NSuite

Jun 27, 2022

Contents

1 Getting Started **3**

2 Funding **5**

3 Contents **7**

3.1 Installing NSuite . . . 7
3.2 Running NSuite . . . 12
3.3 Benchmarks . . . 17
3.4 Validation . . . 17
3.5 Simulation Engines . . . 20

NSuite is a framework for maintaining and running benchmarks and validation tests for multi-compartment neural network simulations on HPC systems. NSuite automates the process of building simulation engines, and running benchmarks and validation tests. NSuite is specifically designed to allow easy deployment on HPC systems in testing workflows, such as benchmark-driven development or continuous integration.

There are three motivations for the development of NSuite:

1. The need for a definitive resource for comparing performance and correctness of simulation engines on HPC systems.
2. The need to verify the performance and correctness of individual simulation engines as they change over time.
3. The need to test that changes to an HPC system do not cause performance or correctness regressions in simulation engines.

The framework currently supports the simulation engines Arbor, NEURON, and CoreNeuron, while allowing other simulation engines to be added.

CHAPTER 1

Getting Started

NSuite implements a simple workflow with two stages using bash scripts:

1. Compile and install simulation engines.
2. Run and record results from benchmarks and validation tests.

Below is the simplest example of a workflow that compiles all simulation engines and runs benchmarks and validation tests:

```
# clone the NSuite framework from GitHub
git clone https://github.com/arbor-sim/nsuite.git
cd nsuite/

# install Arbor, NEURON and CoreNeuron
./install-local.sh arbor neuron coreneuron

# run the ring and kway benchmarks in small configuration for Arbor, NEURON and CoreNeuron
./run-bench.sh arbor neuron coreneuron --model="ring kway" --config=small

# run all validation tests for Arbor and NEURON
./run-validation.sh arbor neuron
```

HPC systems come in many different configurations, and often require a little bit of “creativity” to install and run software. Users of NSuite can customise the environment and how simulation engines are built and run by providing configuration scripts, which is covered along with details about the simulation engines in the *simulation engine documentation*.

More information about running and writing new tests can be found in the *benchmark* and *validation* documentation respectively.

CHAPTER 2

Funding

NSuite is developed as a joint collaboration between the Swiss National Supercomputing Center (CSCS), and Forschungszentrum Jülich, as part of the Human Brain Project (HBP).

Development was fully funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreement No. 785907 (Human Brain Project SGA2).

CHAPTER 3

Contents

3.1 Installing NSuite

The first stage of the NSuite workflow is to install the simulation engine(s) to benchmark or validate. This page describes how to obtain NSuite and then perform this step so that benchmarks and validation tests can be run.

3.1.1 Obtaining NSuite

Before installing, first get a copy of NSuite. The simplest way to do this is to clone the repository using git:

```
git clone https://github.com/arbor-sim/nsuite.git
cd nsuite
git checkout v1.0
```

In the example above, `git checkout v1.0` is used to pick a tagged version of NSuite. If omitted, the latest development version in the master branch will be used.

3.1.2 Installing Simulation Engines

NSuite provides a script `install-local.sh` that performs the following operations:

- Obtain the source code for simulation engines.
- Compile and install the simulation engines.
- Compile and install benchmark and validation test drivers.

Basic usage of `install-local.sh` is best illustrated with some examples:

```
# download and install Arbor
./install-local.sh arbor

# download and install NEURON and CoreNEURON
```

(continues on next page)

(continued from previous page)

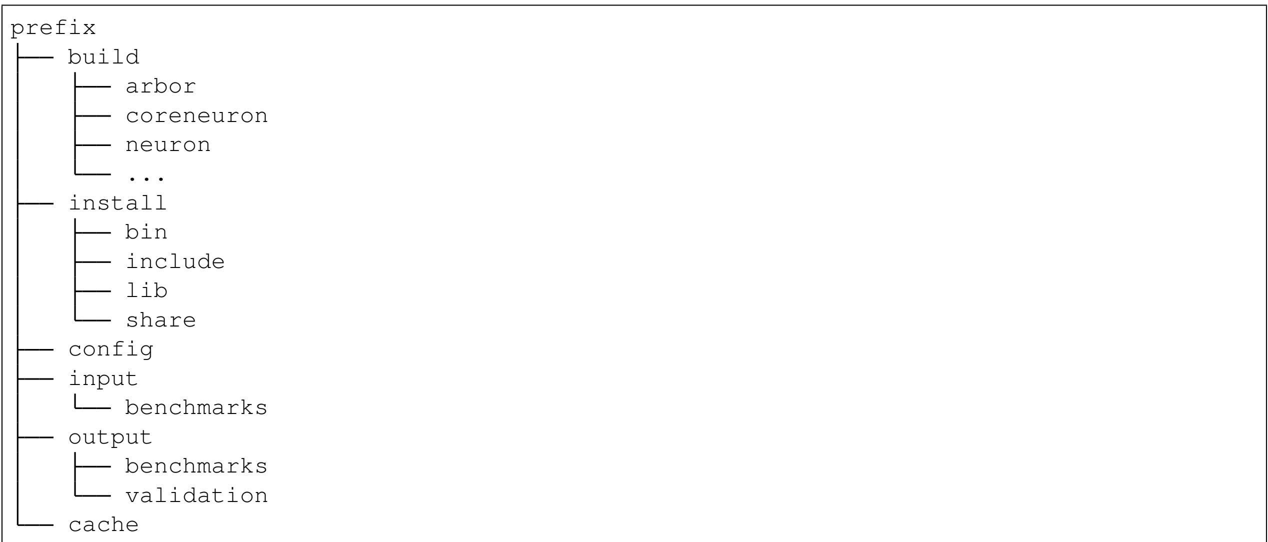
```
./install-local.sh neuron coreneuron

# download install all three of Arbor, NEURON and CoreNEURON
./install-local.sh all

# download install NEURON in relative path install
./install-local.sh neuron --prefix=install

# download install NEURON in relative path that includes time stamp
# e.g. install-2019-03-22
./install-local.sh neuron --prefix=install-$(date +%F)

# download install NEURON in absolute path
./install-local.sh neuron --prefix=/home/uname/install
```

The simulation engines to install are provided as arguments. Further options for installing the simulation engines in a user-specified path and customising the build environment can be provided:

Flag	Default value	Explanation
simulator	none	Which simulation engines to download and install. Any number of the following: {`arbor`, `neuron`, `coreneuron`, `all`}.
`--prefix`	current path	Path for downloading, compiling, installing simulation engines. Also used to store inputs and outputs from benchmarks and validation tests. Can be either a relative or absolute path.
`--env`	none	Optional script for configuring the environment and build steps. See *Directory Structure*.

Directory Structure

The following directory structure will be generated when `install-local.sh` is run:



```
prefix
├── build
│   ├── arbor
│   ├── coreneuron
│   ├── neuron
│   └── ...
├── install
│   ├── bin
│   ├── include
│   ├── lib
│   └── share
├── config
├── input
│   └── benchmarks
├── output
│   ├── benchmarks
│   └── validation
└── cache
```

If no prefix is provided, the directory structure is created in the nsuite path. The contents of each sub-directory are summarised:

build	Source code for simulation engines is checked out, and compiled here.
install	Installation target for the simulation engine libraries, executables, headers, etc.
config	The environment used to build each simulation engine is stored here, to load per-simulator when running benchmarks and validation tests.
cache	Validation data sets are stored here when generated during the installation phase.
input	**generated by running benchmarks** Input files for benchmark runs in sub-directories for each benchmark configuration.
output	**generated by running benchmarks/validation** Benchmark and validation outputs in sub-directories for each benchmark/validation configuration.

Customizing the environment

NSuite attempts to detect features of the environment that will influence how simulation engines are compiled and run, including compilers, MPI support and CPU core counts. HPC systems have multiple compilers, MPI implementations and hardware resources available, which are typically configured using modules. It isn't possible for NSuite to detect which options to choose on such systems, so user can customise the compilation and execution of simulation engines. To do this, a user provides an *environment configuration script* that will sourced after NSuite has performed automatic environment detection and configuration.

The script is specified with the `--env` flag:

```
./install-local arbor  --env=arbor-config.sh
./install-local neuron --env=neuron-config.sh
```

In the example above, different configurations are used for Arbor and NEURON. This can be used, for example, to choose compilers that produce optimal results on each respective simulator, or when different simulators require different versions of a library.

Examples of scripts for two HPC systems, Piz Daint and JUWELS, can be found in the `scripts` sub-directory in NSuite.

General Variables

The following variables are universal to all of the simulation engines.

Variable	Default value	Explanation
ns_cc	mpicc if available, else gcc/clang on Linux/OS X	The C compiler for compiling simulation engines.
ns_cxx	mpicxx if available, else g++/clang++ on Linux/OS X	The C++ compiler for compiling simulation engines.
ns_with_mpi	ON iff MPI is detected	ON/OFF to compile simulation engines with MPI enabled. Also controls whether mpirun is used to launch benchmarks.
ns_makej	4	Number of parallel jobs to use when compiling.
ns_python	which python3	The Python interpreter to use. Must be Python 3.
ns_threads_per_core	automatic	The number of threads per core for parallel benchmarks.
ns_cores_per_socket	automatic	The number of cores per socket for parallel benchmarks.
ns_sockets	1	The number of sockets for parallel benchmarks. One MPI rank is used per socket if MPI support is enabled.
run_with_mpi	Bash function for OpenMPI	A bash function for launching an executable and flags with multi-threading and optionally MPI, based on the ns_threads_per_core, ns_cores_per_socket, ns_sockets variables.

Simulator-Specific Variables

There are Arbor-specific options for checking out Arbor from a Git repository, and for configuring target-specific optimizations.

Variable	Default value	Explanation
ns_arb_git_repo	https://github.com/arbor-sim/arbor.git	URL or directory for the Git repository to check out Arbor source from.
ns_arb_branch	v0.5	The branch/tag/SHA to check out. Master will be used if empty.
ns_arb_arch	native	The CPU architecture target for Arbor. Must be set when cross compiling. Default native targets the architecture used to configure NSuite.
ns_arb_gpu	none	Build Arbor with/without GPU support. Available options are none, cuda, clang, hip-clang
ns_arb_vectorize	ON	Whether to use explicit vectorization for Arbor.

The NEURON-specific options are for configuring where to get NEURON's source from. NEURON can be downloaded from a tar ball for a specific version, or cloned from a Git repository.

The official versions of NEURON's source code available to download are inconsistently packaged, so it is not possible to automatically determine how to download and install from a version string alone, e.g. "7.6.2". This is why three variables must be set if downloading a NEURON tarball.

Variable	Default value	Explanation
ns_nrn_tarball	nrn-7.6.5.tar.gz	The name of the tar ball file (caution: not named consistently between versions).
ns_nrn_url	https://neuron.yale.edu/ftp/neuron/ versions/v7.6/7.6.5/ ${ns_nrn_tarball}	The URL of the tar ball (caution: not name consistently between versions).
ns_nrn_path	nrn-7.6	The name of the path after expanding the tar ball (caution: not name consistently between versions).
ns_nrn_git_repo	empty	URL or path of Git repository. If set it will be used instead of downloading a tarball.
ns_nrn_branch	master	Branch or commit SHA to use if sourcing from Git.

CoreNEURON has more support than NEURON for targeting different hardware, either via automatic vectorization, or using OpenACC for GPUs. However, it is quite difficult to build, particularly as part of an automated pipeline: users have to directly provide architecture- and compiler-specific flags to CMake. As soon as we are able to build CoreNEURON this way ourselves, we will add more flags for targeting different architectures.

Variable	Default value	Explanation
ns_cnrn_git_repo	https://github.com/BlueBrain/CoreNeuron.git	URL or path of Git repository.
ns_cnrn_sha	0.14	Branch, tag or commit SHA of Git repository.

Example custom environment

Below is a custom configuration script for a Cray cluster with Intel KNL processors. It configures all platform-specific details that can't be automatically detected by

- loading and swapping required modules;
- setting a platform-specific magic variable CRAYPE_LINK_TYPE required to make CMake play nice;
- configuring MPI with the Cray MPI wrapper;
- configuring Arbor to compile with KNL support;
- configuring the number of threads and MPI ranks with which to run benchmarks.

```
# set up Cray Programming environment to use GNU toolchain
[ "$PE_ENV" = "CRAY" ] && module swap PrgEnv-cray PrgEnv-gnu

# load python, gcc version and CMake
module load cray-python/3.6.5.1
module swap gcc/7.3.0    # load after cray-python
module load CMake

# set for CMake to correctly configure Arbor and CoreNEURON
export CRAYPE_LINK_TYPE=dynamic

# Python, MPI and build options for this system
ns_python=$(which python3)
ns_cc=$(which cc)
```

(continues on next page)

(continued from previous page)

```
ns_cxx=$(which CC)
ns_with_mpi=ON
ns_makej=20

# simulator-specific options
ns_arb_arch=knl

# cluster-specific options
ns_threads_per_core=1
ns_cores_per_socket=64
ns_sockets=1
ns_threads_per_socket=64

run_with_mpi() {
    # this system uses Slurm's srun to launch MPI jobs on compute nodes
    srun -n $ns_sockets -c $ns_threads_per_socket $*
}
```

3.2 Running NSuite

The second stage of the NSuite workflow is running benchmark and validation tests on simulation engines that were installed in the first stage when *Installing NSuite*.

Benchmarks and validation tests are launched with the respective scripts `run-bench.sh` and `run-validation.sh`.

In the example workflow below, NEURON and CoreNEURON are first installed in a path called `nrn` using a user-specified environment configuration `neuron-config.sh`, then benchmark and validation tests are run on the installed engines.

```
# download and install NEURON and CoreNEURON in a directory called nrn
./install-local.sh neuron coreneuron --prefix=nrn --env=neuron-config.sh

# run default benchmarks for NEURON and CoreNEURON
./run-bench.sh neuron coreneuron --prefix=nrn

# run validation tests for NEURON
./run-validation.sh neuron --prefix=nrn
```

The benchmark and validation runners take as arguments the simulators to test, and the *prefix* where the simulation engines were installed.

Note: The environment does not have to be specified by the user using the `--env` flag, because the environment used to configure and build each simulation engine is saved during the installation with `install-local.sh`, and automatically loaded for each simulation engine by the runners.

Flags and options for benchmark and validation runners are described in detail below.

3.2.1 Benchmarks

The full set of command line arguments for the benchmark runner `run-bench.sh` are:

Flag	Default value	Explanation
--help		Display help message.
simulator	none	Which simulation engines to benchmark. Any number of the following: {arbor, neuron, coreneuron}.
--prefix	current path	Path where simulation engines to benchmark were installed by install-local.sh. All benchmark inputs and outputs will be saved here. Can be either a relative or absolute path.
--model	ring	A list of benchmark models to run. At least one of {ring, kway}.
--config	small	A list of configurations to run for each benchmark model. At least one of {small, medium, large}.
--output	'%m/%p/%s'	Override default path to benchmark outputs. The provided path name will be appended to prefix. Use --help for all format string options.

The --model and --config flags specify which benchmarks to run and how they should be configured. Currently there are two benchmark models, *ring* and *kway*; detailed descriptions are in *Benchmarks*.

```
# run default benchmarks with Arbor
./run-bench.sh arbor

# run ring and kway benchmarks with Arbor
./run-bench.sh arbor --model='ring kway'

# run kway benchmark in medium and large configuration with Arbor
./run-bench.sh arbor --model=kway --config='medium large'
```

Each benchmark model has three configurations to choose from: small, medium and large. The configurations can be used to test simulation engine performance at different scales. For example, the *small* configuration has fewer cells with with simpler morphologies than the *medium* and *large* configurations. The *small* configuration requires little time to run, and is useful for modelling performance characteristics of simpler models. Likewise, models in *large* configuration take much longer to run, with considerably more parallel work for benchmarking performance of large models on powerful HPC nodes.

Note: NEURON is used to generate input models for CoreNEURON. Before running a benchmark in CoreNEURON, the benchmark must first be run in NEURON.

Benchmark output

Two forms of output are generated when a benchmark case is run. The first is a summary table printed to standard output, and the second is a CSV file that can be saved for use by tools later analysis of benchmark output. In the example below the *kway* model is run in the *small* configuration for Arbor and NEURON.

```
./run-bench.sh arbor neuron --model=kway --config=small --prefix=install
==  platform:          linux
==  cores per socket:  4
==  threads per core:  1
==  threads:           4
==  sockets:           1
==  mpi:               ON

==  benchmark: arbor kway-small
```

(continues on next page)

(continued from previous page)

```
  cells compartments    wall(s)  throughput  mem-tot(MB) mem-percell(MB)
      2           90      0.041        48.8        0.318       0.159
      4          184      0.038       105.3        0.529       0.132
      8          368      0.039       205.1        0.822       0.103
     16          736      0.058       275.9        1.449       0.091
     32         1462      0.106       301.9        2.642       0.083
     64         2882      0.206       310.7        5.010       0.078
    128         5778      0.406       315.3        9.517       0.074
    256        11516      0.802       319.2       18.705       0.073

==  benchmark: neuron kway-small
  cells compartments    wall(s)  throughput  mem-tot(MB) mem-percell(MB)
      2           84      0.174        11.5            -           -
      4          172      0.179        22.4            -           -
      8          348      0.342        23.4            -           -
     16          688      0.711        22.5            -           -
     32         1384      1.380        23.2            -           -
     64         2792      3.600        17.8            -           -
    128         5596     14.049         9.1            -           -
    256        11188     33.246         7.7            -           -
```

Benchmark output for each {simulator, model, config} tuple is stored in the output path `prefix/output/benchmarks/${output}`. By default `${output}` is, `model/config/simulator`, which can be overriden by the `--output` flag. For the example above, two output files are generated, one for each simulator:

`install/output/benchmark/kway/small/arbor/results.csv`

```
cells,    walltime,      memory,  ranks,threads,    gpu
    2,       0.041,       0.318,      1,       4,     no
    4,       0.038,       0.529,      1,       4,     no
    8,       0.039,       0.822,      1,       4,     no
   16,       0.058,       1.449,      1,       4,     no
   32,       0.106,       2.642,      1,       4,     no
   64,       0.206,       5.010,      1,       4,     no
  128,       0.406,       9.517,      1,       4,     no
  256,       0.802,      18.705,      1,       4,     no
```

`install/output/benchmark/kway/small/neuron/results.csv`

```
cells,    walltime,      memory,  ranks,threads,    gpu
    2,       0.174,            ,      1,       4,     no
    4,       0.179,            ,      1,       4,     no
    8,       0.342,            ,      1,       4,     no
   16,       0.711,            ,      1,       4,     no
   32,       1.380,            ,      1,       4,     no
   64,       3.600,            ,      1,       4,     no
  128,      14.049,            ,      1,       4,     no
  256,      33.246,            ,      1,       4,     no
```

Descriptions and units for each column are tabulated below.

Column	Units	Explanation
cells	•	Total number of cells in the model.
walltime	seconds	Time taken to run the simulation. Does not include model building or teardown times.
memory	megabytes	Total memory allocated during model building and simulation. Measured as the difference in total memory allocated between just after MPI is initialized and the simulation finishing.
ranks	•	The number of MPI ranks.
threads	•	Number of threads per MPI rank.
gpu	•	If a GPU was used. One of yes/no.

3.2.2 Validation Tests

Validation tests are composed of a model, corresponding to a physical system to be simulated, and a parameter set, which specifies parameters within that system.

The *run-validation.sh* script runs all or a subset of the models for one or more installed simulators, saving test artefacts in a configurable output directory and a presenting pass/fail status for each test on standard output.

Requirements

The existing validation scripts use functionality from the `scipy` and `xarray` Python modules. These modules need to be available in the Python module search path.

Invocation

```
run-validation.sh [OPTIONS] SIMULATOR[:TAG ...] [SIMULATOR...]
```

`SIMULATOR` can be any of the simulators installed with *install-local.sh*. By default, *run-validation.sh* will use the current directory as the installation and output base directory. If no models are explicitly selected with the `--model` option (see below), all models and parameter sets will be run against each specified simulator.

`SIMULATOR` can optionally have a sequence of _tags_ appended, which are keywords specific to simulator implementations of validation models that change the global behaviour of that simulator. For any given simulator, the set of supported tags may differ from model to model. See the `README.md` file in each validation model directory for information regarding supported tags.

Options are as follows:

Option	Explanation
-h, --help	Display help message and exit.
-l, --list-models	List all available model/parameter sets.
--prefix=PREFIX	Base directory for local installation and output directories. Validation tests may also create reference datasets in PREFIX/cache.
-m, --model=MODEL[/PARAM]	A model or model/parameter set to run. MODEL alone is equivalent to MODEL/default.
-r, --refresh	Regenerate any required cached reference data sets.
-o, --output=FORMAT	Substitute fields in FORMAT and use the resulting absolute or relative path for the validation test output directory. Relative paths are with respect to PREFIX/output/validation.

By default, the outputs for a validation test run are stored in PREFIX/output/validation/SIMULATOR/MODEL/PARAM, corresponding to an output format of %s/%m/%p. Fields in the FORMAT string are substituted as follows:

%T	Timestamp of invocation of install-local.sh (ISO 8601/RFC 3339)
%H	NSuite git commit hash (with + suffix if modified)
%h	NSuite git commit short hash (with + suffix if modified)
%S	System name (if defined in system environment script) or host name
%s	Simulator name (with tags, if any)
%m	Model name
%p	Parameter set name
%%	Literal '%'

Output

run-validation.sh will print pass/fail information to stdout, but will also record information in the per-test output directories:

File	Content
run.out	Captured standard output from test script
run.err	Captured standard error from test script
status	Pass/fail status (see below)

The status is one of:

1. pass — validation test succeeded.
2. fail — validation test failed.
3. missing — no implementation for the validation test found for requested simulator.
4. error — an error occurred during validation test execution.

The output directory may contain other test artefacts. By convention only, these may include:

File	Content
run.nc	Numerical results from simulator run in NetCDF4 format.
delta.nc	Computed differences from reference data.

3.3 Benchmarks

3.3.1 Architecture

Benchmarks are set up in the NSuite source tree according to a specific layout. Different benchmarks models can share an underlying benchmark. For example, the *ring* and *kway* benchmarks are different configurations of what we call a *busy-ring* model. In this case, the *busy-ring* is called a benchmark *ENGINE* and *kway* is a benchmark *MODEL*. All scripts and inputs for *ENGINE* are in the path `benchmarks/engines/ENGINE`, and inputs for a *MODEL* are in `benchmarks/models/MODEL`.

Every model *MODEL* must provide a configuration script `benchmarks/models/MODEL/config.sh` that takes the following arguments:

```
config.sh $model                \ # model name
          $config               \ # configuration name
          $ns_base_path         \ # the base path of nsuite
          $ns_config_path       \ # path to config directory
          $ns_bench_input_path  \ # path to benchmark input base directory
          $ns_bench_output      \ # path to benchmark output base directory
          $output_format          # format string for simulator+model+config
```

The script will in turn generate a benchmark runner for each simulation engine:

1. `$ns_bench_input_path/$model/$config/run_arb.sh`
2. `$ns_bench_input_path/$model/$config/run_nrn.sh`
3. `$ns_bench_input_path/$model/$config/run_corenrn.sh`

These scripts should generate benchmark output in the per-simulator path `$ns_bench_output/$output_format` where the `$output_format` defaults to `$model/$config/$engine`.

Note: NSuite does not specify how the contents of `benchmarks/engines/ENGINE` have to be laid out.

Performance reporting

Each benchmark run has to report metrics such as simulation time, memory consumption, the number of cells in model, and so on. These are output in the formats described in *Benchmark output*.

Arbor has a standardised way of measuring and reporting metrics using what it calls *meters*. NSuite provides a Python module in `common/python/metering.py` that offers the same functionality in Python, which can be used for the NEURON benchmarks.

With this standard output format, the `scrpts/csv_bench.sh` script can be used to automatically generate the CSV output.

3.4 Validation

A validation test runs a particular model, representing some physical system to simulate, against one or more sets of parameters and compares the output to a reference solution. If the output deviates from the reference by more than a given threshold, the respective test is marked as failed for that simulator.

Simulator output for each model and parameter set is by convention stored in NetCDF format, where it can be analysed with generic tools.

3.4.1 Architecture

Validation models are set up in the NSuite source tree according to a specific layout.

Data and scripts required to run a particular validation model *MODEL* will all be found under in the `validation/MODEL` directory. At minimum, there must be an executable run script called `run` (see below) and a default parameter set `default.param`. Any additional parameter sets must have a `.param` suffix.

The interpretation of a parameter set file is particular to any given model, but by convention, and for compatibility with the existing run scripts, they should comprise a sequence of `key=value` assignments, one per line, with `key` being a string without any whitespace and `value` a (possibly fractional) decimal number.

Model run scripts

A run script is invoked with the following arguments:

1. The output directory.
2. The simulator name (with tags).
3. The parameter set name.

The script should run the implementation of the model for the simulator, if it exists, with the parameters described in the corresponding parameter set file.

The simulator name may have one or more tag suffixes, of the form `:tag` — these correspond to global flags applied to a simulator to modify its behaviour. It is hoped that any supported tags for a simulator have the same meaning across different models; `neuron:firstorder`, for example, should be interpreted uniformly as asking NEURON to run with its first order solver. This behaviour, however, is not enforced (see the implementation notes below).

The exit code determines the status of the test:

Exit code	Interpretation
0	Success
96	Test failure
97	Missing implementation
98	Unsupported tag
other	Execution error

Apart from cached reference data, any files created by the run script should be restricted to the output directory. As the files `run.out`, `run.err`, and `status` in the output directory are written by the `run-validation.sh` script, these files should not be written to by the run script itself.

Reference data generated by the run script can be stored in the output directory, or optionally in the NSuite cache directory. The cache directory is defined in the environment variable `ns_cache_path`; data for a particular model *MODEL* should be stored in a subdirectory of the cache directory also named *MODEL*.

If a validation run script does use cached data, that data should be regenerated if the environment variable `ns_cache_refresh` has a non-empty value.

3.4.2 Building tests

In order to generate reference data or to construct a simulator implementation of particular model, there may be a requirement to build some extra software at install time.

When `install-local.sh` is run, the directory `validation/src` is scanned for subdirectories containing a `CMakeLists.txt` file. These are then built with CMake unless there is a file named `BUILDFOR` in the subdirectory.

The BUILDFOR file, if present, contains a whitespace-separated list of relevant simulators; the project will only be built if the corresponding simulator has been installed in the invocation of install-local.sh.

3.4.3 Common tools

There is no requirement that validation tests use NetCDF as a format for simulator results and reference data, but there are two tools provided in common/bin, *viz.* comparex and thresholdx, that may simplify the creation of tests that do use NetCDF representations.

The comparex program compares variables across two different NetCDF files, producing deltas, absolute errors, and relative errors. It can optionally compare a variable against an interpolated reference variable and estimate a lower bound on the absolute and relative errors via a computed estimate of the interpolation error.

The thresholdx program applies a sequence of simple predicates of the form *variable* op *value* to the data in a NetCDF file, where op is one of =, <, >, <=, >=. It prints the predicate and a pass or fail message, and exits with a non-zero value if any of the predicates failed.

3.4.4 NetCDF conventions

If NetCDF is used as the output representation for simulation results, it is strongly recommended that the follow convention be followed:

- All *key* = *value* settings in a model paramset should be recorded as scalar global attributes of type NC_DOUBLE.
- In addition, there should be global string attributes simulator, simulator_build, and validation_model:
  - simulator should be set to the name of the simulator followed by any tags, separated by ':' and in alphabetical order, e.g. neuron:firstorder.
  - simulator_build should contain version information for the simulator used to produce the output, ideally with sufficient detail to be able to recreate the simulator binary.
  - validation_model should contain the name of the model, but not include the name of the parameter set, e.g. rallpack1, not rallpack1/default.
- Variables should have a units attribute describing the units for the data in SI using standard abbreviations compatible with UDUNITS and as much as feasible, other unit parsing libraries.
- If other metadata is provided, it should broadly follow common NetCDF conventions such as those described by the CF conventions.

3.4.5 Implementation notes

The existing run scripts use a helper script scripts/model_common.sh to assist in marshalling parameters and invoking particular model implementations; please refer to the comments in this script for details. For a simulator SIM, the run scripts then look for an implementation-specific script called run-SIM, which expects command line arguments of the form:

```
run-SIM -o OUTPUT [--tag TAG]... [KEY=VALUE ...]
```

The simulator-specific run script is responsible for returning the unsupported tag error code if it does not support a requested tag.

A python helper module nsuite.stdarg is intended to make parsing these options more straightforward. Similarly, the C++ header validation/src/include/common_args.h is used for the Arbor model implementations.

As much as is feasible, it is recommended that model implementations for a given simulator support the same set of tags. Tags used in current implementations include:

- Arbor:
  - `binevents`: bin event delivery times to simulation dt. Default behaviour is to use precise event times, without any binning.
- NEURON and CoreNEURON:
  - `firstorder`: use the first order, fixed time step integrator. Default behaviour is to use the second order fixed time step integrator.

3.5 Simulation Engines

A simulation engine is a library or application for simulating multi-compartment neural network models. NSuite supports three simulation engines: Arbor, NEURON and CoreNEURON.

Table 1: Default versions of each supported simulation engine

Engine	Version	Kind	Source
Arbor	0.2	git tag	GitHub arbor-sim/arbor
NEURON	7.6.5	tar ball	FTP neuron.yale.edu
CoreNEURON	0.14	git tag	GitHub BlueBrain/CoreNeuron

Each benchmark and validation test is implemented for each engine that has the features required to run the test.

3.5.1 Required features

For a simulation engine to run at least one of the benchmark and validation tests, it must support the following features:

- **[required]** Support for compilation and running on Linux or OS X.
- **[required]** Support for arbitrary cell morphologies
- **[required]** Common ion channel types, specifically passive and Hodgkin-Huxely.
- **[required]** Support for user defined network connectivity.
- **[required]** Synapses with exponential decay, i.e. the expsyn and exp2syn synapse dynamics as defined in NEURON.
- Output of voltage traces at user-defined locations and time points.
- Output of gid and times for spikes.

Note: If a simulation engine doesn't support a feature required to run a test, the test will be skipped. For example, the only simulation output provided by CoreNEURON is spike times, so validation tests that require other information such as voltage traces are skipped when testing CoreNEURON.

NSuite does not describe models using universal model descriptions such as SONATA or NeuroML. Instead, benchmark and validation models are described using simulation engine-specific descriptions.

Arbor models

Models for Arbor are described using Arbor's C++ API, and as such, they need to be compiled before they can be run. Compilation of each model is performed during the installation phase, see *Installing NSuite*.

NEURON models

Models to run in NEURON are described using NEURON's Python interface. The benchmarking and validation runners launch the models using with the Python 3 interpreter specified by the `ns_python` variable (see *General Variables*).

CoreNEURON models

NEURON is required to build models used as input for CoreNEURON. There are two possible workflows for this:

1. Build a model in NEURON, write it to file, then load and run the model using the stand-alone CoreNEURON executable.
2. Build a model in NEURON, then run the model using CoreNEURON inside NEURON.

Benchmark models are run using the first approach, to minimise memory overheads and best reflect what we believe will be the most efficient way to use CoreNEURON for HPC.

The second approach is used for validation tests, which run small models with low overheads, to simplify the validation workflow by not requiring execution of separate NEURON and CoreNEURON scripts and applications for a single model.

For more information about the different ways to run CoreNEURON, see the CoreNEURON documentation.

3.5.2 Adding a simulation engine

Support for a new simulation engine can be added using the steps described below. All of the steps are implemented in bash scripts, and can be done by using the scripts for Arbor, NEURON and CoreNEURON as templates.

Write installation script

Write an installation script that is responsible for:

- Downloading/checking out the code;
- Compiling and installing the library/application;
- Compiling benchmark and validation code if required.

The following scripts can be used as templates.

- Arbor: `scripts/build_arbor.sh`
- NEURON: `scripts/build_neuron.sh`
- CoreNEURON: `scripts/build_coreneuron.sh`

Add simulator-specific variables

Each simulation engine has unique options specific to that engine, for example:

- Arbor can specify which CPU architecture to target.
- Arbor can optionally be built with GPU support.
- NEURON requires parameters that describe how to download official release tar balls.

These options are configured using variables with prefixes of the form `ns_{sim}_{feature}`, for example `ns_arb_arch` and `ns_nrn_tarball`. You can define variables as needed, and configure their default value, in `scripts/environment.sh`, in the `default_environment` function.

Add engine to `install-local.sh`

The `install-local.sh` script has to be extended to support optional installation of the new simulation engine. Follow the steps used by the existing simulation engines.

Note: If the simulation engine requires separate compilation of individual benchmark and validation models, follow the example of how Arbor performs this step in `scripts/build_arbor.sh`.

Implement benchmarks and validation tests

See *Benchmarks* and *Validation* pages for details on how to add benchmark and validation tests.