A testsuite for a neural simulation engine

Jochen Martin Eppler[1,2], Rüdiger Kupper[1], Hans Ekkehard Plesser[3], Markus Diesmann[4]
eppler@biologie.uni-freiburg.de, ruediger.kupper@gmail.com, hans.ekkehard.plesser@umb.no, diesmann@brain.riken.jp

1
Honda Research Institute Europe
Carl-Legien-Str. 30
63073 Offenbach/Main, Germany
http://www.honda-ri.de

2
Bernstein Center for Computational Neuroscience
Hansastraße 9a
79104 Freiburg, Germany
http://www.bccn.uni-freiburg.de

3
Dept. of Mathematical Sciences and Technology
Norwegian University of Life Sciences
1432 Ås, Norway
http://www.umb.no

4
RIKEN Brain Science Institute &
Computational Science Research Program
Wako City
351-0198 Saitama, Japan
http://www.brain.riken.jp

- NEST [1, 2] is a simulator for large networks of spiking neurons that runs on workstation computers and clusters.
- We present experience from the testsuite of NEST, a framework for systematic unit tests. Its key features are
  - Hierarchical tests: Basic features are tested before high-level features
  - Tests for the parallel and distributed execution of NEST
  - A unit-test library that eases the task of writing tests for NEST
- In a companion contribution, we give a general demonstration of NEST during the demo-session (D11).

Introduction

In Computer Science, testing is a standard activity of the software development process [3, 4]. However, little research has been carried out on the specific problems of testing neuronal simulation engines. We provide insight into our experience from building a comprehensive testsuite for NEST.

We found that with the growing complexity of the software and the growing number of developers, formalized and systematic testing becomes critical. The rapid growth of neuroscience knowledge and the changing research directions require an incremental/iterative development style [5] which extends over the full life time of the product. Thus there is no single testing phase but the same tests need to be carried out repetitively over a time span of many years.

The testsuite consists of a set of small unit tests and a shell script to execute the scripts in a hierarchical order from simple to complex:

1. Self-tests: test the ability to report errors
2. Test that objects have correct default values and accept parameter changes
3. Compare simulation results with analytical results for simple scenarios
4. Check correctness of results with simpler algorithms
5. Test the convergence of results with decreasing simulation time step
6. Check for expected accuracy
7. Test the invariance of results with increasing numbers of processors
8. Test the higher-level user interface functions
9. Create regression tests for fixed problems

The testsuite should be run after compilation and installation by typing `make installcheck` in the build directory or using the command `test` in NEST.

NEST's unittest library

NEST provides a collection of functions to ease the task of writing consistent unit tests in compact and human readable form. All of these functions test a certain criterion and quit NEST with an exit code not equal to 0 in case the criterion is not fulfilled. As usual in shell programming, an exit code of 0 is interpreted as success. The most important functions in the unittest library are:

`assert_or_die` - Check the given condition and quit with exit code 1 if it fails or with exit code 2 if it raises an error.

`pass_or_die` - Execute a code block and quit with exit code 2 if it raises an error.

`fail_or_die` - Execute a code block and quit with exit code 3 if it does *not* raise an error.

`failbutnocrash_or_die` - Execute a code block and quit with exit code 3 if it does *not* raise a scripterror. To make the function robust against crashes of NEST (like e.g. segmentation faults or failed C assertions) the code block is executed in a new instance of NEST.

`crash_or_die` - Execute a code block and quit with exit code 3 if NEST does *not* crash. This also executes the code block in a new NEST instance, but expects it to crash (e.g. segmentation fault or failed C assertion).

`distributed_assert_or_die` - Checks whether the given code block is independent of the number of processes. This function reruns the code block with different numbers of MPI processes and compares the result. The equality of the results is tested with the serial version of `assert_or_die`.

`ToUnitTestPrecision` - Reduce the argument to the given precision. This function is needed to convert different floating point numbers to a the same precision and thus allow to compare them.

`InflateUnitTestData` - Reformat compressed reference data. Reference data (e.g. analytical results) is stored in the test script in a human readable form and to allow to comment the data appropriately. To compare the reference data to the results from the simulation, this function reformats the data into a machine readable format.

Comparison with analytical results

For many neuron models, analytical solutions for basic properties exist. Examples are the time to threshold crossing, given a specific input, or the correct reset of the membrane potential. The analytical solutions are stated in the testscript and compared to the results from the simulation. An error is issued if the analytical and simulated results disagree.

Testing the convergence of a simulation result with decreasing simulation step size is a powerful method. However, the simulation may still converge to the wrong value or converge unexpectedly slow because of errors. Thus, comparison with analytical results is an invaluable additional tool.

An important requirement for this method of testing is the availability of a high-level math programming language to state expressions in readable form. To this end, SLI has been extended with a parser for standard infix math notation.

Parallel and distributed tests

NEST runs on a range of architectures, from ordinary desktop computers to large computer clusters with thousands of processor cores. Thus, testing threaded and distributed simulation is crucial.

The goal is to test the invariance of simulation results with respect to the number of jobs. This is important, because a simulation in NEST should not be dependent on the details of parallelization.

Because the `mpirun` command differs in different MPI implementations, the user defines the SLI function `mpirun` in NEST's configuration file `~/.nestrc`, which tells NEST how to run distributed simulations.

The shell script `nest_serial` is used to run serial test scripts, while the script `nest_indirect` is used to re-spawn a distributed version of NEST with a given number of processes.

PyNEST tests

PyNEST [6] is the new user interface of NEST. It provides Python functions that wrap the basic NEST functions for the creation, connection and manipulation of neurons and devices. In order to verify the correctness of the wrapper functions of the PyNEST high-level API, we have created test scripts for them based on the unittest library (`http://docs.python.org/library/unittest.html`) for Python. These scripts are run by the top-level testsuite script as part of the normal testing procedure.

Current status and outlook

The testsuite currently contains 21 self tests for basic functionality, 67 unit tests, 13 regression tests, 5 tests for parallel and distributed simulations, and 42 tests for the PyNEST high-level API. It also contains 47 manual tests, which are more complex and require manual judging of correctness.

We currently plan two main extensions to the testsuite framework:

The first is full support for *test driven programming*. This means that the test scripts for a new feature are written prior to the actual code. In an iterative way, this ensures that the finished code fulfills all requirements that were specified.

The second extension is the automatic execution of the examples in the documentation. The user documentation already contains a lot of examples for the correct usage of a function and provides an easy way to state invariants for functions directly, instead of writing separate tests.

References

[1] M.-O. Gewaltig and M. Diesmann (2007). NEST (Neural Simulation Tool). Scholarpedia 2(4), 1430.

[2] H.E. Plesser et al (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. Euro-Par 2007: Volume 4641 of LNCS. doi:10.1007/978-3-540-74466-5

[3] I. Sommerville (2007) Software Engineering (8th edition). Addison-Wesley. ISBN: 978-0321313799

[4] K. Beck and C. Andres (2004). Extreme programming explained: Embrace change (2nd edition). Addison-Wesley. ISBN: 978-0321278654

[5] M. Diesmann and M.-O. Gewaltig (2002). NEST: An environment for neural systems simulation. Forschung und wissenschaftliches Rechnen, GWDG-Bericht. Pages 43-70. Ges. für Wiss. Datenverarbeitung, Goettingen

[6] J.M. Eppler et al (2009). PyNEST: A convenient interface to the NEST simulator. Frontiers in Neuroinformatics. 2:12. doi:10.3389/neuro.11.012.2008

NEST can be downloaded from the homepage of the NEST Initiative at
`www.nest-initiative.org`