg` is setup to work with OpenMPI. We've included configuration files for MPICH in the `conf` directory, and making use of them is just a matter of putting them in the right place.

On Ubuntu 16.04 (Xenial):

```
cp conf/mpich.cfg mpi.cfg
```

and on Ubuntu 14.04 (Trusty):

```
cp conf/mpich_trusty.cfg mpi.cfg
```

Finally install nengo_mpi:

```
pip install --user .
```

If you're inside a virtualenv (recommended!) you can omit the `--user` flag. If you're developing on nengo_mpi, you can also add the `-e` flag so that changes you make to the code will be reflected in your python environment. You can also add `--install-options="-n"` to install without building the C++ portion of nengo_mpi, which can be useful when installing nengo_mpi for building network files that you intend to simulate on a different machine.

Installing On High-Performance Clusters

High-performance computing environments typically place additional constraints on installing software, so the above installation process may not be available in all such environments. Here we provide some pointers for getting nengo_mpi installed on such environments, with more specific advice for the particular supercomputers that nengo_mpi is most often used on, namely Scinet's General Purpose Cluster and BlueGeneQ.

Dependencies

Installation

With Python

Without Python

How It Works

Here we attempt to give a rough idea of how `nengo_mpi` works under the hood, and, in particular, how it achieves parallelization. `nengo_mpi` is based heavily on the reference implementation of `nengo`. The reference implementation works by converting a high-level neural model specification into a low-level computation graph. The computation graph is a collection of operators and signals. In short, signals store data, and operators perform computation on signals and store the results in other signals. To run the simulation, `nengo` simply executes each operator in the computation graph once per time step. For a concrete example of how this works, consider the following simple `nengo` script:

```
import nengo

model = nengo.Network()
with model:
    A = nengo.Ensemble(n_neurons=50, dimensions=1)
    B = nengo.Ensemble(n_neurons=50, dimensions=1)
    conn = nengo.Connection(A, B)

sim = nengo.Simulator(model)
sim.run(time_in_seconds=1.0)
```

The conversion from the high-level specification (e.g. the `nengo Network` stored in the variable `model`) to computation graph is called the **build** step, and takes place in the line `sim = nengo.Simulator(model)`. The generated computation graph looks something like this:

A few signals and operators whose purposes are somewhat opaque have been omitted here for clarity. Now suppose that we're impatient and find that the call to `sim.run` is too slow. We can easily parallelize the simulation step by making use of `nengo_mpi`. Making the few necessary changes, we end up with the following script:

```
import nengo
import nengo_mpi

model = nengo.Network()
with model:
    A = nengo.Ensemble(n_neurons=50, dimensions=1)
    B = nengo.Ensemble(n_neurons=50, dimensions=1)
    nengo.Connection(A, B)

# assign the ensembles to different processors
assignments = {A: 0, B: 1}
sim = nengo_mpi.Simulator(model, assignments=assignments)

sim.run(time_in_seconds=1.0)
```

Now ensembles A and B will be simulated on different processors, and we should get a factor of 2 speedup in running the simulation (though it will hardly be perceptible given how tiny our network is). `nengo_mpi` will produce a computation graph quite similar to the one produced by vanilla `nengo`, except it will use operators that are implemented in C++ rather than python, and will add a few new operators to achieve the

inter-process communication:

The `MPISend` operator stores the index of the processor to send its data to, and likewise the `MPIRecv` operator stores the index of the processor to receive data from. Moreover, they both share a “tag”, a unique identifier which bonds the two operators together and ensures that the data from the `MPISend` operator gets sent to the correct `MPIRecv` operator. This basic pattern can be scaled up to simulate very large networks on thousands of processors.

Some readers may have noticed something odd by now: it may seem like it would be impossible to achieve accelerated performance from the set-up depicted in the above diagrams. In particular, it seems as if the operators on processor 1 will need to wait for the results from processor 0, so the computation is still ultimately a serial one, just that now we have added inter-process communication in the pipeline to slow things down.

This turns out not to be the case, because the `Synapse` operator is special in that it is what we call an “update” operator. Update operators break the computation graph up into independently-simulatable components. In the first diagram, the `DotInc` operator in ensemble B performs computation on the value of the *Input signal from the previous time-step*¹. Thus, the operators in ensemble B do not need to wait for the operators in ensemble A and the connection, since the values from the previous time-step should already be available. Likewise, in the second diagram, the `MPIRecv` operator actually receives data from the previous time-step. Thanks to this mechanism, we are in fact able to achieve large-scale parallelization, demonstrated empirically by our *Benchmarks*.

Workflows

There are two distinct ways to use `nengo_mpi`.

1. Build With Python, Simulate From Python

This is the most straightforward way to run simulations. Existing `nengo` scripts can quickly be adapted to use `nengo_mpi` with this method. This workflow is described in *Getting Started*.

2. Build With Python, Simulate Using Stand-Alone Executable

Using this workflow, the process of building networks is similar to the first workflow, while the process of running the simulations is quite different. This approach offers more flexibility, allowing simulations to be built on one computer (which we’ll call the “build” machine) with full python support but no MPI installation, and then simulated on another computer (which we’ll call the “sim” machine) with a full MPI installation but no python support (e.g. a cluster).

One point to be aware of with this method is that it has some limitations. In particular, it cannot deal with networks containing non-trivial `nengo` Nodes. The reason is that at simulation time, python will be completely out of the picture, so there is no way to execute the python code that Nodes contain. It is possible that this could be fixed in the future by spinning up a python interpreter at simulation time, though this would involve a significant amount of work. At the present time, the only `nengo` Nodes allowed are

¹ “Delays” like this are necessary from a biological-plausibility standpoint as well. Otherwise, neural activity elicited by a stimulus could be propagated throughout the entire network in a single time step, regardless of the network’s size.

passthrough nodes, nodes that output a constant signal, and SpaunStimulus nodes. The first two are trivial to implement, and the third we have made special accommodations for.

Building and Saving a Network

The first step is to build a network and save it to a file. To do this, we need to make a change to how we call `nengo_mpi.Simulator`. In particular, we supply the `save_file` argument:

```
sim = nengo_mpi.Simulator(model, partitioner=partitioner, save_file="model.net")
```

This call will create a file called `model.net` in the current directory, which stores the operators and signals required to simulate the nengo Network specified by `model`. This file will actually be an HDF5 file, but we typically give it the `.net` extension to indicate that it stores a built network. The script can then be executed (on the “build” machine) using a simple invocation:

```
python nengo_script.py
```

Loading and Simulating a Network

Now we can make use of the network file we’ve created using the `nengo_mpi` executable (see [Modules](#) for more info on the executable). Assuming that we are now on the “sim” machine, and that the `nengo_mpi` executable has been compiled, we can run:

```
mpirun -np NP nengo_mpi model.net 1.0
```

where `NP` is the number of MPI processors to use. This will simulate the network stored in `model.net` for 1 second of simulation time.

The result of the simulation (the data collected by the probes) will be stored in an HDF5 file called `model.h5`. We can specify a different name for the output file as follows:

```
mpirun -np NP nengo_mpi --log results.h5 model.net 1.0
```

Finally, if MPI is not available on the “sim” machine, we can instead use:

```
nengo_cpp --log results.h5 model.net 1.0
```

but this will run serially.

Modules

`nengo_mpi` is composed of several fairly separable modules.

python

The python code consists primarily of alternate implementations of both `nengo.Simulator` and `nengo.Model`. `nengo_mpi.Simulator` is a wrapper around the C++/MPI code which provides an interface nearly identical to `nengo.Simulator` (see [Getting Started](#)). The `nengo_mpi.Model` class

is primarily responsible for adapting the output of the reference implementation's *build* step (converting from a high-level model specification to a concrete computation graph; see [How It Works](#)) to work with `nengo_mpi.Simulator`.

The final major chunk of python code handles the complex task of partitioning a nengo Network into a desired number of components that can be simulated independently.

C++

The directory `mpi_sim` contains the C++ code. This code implements a back-end for nengo which can use MPI to run simulations in parallel. The C++ code only implements simulation capabilities; the *build* step is still done in python, and `nengo_mpi` largely uses the builder provided by the reference implementation. The C++ code can be used in at least three different ways.

`mpi_sim.so`

A shared library that allows the python layer to access the core C++ simulator. The python code creates an HDF5 file encoding the built network (the operators and signals that need to be simulated), and then makes a call out to `mpi_sim.so` with the name of the file. `mpi_sim.so` then opens the file (in parallel if there are multiple processors active) and runs the simulation.

`nengo_mpi` executable

This is an executable that allows the C++ simulator to be used directly, instead of having to go through python. The executable accepts as arguments the name of a file specifying the operators and signals in a network, as well as the length of time to run the simulation for, in seconds. Removing the requirement that the C++ code be accessed through python has a number of advantages. In particular, it can make attaching a debugger much easier. Also, some high-performance clusters (e.g. BlueGene) provide only minimal support for python. The `nengo_mpi` executable has no python dependencies, and so it can be used on these machines. A typical workflow is to use the python code to create the HDF5 file on a machine with full python support, and then transfer that file over to the high-performance cluster where the network encoded by that file can be simulated using the `nengo_mpi` executable. See [Workflows](#) for further details.

`nengo_cpp` executable

This is just a version of the `nengo_mpi` executable which does not have MPI dependencies ¹ (which, of course, means that there is no parallelization). It is possible that some users may find this useful in some situations where the MPI dependency cannot be met, as the C++ simulator is often significantly faster than the reference implementation simulator even without parallelization (see our [Benchmarks](#)).

¹ This is currently a lie, but it will be true soon.

Benchmarks

Benchmarks testing the simulation speed of nengo_mpi were performed with 3 different machines and 3 different large-scale spiking neural networks. The machines used were a home PC with a Quad-Core 3.6GHz Intel i7-4790 and 16 GB of RAM, Scinet’s [General Purpose Cluster](#), and Scinet’s 4 rack [Blue Gene/Q](#). We tested nengo_mpi using different numbers of processors, and also tested the [reference implementation](#) of nengo (on the home PC only) for comparison.

Stream Network

The stream network exhibits a simple connectivity structure, and is intended to be close to the optimal configuration for executing a simulation quickly in parallel using nengo_mpi. In particular, the ratio of the amount of communication vs computation per step is relatively low. The network takes a single parameter n giving the total number of neural ensembles. The network contains \sqrt{n} different “streams”, where a stream is a collection of \sqrt{n} ensembles connected in a circular fashion (so each ensemble has 1 incoming and 1 outgoing connection). Each ensemble is 4-dimensional and contains 200 LIF neurons, and we vary n as the independent variable in the graphs below. The largest network contains $2^{12} = 4096$ ensembles, for a total of $200 * 4096 = 819,200$ neurons. In every case, the ensembles are distributed evenly amongst the processors. Each execution consists of 5 seconds of simulated time, and each data point is the result of 5 separate executions.

Random Graph Network

The random graph network is constructed by choosing a fixed number of ensembles, and then randomly choosing ensemble-to-ensemble connections to instantiate until a desired proportion of the total number of possible connections is reached. In all cases, we use 1024 ensembles, and we vary the proportion of connections. This network is intended to show how the performance of nengo_mpi scales as the ratio of communication to computation increases, and investigate whether it is always a good idea to add more processors. Adding more processors typically increases the amount of inter-processor communication, since it increases the likelihood that any two ensembles are simulated on different processors. Therefore, if communication is the bottleneck, then adding more processors will tend to decrease performance.

Each ensemble is 2-dimensional and contains 100 LIF neurons, and each connection computes the identity function. With n ensembles, there are n^2 possible connections (since we allow self-connections and the connections are directed). Therefore in the most extreme case we have $0.2 \times 1024^2 \approx 209,715$ connections, each relaying a 2-dimensional value. The number of such connections that need to engage in inter-processor communication is a function of the number of processors used in the simulation, and the particular random connectivity structure that arose. Each execution consists of 5 seconds of simulated time, and each data point is the average of executions on 5 separate networks with different randomly chosen connectivity structure.

SPAUN

SPAUN (Semantic Pointer Architecture Unified Network) is a large-scale, functional model of the human brain developed at the CNRG. It is composed entirely of spiking neurons, and can perform eight separate cognitive tasks without modification. The original paper can be found [here](#). SPAUN is an extremely

large-scale spiking neural network (currently approaching 4 million neurons when using 512-dimensional semantic pointers) with very complex connectivity, and represents a somewhat more realistic test than the more contrived examples used above. In the plots below we vary the dimensionality of the semantic pointers, the internal representations used by SPAUN.

Clearly the larger clusters are providing less of a benefit here. The hypothesized reason is that thus far we have been unable to split SPAUN up into sufficiently small components than can be simulated independently. There are some components with many thousands of neurons on them. Thus the limiting factor for the speed of simulation is how quickly an individual processor is able to simulate one of these large components. We can likely remedy this by playing around with the connectivity of SPAUN and finding ways to reduce the maximum component size.

FAQ

Is there any build step parallelization?

No, nengo_mpi only provides parallelization for the simulation step. The build step is where all the really difficult stuff happens, which, for instance, makes an Ensemble act like an Ensemble. Therefore, nengo_mpi simply uses vanilla nengo's builder, which runs serially in python.

During an invocation such as:

```
mpirun -np 8 python -m nengo_mpi nengo_script.py
```

the build step is performed entirely by the process with index 0.

It is definitely possible to create a parallelized version of the builder. However, that should probably use a more python-friendly, platform-agnostic technology than MPI (something like ZeroMQ). In other words, that's another project.

What is the difference between a cluster, a component, a partition, a chunk, a process, a processor, and a node? I've seen all these words used in the code with apparently similar meanings.

All these terms do in fact have precise meanings in the context of nengo_mpi. They can nicely be divided up into terms that apply at build time and terms that apply at simulation time.

• Build Time

- A `cluster` (distinct from a cluster of machines in high-performance computing) is a group of nengo objects that must be simulated together, for any of a number of reasons (see the class *NengoObjectCluster* in *partition/base.py*). The most prominent reason is that there is an path of Connections between the two objects that does not have a synapse (since synapses are the main source of “update” operators; see *How It Works*). Another common reason is that the two objects are connected by a Connection which has a learning rule. The partitioning step applies a partitioning function to a graph whose nodes are `clusters`.
- A `component` (as in a component of a partition) is a group of `clusters` that will be simulated together. Components are computed by the partitioning step. When creating an instance of `nengo_mpi.Simulator`, we typically specify the number of components that we want

the network to be divided into. When `nengo_mpi` saves a network to file for communication with the C++ code, each `component` is stored separately.

- A `partition` is a collection of `components`. The goal of the partitioning step is to create a partition of the set of clusters, in the sense used [here](#). High-quality partitions are those which do not assign drastically different amounts of work to different components, and which minimize the amount of communication between components.

- **Simulation Time**

- A `process` is, of course, an OS abstraction for a line of computation. A `processor` is a physical computation device. `Processes` run on `processors`. It is generally possible to run a `nengo_mpi` simulation using more `processes` than there are `processors` available on the machine, however the amount of parallelization we can obtain is determined by the number of physical `processors` (though hyperthreading can increase the effective number of `processors`). The number of `processes` used to run a simulation is specified by the `-np <NP>` command-line argument when calling `mpi_run`.
- A `chunk` (see `chunk.hpp`) is the C++ code's abstraction for a collection of `nengo` objects (actually, signals and operators corresponding to those objects) that are being simulated by a single `process`. There is a one-to-one relationship between `chunks` and `processes`. One of the first things that each `process` does is create a `chunk`.
- The relationship between `chunks/processes` and `components` is as follows. At build time the network is divided into some specified number of `components` by partitioning. At simulation time, some specified number of `chunks/processes` will be active. `Components` are assigned to `chunks/processes` in a round-robin fashion. For example, if there are 4 `chunks/processes` active and the network to simulate has 7 `components`, then `process 0` simulates `components 0` and `4`, `process 1` simulates `1` and `5`, etc. If the network instead had only 3 `components`, then `process 3` would be left without anything to simulate, which is perfectly OK.
- In the world of High-Performance Computing, a `node` (distinct from a `nengo Node`) is a physical computer consisting of some number of `processors`. On the General Purpose Cluster there are 8 `processors` per `node` and on Bluegene/Q there are 16 (that becomes 16 for GPC and 64 for BGQ once hyperthreading is taken into account). When running on one of these high-performance clusters, jobs are assigned computational resources in units of `nodes` rather than `processors`.

Developer Guide

Building the documentation

We use [the same process as nengo](#) to build the documentation.

Development workflow

Development happens on [Github](#). Feel free to fork any of our repositories and send a pull request! However, note that we ask contributors to sign a [copyright assignment agreement](#).

Code style

For python code, we use the same conventions as nengo: PEP8, flake8 for checking, and numpydoc For docstrings. See the [nengo code style](#) guide.

For C++ code, we roughly adhere to Google's [style guide](#).

Unit testing

We use [PyTest](#) to run our unit tests on [Travis-CI](#). We run nengo's full test-suite using nengo_mpi as a back-end. We also have a number of tests to explicitly ensure that results obtained using nengo_mpi are the same as those obtained using Nengo to a high-degree of accuracy. Tests are run using the invocation:

```
py.test
```