Architectures for communication between processes and software layers for a simulator for biological neural networks

---

Dissertation zur Erlangung des Doktorgrades
Vorgelegt von Jochen Martin Eppler

Albert-Ludwigs-Universität Freiburg
Technische Fakultät

Dekan der Fakultät: Prof. Dr. Hans Zappe

Erster Gutachter: Prof. Dr. Gerhard Schneider

Zweiter Gutachter: Prof. Dr. Markus Diesmann

Dritter Gutachter: Prof. Dr. Rolf Backofen

Datum der Disputation: 16.12.2010

Contact information

Jochen Martin Eppler
Mail: eppler-biologie@mindzoo.de
Web: http://mindzoo.de

Honda Research Institute Europe GmbH
Carl-Legien-Str. 30
63073 Offenbach am Main
Germany

Bernstein Center for Computational Neuroscience
Hansastraße 9a
79104 Freiburg
Germany

Acknowledgments

This thesis would have never been possible without the help and support of many people.

I would like to thank Gerhard Schneider for his continuous support and for his encouragement during the formally complex process of preparing this thesis. For the excellent supervision through the last five years I especially want to thank Marc-Oliver Gewaltig and Markus Diesmann.

I am grateful to Hans Ekkehard Plesser, who came up with a lot of crazy, cool, and helpful ideas, some of which actually solved problems! I thank Moritz Helias, Eilif Muller and Abigail Morrison for extensive and funny programming sessions. Without you, the source code of NEST would look much more boring! Furthermore I thank all the members of the NEST Initiative for their supportive comments and discussions.

I want to thank Nils Einecke, Andreas Knoblauch, Rüdiger Kupper, Sven Rebhan, Sven Schrader, and all the other people at the Honda Research Institute Europe in Offenbach for the numerous debates, coffee sessions, and enlightenment in the dark valleys of programming and debugging.

For the strong support and the nice atmosphere during the time of my studies, especially in the hot phase at the end of this thesis, I want to thank Susanne Kunkel, Bernd Wiebelt, and the other people at the Bernstein Center for Computational Neuroscience in Freiburg. You have always been a source of inspiration and provided me with the necessary distraction.

Big thank goes to my parents for their care and backing that made my studies possible in the first place. A very warm hug goes to all of my friends, especially to Timo Bonaffini and Florian Benjamin Schock, who stabilized my life even in the oddest moments and gave me the incentive to carry on. You guys simply rock!

Finally, I would like to express my deep affection for Elena Wagner, and thank her for her patience. She was there for me without doubt, even though it was clear from the very beginning that I'm not the easiest guy on this planet :-)

This work was partially funded by DAAD/NFR 313-PPP-N4-lk, DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grant 15879 (FACETS), the Next-Generation Supercomputer Project of MEXT (Japan), and the Helmholtz Alliance on Systems Biology (Germany). My position was funded by the Honda Research Institute Europe GmbH.

Declaration of publication

This section contains a list of publications that originated from the work described in this thesis. For the convenience of the reader, the articles are contained in Appendix A in the original format, in which they were published. In addition, Appendix A contains a detailed summary of my contributions to the publications.

Chapter 2 lead to the following publications:

- Hans Ekkehard Plesser, **Jochen Martin Eppler**, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig (2007) Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.), Euro-Par 2007: Parallel Processing, Volume 4641 of Lecture Notes in Computer Science, Berlin, pp. 672–681. Springer-Verlag. doi:10.1007/978-3-540-74466-5_71.

  The work has been carried out equally by Hans Ekkehard Plesser and Jochen Martin Eppler. Abigail Morrison implemented the original version of the distributed simulation kernel (Paranel), Markus Diesmann and Marc-Oliver Gewaltig wrote the original versions of NEST and supervised the work.

  The original publication is contained in Appendix A.1

- **Jochen Martin Eppler**, Hans Ekkehard Plesser, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig, (2007) Multithreaded and Distributed Simulation of Large Biological Neuronal Networks. In F. Cappello, T. Herault, and J. J. Dongarra (Eds.), EuroPVM/MPI 2007: Recent advances in parallel virtual machine and message passing interface, Volume 4757 of Lecture Notes in Computer Science, Berlin, pp. 391-392. Springer-Verlag. doi:10.1007/978-3-540-75416-9_55.

  The work has been carried out equally by Jochen Martin Eppler and Hans Ekkehard Plesser. Abigail Morrison implemented the original version of the distributed simulation kernel (Paranel), Markus Diesmann and Marc-Oliver Gewaltig wrote the original versions of NEST and supervised the work.

  The original publication is contained in Appendix A.2

Chapter 3 lead to the following publications:

- **Jochen Martin Eppler**, Moritz Helias, Eilif Muller, Markus Diesmann, and Marc-Oliver Gewaltig (2009) PyNEST: A convenient interface to the NEST simulator. Frontiers in Neuroinformatics 2. doi:10.3389/neuro.11.012.2008.

  The work has been carried out equally by Jochen Martin Eppler and Moritz Helias. Eilif Muller implemented the first prototype of the Python bindings for NEST together with Markus Diesmann and Marc-Oliver Gewaltig, who also supervised the work.

  The original publication is contained in Appendix A.3

- **Jochen Martin Eppler** (2009) PyNEST: A convenient interface to NEST. The Neuromorphic Engineer. doi:10.2417/1200912.1703.

  The original publication is contained in Appendix A.4

Chapter 4 lead to the following publication:

- Andrew Davison, Daniel Brüderle, **Jochen Martin Eppler**, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurrent Perrinet, and Pierre Yger (2008) PyNN: A common interface for neuronal network simulators. Frontiers in Neuroinformatics 2. doi:10.3389/neuro.11.011.2008

  The work has been carried out mainly by Andrew Davison. Jochen Martin Eppler provided parts of the implementation of the PyNEST backend and contributed to the overall design of PyNN.

  The original publication is contained in Appendix A.5

Chapter 5 lead to the following publication:

- Mikael Djurfeldt, Johannes Hjorth, **Jochen Martin Eppler**, Niraj Dudani, Moritz Helias, Tobias C. Potjans, Upinder S. Bhalla, Markus Diesmann, Jeanette Hellgren Kotaleski, and Örjan Ekeberg (2009) Run-Time Interoperability between Neuronal Network Simulators based on the MUSIC Framework. Neuroinformatics (2010) 8:43–60. doi:10.1007/s12021-010-9064-z.

  The MUSIC interface for NEST was written by Jochen Martin Eppler together with Moritz Helias. Tobias Potjans carried out the benchmark simulations. The MUSIC interface for MOOSE was written by Johannes Hjorth, Niraj Dudani, and Upinder S. Bhalla. The MUSIC library was implemented by Mikael Djurfeldt and Örjan Ekeberg. Mikael Djurfeldt, Örjan Ekeberg, Jeanette Hellgren, Upinder S. Bhalla, and Markus Diesmann supervised the work.

  The original publication is contained in Appendix A.6

Zusammenfassung

Die Simulation biologischer neuronaler Systeme entwickelt sich immer mehr zu einem Grundpfeiler der modernen Neurobiologie. Dies hat vor allem zwei Gründe: Zum einen sind Simulationen ein wichtiges Werkzeug um die Flut anatomischer und physiologischer Daten integrieren und auf Konsistenz testen zu können, zum anderen können durch Simulationen Fragen geklärt werden, die sich experimentell oder mit analytischen Methoden nicht beantworten lassen.

Das Gehirn von Wirbeltieren ist jedoch eine höchst komplexe Struktur mit bis zu 10^{12} Nervenzellen, den sogenannten Neuronen. Jedes dieser Neuronen erhält Eingänge von ca. 10^4 Neuronen und produziert seinerseits Signale für ebensoviele Neuronen.

Die über die große Zahl von Verbindungen (Synapsen) vermittelte Wechselwirkung zwischen den Neuronen erfordert spezialisierte Simulationsprogramme. NEST ist ein Simulator für biologische neuronale Netze, der für die Simulation großer Netzwerke von sogenannten Punktneuronen optimiert ist. Er läuft auf einer Vielzahl von Architekturen, von normalen Desktop-Computern bis zu Supercomputern mit mehreren tausend Prozessoren. Diese Arbeit beschreibt vier wichtige Erweiterungen für NEST:

- Der Algorithmus zu Simulationssteuerung wurde optimiert, um sowohl threadbasierte als auch verteilte Simulationen zu erlauben. Die Kommunikation wurde auf Basis des MPI-Standards (Message Passing Interface Forum, 1994) implementiert und erweitert die bereits in Eppler (2006) vorgestellten Methoden. NEST wurde so angepaßt, dass Neuronen nicht nur innerhalb eines Prozesses miteinander kommunizieren können, sondern auch über Prozessgrenzen hinweg. Damit wird die Flexibilität von NEST mit der Performance des Pilotprojekts Paranel verbunden (Morrison et al., 2005).
- Es wurden Funktionen für die Kommunikation zwischen dem Simulationskern, dem Interpreter und einer neuen Benutzerschnittstelle als Modul für die Programmiersprache Python implementiert (PyNEST; Eppler et al., 2009). Im Gegensatz zum üblichen Ansatz, bei dem die Klassen und Funktionen direkt in Python zur Verfügung gestellt werden, bietet PyNEST eine minimale Schnittstelle zum Interpreter von NEST und erlaubt dadurch dessen vollständige Steuerung.
- Im Rahmen des EU Projektes FACETS haben wir an der Implementation und Konzeption einer Programmierschnittstelle mitgewirkt, die verschiedene Simulatoren steuert (PyNN; Davison et al., 2008) und so die einfache Übertragung von Simulationsspezifikationen auf andere Simulatoren erlaubt. Dies ist ein wichtiges Mittel um die Reproduzierbarkeit von Simulationen zu ermöglichen und die Unabhängigkeit der Modelle von ihrer Implementation zu gewährleisten.
- Um Simulationen unterschiedlicher Detailstufen zu verbinden wurde eine Anbindung an die MUSIC Bibliothek (Ekeberg & Djurfeldt, 2008) geschaffen. Dadurch kann NEST zur Laufzeit Daten mit anderen Simulatoren und Programmen austauschen. Dies ist ein wichtiger Schritt, um die Lücke zwischen verschiedenen Modellierungsansätzen zu überbrücken und einen effizienten Arbeitsablauf vom Zugriff auf eine Datenbank bis zur Visualisierung zu ermöglichen.

Die oben beschriebenen Änderungen wurden nach internen Tests direkt in die öffentliche Release von NEST übertragen und so den Benutzern zur Verfügung gestellt.

Contents

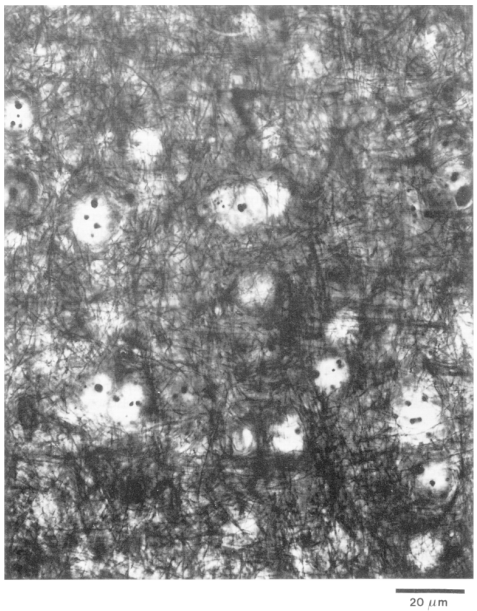
Figures

Chapter 1

Introduction

To understand how the brain stores and processes information is one of the biggest scientific challenges of our time. The complexity of this undertaking becomes evident if one looks at the number of elements involved: the human brain, for example, consists of up to 10^{12} nerve cells, each receiving input from approximately 10^4 other nerve cells and generating output to about as many. This means that already a single cubic millimeter of cortex contains at least 10^5 cells and about 10^9 connections (Figure 1.1; Braitenberg & Schüz, 1998). Simulations have become a valuable tool to understand the mechanisms behind the function of the brain.

(A)



(B)

Table 1.5.1. *Densities of neurons in the cortex (thousands per cubic millimeter)*

A		B		C	
Animal	Density	Region	Density	Layer	Density
Mouse	142.5	Visual	106	I	20
Rat	105.0	Somatosensory	60	II	82
Guinea pig	52.5	Auditory	43	III	62
Rabbit	43.8	Motor	30	IV	67
Cat	30.8			V	61
Dog	24.5			VI	77
Monkey	21.5				
Human	10.5				
Elephant	6.9				
Whale	6.8				

(C)

Table 1.5.4. *Typical compositions of cortical tissues*

Variable	Value
Neuronal density	40,000/mm^3
Neuronal composition:	
Pyramidal	75%
Smooth stellate	15%
Spiny stellate	10%
Synaptic density	$8 \cdot 10^9$/mm^3
Axonal length density	3,200 m/mm^3
Dendritic length density	400 m/mm^3
Synapses per neuron	20,000
Inhibitory synapses per neuron	2,000
Excitatory synapses from remote sources per neuron	9,000
Excitatory synapses from local sources per neuron	9,000
Dendritic length per neuron	10 mm

Figure 1.1: *Neural connectivity in the cortex (taken from Abeles, 1991): (A) Stained axons in the cortex. The empty spaces are due to blood vessels and cell bodies. (B) A: neuronal densities in the motor cortex in various animals; B: neuronal densities in various cortical regions in the human; C: neuronal densities in the various cortical layers in the visual cortex in the cat. (C) Typical composition of cortical tissues.*

1.1 Scientific context of this thesis

In 1891, Heinrich Wilhelm Gottfried von Waldeyer-Hartz formally proposed the *neuron doctrine*, which states that the nervous system is made up of discrete individual cells, called *neurons* (Barlow, 1972; Shepherd, 1991). Today, it is a commonly recognized fact that the interaction between neurons is the basis for brain function on all levels, starting with simple reflexes up to complex cognitive tasks such as planning, reasoning and consciousness.

The investigation of the nervous system on a molecular and cellular level in recent years lead to considerable insights into the function of the brain. Examples are the mechanisms that underlie plasticity and learning, attention, the function of the visual and auditory systems, motor control, and many more. It is also well known that malfunctions in the communication between neurons, the structure of the brain, and its bio-chemical processes can lead to severe disorders in higher brain function. Brain research made it possible to better understand the causes for Alzheimer's and Parkinson's disease, depression, epilepsy, deficits in auditory and visual processing, and other diseases. These insights allow to find remedies for the disorders. Recent examples are drugs for the treatment of depressions, neural prosthetics like cochlear implants, or devices for deep brain stimulation to break the synchronization, which causes the tremor in Parkinson's disease and some forms of epilepsy.

Classical neuroscience tries to gain understanding of the brain circuitry on the microscopic level by investigating the properties of single neurons and networks thereof in the intact brain of animals and humans (*in vivo*), and in preparations of brain slices and cell cultures (*in vitro*). Neuroscientists use a multitude of methods (e.g. microscopy, electrophysiology, and different imaging techniques) in their day-to-day work to measure the electrical and chemical properties and the connectivity of neurons in the brains of different species. These investigations provide important insights into the *structure* of the brain and into the function of single neurons and small populations of neurons. However, because of the complexity of the studied systems, they do not provide *functional* explanations for larger brain structures and it is often hard to identify the relevant circuits and connectivity patterns due to the high packing density of the tissue (see Figure 1.1).

Mathematical models of the nervous system and its parts are a valuable method for understanding the working principles in the brain and to integrate the huge amount of data produced by classical neuroscience. In *computational neuroscience*, researchers create and use such models to find hypotheses that connect the microscopic elements of the brain to functional units. These hypotheses can be tested analytically and, if too complex, in computer simulations, to understand how a specific structure creates a certain function. Researchers in different fields are working on different levels of detail, from the molecular level of ion channels and molecule dynamics over detailed morphological models of neurons to functional models of whole brain areas (Dayan & Abbott, 2001).

One branch of computational neuroscience is concerned with the application of computer and information science methods to problems in neuroscience. This branch is referred to as *neuroinformatics*. The research goals of neuroinformatics are databases for experiments and simulation, brain atlases, software for the analysis and visualization of neuroscientific data and the development of software tools to support neuroscience in general (see Figure 1.2). One part of neuroinformatics is dedicated to the development of methods and algorithms for the simulation of neural systems.

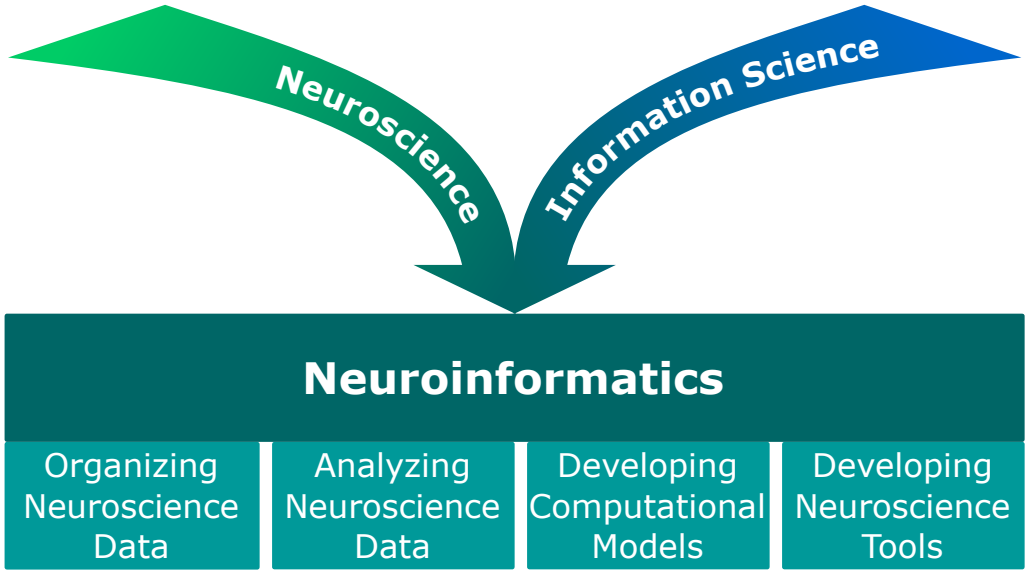


Figure 1.2: *Different branches of neuroinformatics (illustration by courtesy of Susanne Kunkel): The field of neuroinformatics is positioned a the intersection of neuroscience and information science. The lower part shows the different research areas.*

The simulation of neural networks is an ambitious task that requires sophisticated software. Many different attributes contribute to the success of a simulator. The most visible part of a simulator is the user interface. It should be convenient and easy to use, and allow a researcher to get started quickly by keeping the learning curve low. The simulator has to support the storage, update, and communication of a large number of neurons and connections in an efficient way. This can only be achieved by using efficient algorithms and data structures to represent and execute models. Moreover, it is important that the simulator works together nicely with other tools. This includes tools for stimulus generation, storing results in databases, analyzing and visualizing the data, and the communication with other simulators.

1.2 Aims of this thesis

The aim of this thesis is to present novel solutions for the problems connected to the simulation of biological neural networks, namely the algorithms and data structures for the communication inside the simulator, the communication between the user and the simulator, and for the interoperability between different software packages in the field of (computational) neuroscience. Studying the NEST simulation software (Gewaltig & Diesmann, 2007) and its requirements as an example, this thesis contributes to three main areas:

1. Technology for large-scale simulations of brain structures.
2. Reliable interfaces to set up and control such simulations.
3. Interoperability of simulation tools specialized at different levels of abstraction.

The communication architectures designed and developed in this thesis allow different processes, software layers, and applications to communicate efficiently with each other. This includes the data structures and algorithms for multi-threaded and distributed simulations of neural networks, the communication between new user interfaces of NEST and the simulation engine, and interfaces to share data with other applications at run-time.

1.3 The brain and its building blocks

The brain is the largest part of the central nervous system of all vertebrates and most invertebrates. It is located at the end of the spinal cord and consists of several regions (see Figure 1.3). These can be distinguished by their epigenetic origin and by their role in the information processing. The most remarkable region in mammals is the *cerebral cortex*, which is the place for the higher cognitive abilities that differentiate them from other vertebrates. The structure of the following paragraphs on brain regions is roughly based on the description in Kirsch (2010).

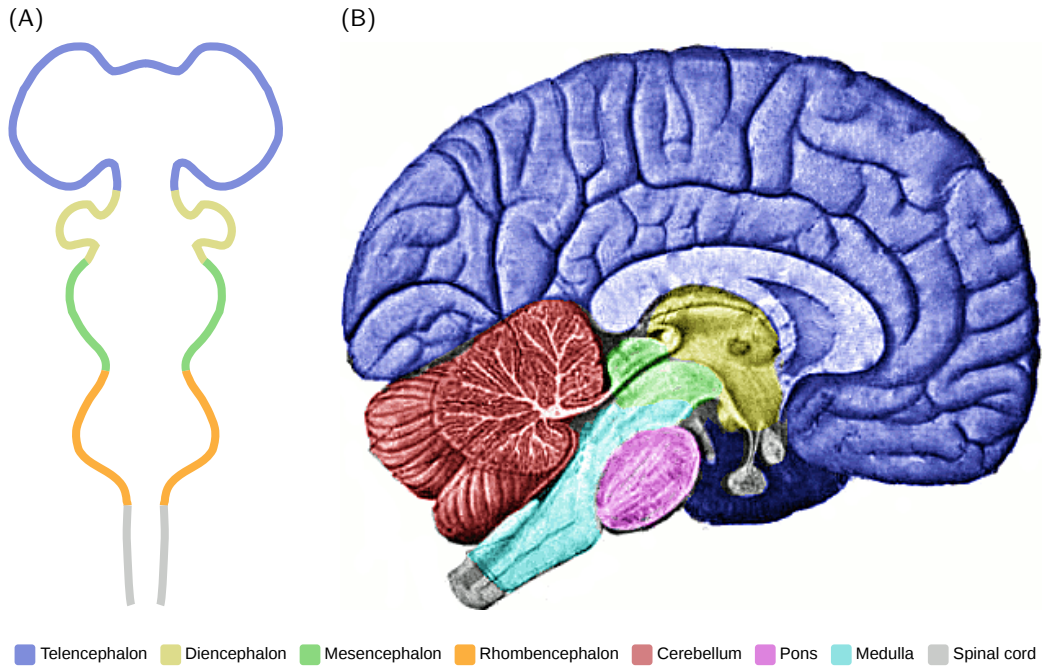


Figure 1.3: *Regions of the vertebrate brain: (A) Regions in the embryonic brain (modified from Surachit, 2007). The rhomencephalon differentiates into cerebellum, pons, and medulla during epigenesis. (B) Regions in the human brain (cross section; modified from Looie, 2008, based on Ranson, 1920).*

The *spinal cord* is the part of the central nervous system that extends from the basis of the skull to the first or second lumbar vertebra. The spinal cord receives sensory information from the skin, the joints, and the muscles of the body and contains motor neurons, which are responsible for willful movements as well as for reflexes. Additionally, it receives information from the inner organs and contains neurons that control numerous vegetative functions.

Following the spinal cord, the next three components of the central nervous system are the medulla oblongata, the pons and the mid brain, which form a continuous structure, called *brain stem*. It receives sensory information from the joints and skin of the head, the neck, and the face, and contains motor neurons that control the movement of head and neck. The brain stem also plays an important role in hearing, tasting, and in the sense of balance.

The *medulla oblongata* is an extension of the spinal cord, with which it has strong structural and functional similarities. Together with the pons, the medulla is responsible for the regulation of blood pressure and respiration.

The *pons* (latin for "bridge") contains a large number of neurons, which connect the information from both hemispheres of the end brain and the cerebellum.

The *cerebellum* has a strongly grooved surface and consists of several lobes, each of which has a specific function. The cerebellum receives sensory information from the spinal cord, motor information from the cortex, and information about balance from the vestibular system in the inner ear. The convergence of these inputs allows the cerebellum to coordinate the temporal sequence of movements, and plan the activation of skeletal muscles. In addition it plays a role in the coordination of head and eye movements.

The *mesencephalon* (*mid brain*) is the smallest part of the brain stem. Some of its regions play an important role in the direct control of eye movements, while others contribute to the control of skeletal muscles. The mid brain also constitutes an important relay station for auditory and visual signals.

Thalamus and *hypothalamus* form the *diencephalon* (*interbrain*). Its name originates from its position between mid brain and the two hemispheres of the end brain. The thalamus processes and relays most sensor and motor information that enters the cortex. In addition it is most likely responsible for the regulation of alertness and the emotional aspects of perception. The hypothalamus controls the autonomic nervous system, and, via the pituitary gland, the release of hormones. It has extensive connections to the thalamus, the mid brain, and to the regions of cortex, which receive information from the autonomic nervous system.

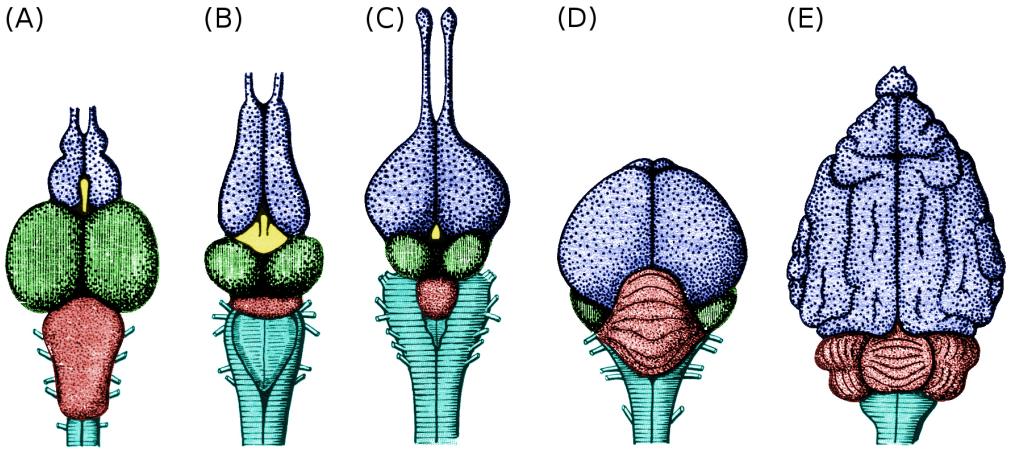


Figure 1.4: *Brains of different vertebrates, seen from above (modified from Kirsch, 2010): (A) Bony fish; (B) Amphibian (frog), (C) Reptile; (D) Bird (dove); (E) Mammal (dog). The colors denote the same regions as in Figure 1.3.*

The *telencephalon* (*end brain*) by far constitutes the largest region of the brain. It consists of the *cerebral cortex*, the white matter (mainly myelinated axons and glial cells), and three aggregations of nerve cells, called *nuclei* (basal ganglia, hippocampus, and amygdala). The term cerebral cortex literally means "brain rind" and characterizes the superficial layers of tissue of the two end brain hemispheres. In humans, the cortex is strongly grooved, which increases the surface, without increasing the volume of the brain. The telencephalon is responsible for willful movements, the analysis of sensory input, and plays a crucial role in all higher cognitive processes.

Although the brains of different vertebrates differ considerably on a macroscopic scale (Figure 1.4), they still consist of the same microscopic building blocks: nerve cells (*neurons*) that communicate via electric pulses (*action potentials*) over connections called *synapses*.

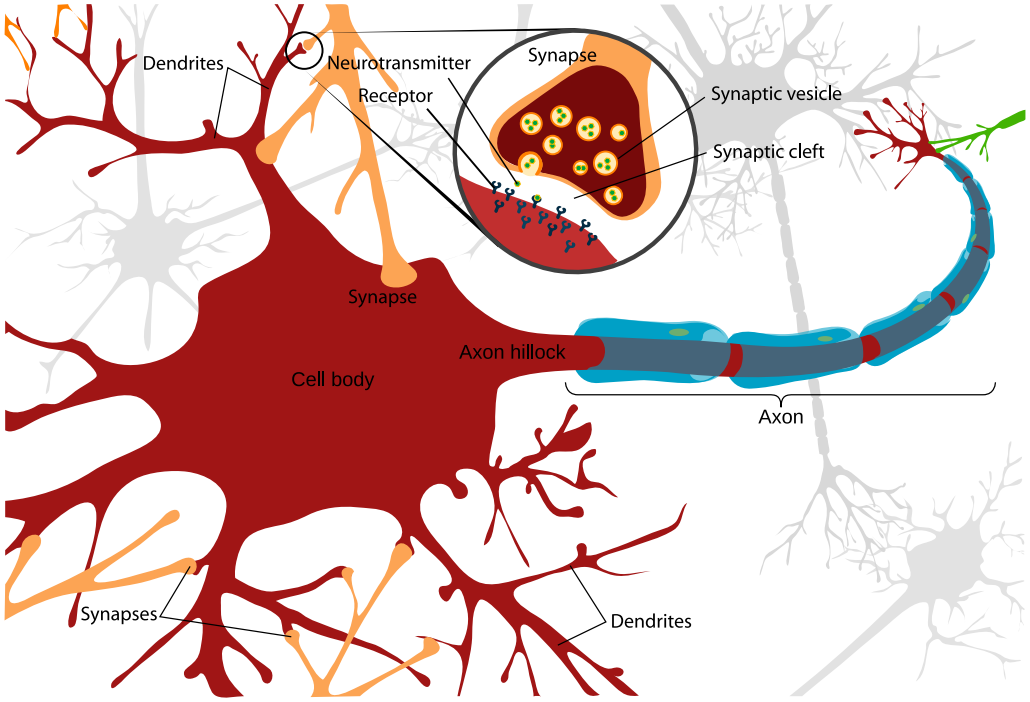


Figure 1.5: *Diagram of a typical myelinated vertebrate neuron (modified from Villarreal, 2007): The red neuron receives connections from the orange neurons and sends connections to the green neuron. The blue structures are Schwann cells that wrap around the axon and act as insulators to prevent the proliferation of a signal to other axons. The inset (circle) shows the structure of a synapse in detail, with neurotransmitters on the presynaptic bouton, and receptors on the postsynaptic side. Neurotransmitters are stored in synaptic vesicles before they are released into the synaptic cleft.*

The typical structure of a vertebrate neuron is shown in Figure 1.5. Neurons actively maintain an electrical potential across their membrane through various biochemical processes. Ion pumps constantly transport specific ion types out of the cell or into the cell. The cell membrane contains ion channels that are permeable only for specific types of ions (e.g. Na^+, K^+, or Cl^-) or larger charged molecules. Most of the channels are not static, but are opened or closed depending on the membrane potential of the cell. In the resting state, the membrane

potential fluctuates around a resting potential, which lies around -70 mV for a typical pyramidal cell in the cortex. The fluctuations are caused by the irregular input the neuron constantly gets from other cells that have connections to it. If the fluctuations of the membrane potential reach a certain threshold value in the cell body, the cell fires an action potential (*spike*), which is propagated along the axon to other neurons. An action potential is a short but large excursion of the membrane potential, which travels along the axon as a wave of opening and closing ion channels in the membrane. After the spike, the neuron is inactive (*refractory*) for a certain time in the order of some milliseconds. The inactivity is caused by an inactivation of the channels that contributed to the action potential. If the traveling wave reaches a synapse, it leads to the release of a chemical neurotransmitter into the synaptic cleft. This is registered by the receptors in the membrane of the postsynaptic cell. Depending on the type of the postsynaptic neuron and the synapse, the receptors trigger a rise (*excitation*) or decline (*inhibition*) of the membrane potential in the postsynaptic cell, which propagates to the cell body (Nicholls et al., 2001; Kandel et al., 2000).

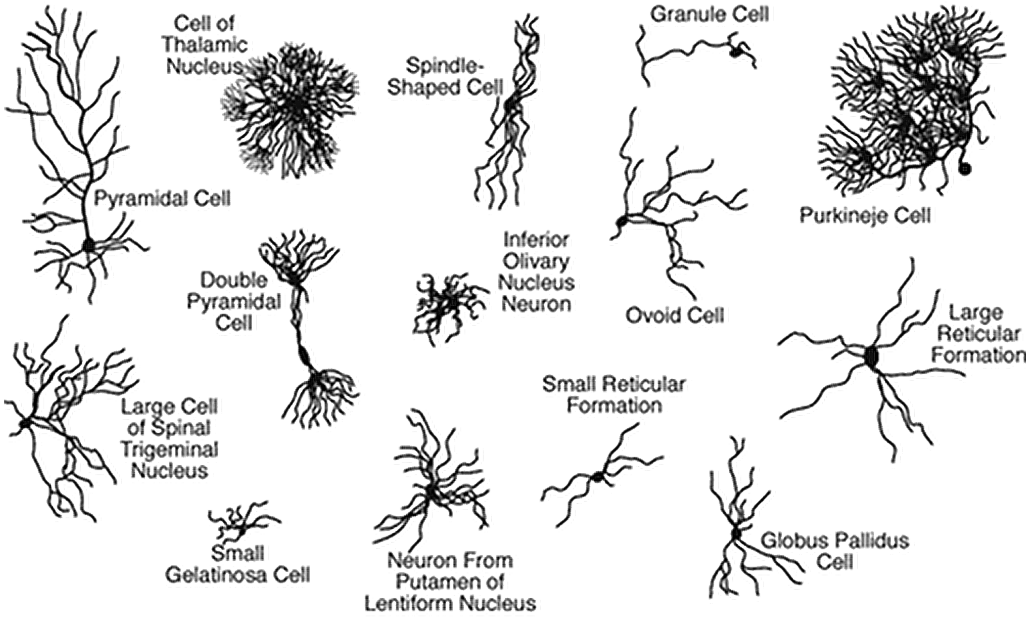


Figure 1.6: *Different neuron types (taken from Stufflebeam, 2008, based on drawings made by Santiago Ramon y Cajal): Neurons in different brain areas and in different laminar locations differ in their shape. Shown are the most important classes of neurons.*

These basic working principles are the same in all neurons. However, different types of neurons exist. They have different shapes (see Figure 1.6) and express different genes, which result in the embedding of different molecules and channels into their membrane, and thus in different electrical properties of the whole cell. For example, if a constant step-current is applied to the neuron, some neurons respond by firing an initial burst of spikes and are silent after that, although the stimulus is still present (*bursting neuron*). Other neurons respond by firing constantly with a high (*fast spiking neuron*) or a low (*regular spiking neuron*) rate. The different types can be classified using electrophysiological methods (e.g. Nowak et al., 2003). Likewise, different types of neurons react differently to spike input from other neurons. Another

difference between neurons is the type of synapses they make. Dale's principle (Eccles et al., 1954) states that a neuron either makes only inhibitory or excitatory connections, independent of the type of the neuron it connects to. The type of neurons one finds when looking into the brain depends on the brain area and the laminar position.

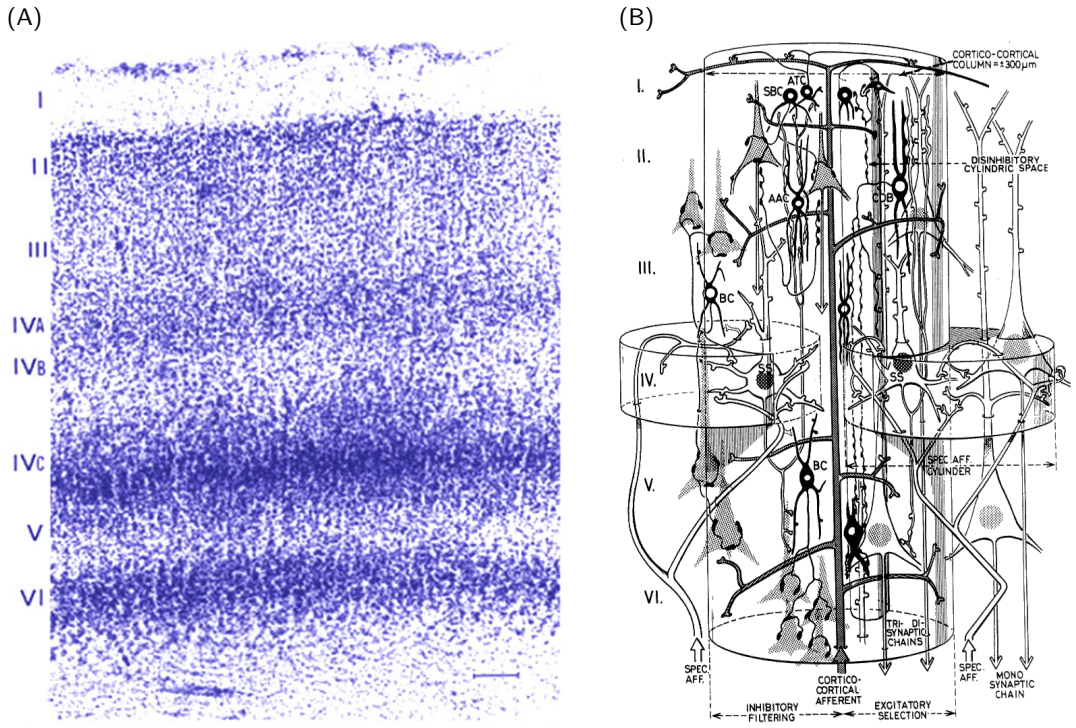


Figure 1.7: *Organization of the cortex into layers and columns: (A) Nissl stain reveals the different layers (I - IV) of the cortex quite clearly (taken from Schmolesky, 2000). (B) Columnar structure of the cortex (taken from Szentágothai, 1978). A column is believed to provide a functional unit that is replicated all over the cortex and carries out the same operation on different input data.*

Vertically, neurons in the cortex are arranged in (up to) six distinct *layers* that are clearly visible if brain slices are stained using histo-chemical methods like for example the Nissl method. The concrete number of layers depends on the brain area and species. One theory is that each layer performs a specific role in the information processing: for example, neurons in the layers II and III project mainly to other areas of the cortex, while those in the layers V and VI project primarily out of the cortex, e.g. to the thalamus. Layer IV is the main input layer. In primary sensory areas, layer IV receives synaptic connections from outside the cortex (e.g. from thalamus), while in higher cortical areas it receives information from lower areas. Neurons in layer IV mostly make local connections to other cortical layers. This means that layer IV receives incoming sensory information and distributes it to the other layers for further processing (Thomson & Bannister, 2003; Binzegger et al., 2004). Horizontally, the neurons are arranged in *columns* that are believed to be small microcircuits that are repeated all over the brain and carry out the same function but on different input signals (Szentágothai, 1978; Schrader et al., 2009). The microscopic organization of the cortex is shown in Figure 1.7.

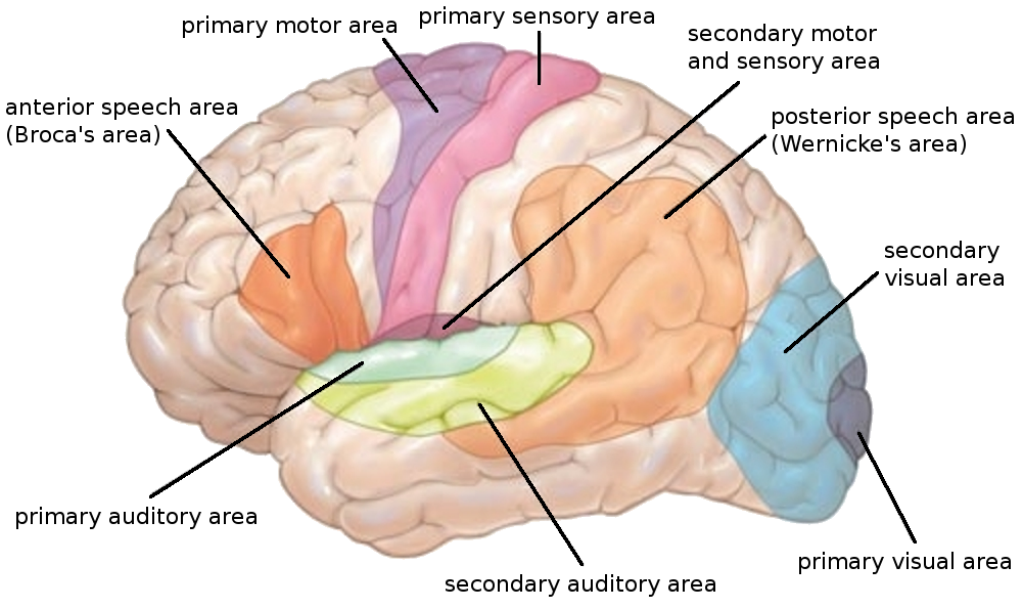


Figure 1.8: *Cortical areas (taken from Encyclopædia Britannica, 2007): The colored areas show the separation of the cortex into areas for the processing of different responsibilities. Shown are the areas for processing auditory, visual, and sensory information, for speech processing, and for motor control.*

On a coarser scale, the brain is organized into *areas* that are specialized for a specific task, e.g. processing of sensory information for the different modalities (audition, vision and somato-sensory), motor control, and higher cognitive functions like speech recognition and production, or object recognition (Brodmann, 1905). Although this macroscopic organization is known for over a century now, current research is still concerned with the investigation of the detailed function of the different areas. The function of the brain emerges from the interaction between the areas, which are connected by long-range connections to integrate the information accumulated in different parts of the brain. Figure 1.8 shows how the different areas are distributed over the surface of the cortex.

1.4 Models of the nervous system

Since the discovery of the basic building blocks of the nervous system, researchers in the natural sciences try to understand the working principles of the brain, and engineers try to utilize their computational power for technical applications.

1.4.1 Artificial neural networks

In Computer Science and Engineering, a method called *artificial neural network* is used since the 1940s. It is based on the idea that brain function can be understood in terms of many identical processing elements (the neurons) with weighted connections between them. These networks are successfully used for pattern classification, completion and storage, for solving optimization

tasks, and for machine learning tasks in general. Prominent examples for such networks are the Perceptron (Rosenblatt, 1958), associative memories like Hopfield Nets (Hopfield, 1982), and Self-Organizing Maps (Kohonen, 1984). In this framework, a neural network is described by a real valued state vector, a weight matrix, and a transfer function, typically a sigmoid function such as $\tanh(x)$. A learning rule is used to optimize the weight matrix, such that a given target function is approximated. The networks used in this domain are usually quite small and have a connectivity structure that fits the task, rather than reflecting the connectivity in the brain. In neuroscience, these models were quickly abandoned, because they cannot describe the properties of real neurons, and, more importantly, they cannot explain brain function.

1.4.2 Biological neural networks

Already in the 1950s and 1960s, neuroscientists conceived accurate models of the electrical signal flow in and between nerve cells. Models resulting from these efforts are called *biological*, *natural*, or *spiking neural networks*. The key difference to artificial neural networks is that the biological networks consist of neuron models that replicate the behavior of *real* nerve cells in great detail, instead of being optimized for solving a certain task.

Neuron models

The diversity of real neurons must also be reflected in the models of these neurons. Over the years, many different models were published by the researchers in computational neuroscience (for an overview, see Gerstner & Kistler, 2002; Dayan & Abbott, 2001) with the goal to create a mathematical description that replicates certain aspects of real neurons, such as its spiking behavior, its membrane potential, or its response to a certain stimulus.

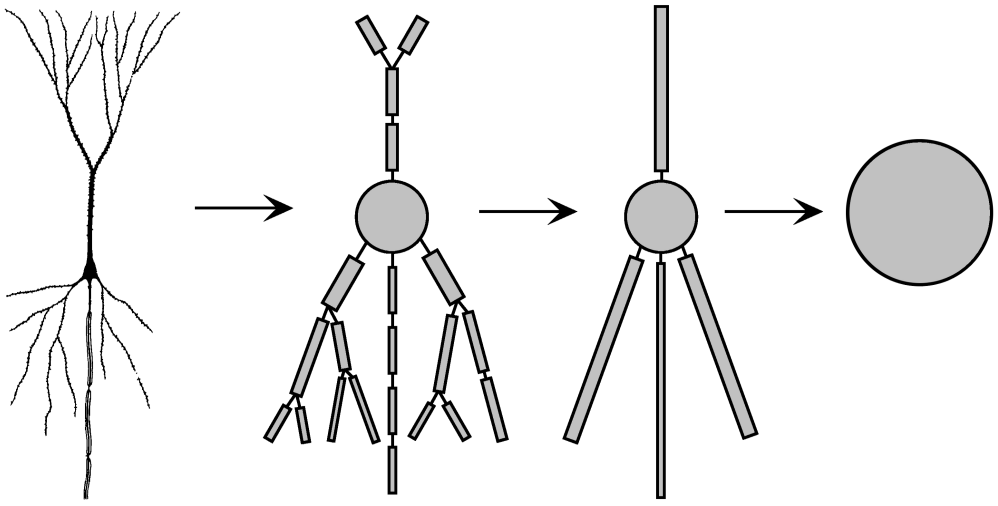


Figure 1.9: *Sketch of the abstraction levels for neuron models (taken from Dayan & Abbott, 2001): The neuron is represented by a variable number of discrete compartments, each representing a region that is described by a single membrane potential. Starting from a real cortical pyramidal neuron the neuron is simplified from several-compartmental models to a point neuron (i.e. a single compartment) model.*

Different fields of computational neuroscience investigate the brain on different levels of detail, starting from the molecular level of reaction-diffusion equations over networks consisting of more or less detailed neuron models, up to the functional level of models describing whole brain areas. Biological neuron models are mathematical descriptions that are based on the observations and measurements in *real* neural systems. The abstraction level of a model is chosen according to the aim of the study. For example, if signal transduction in single cells is to be investigated, detailed models of single neurons are the best choice, while network effects can be studied much better with neuron models that allow an investigation of the dynamics within large populations of neurons. The range of possible abstractions for such models is sketched in Figure 1.9. The most important categories of biological neuron models are summarized in the following list:

Reaction-diffusion models describe the interaction of molecules inside cells or at the cell membrane. Using this type of model gives insight into the chemical processes that lead to the higher-level behavior of the cell that is observable in electrophysiological experiments. Due to the complexity of this approach, it is currently not possible to simulate whole cells or networks thereof on a computer.

Compartmental models (Rall, 1964) have a large number of electrical compartments that describe the propagation of action potentials, and the dynamics of parameters like the membrane potential in the dendrites and axon of a cell. Each compartment is modeled by a set of equations from cable theory that describes the signal transduction in the respective section. These models are often based on three-dimensional reconstructions of real neurons. The simulation of networks of these neurons is possible, but very demanding with respect to memory and computing power.

Point neuron models (MacGregor, 1987) only have a single compartment and are thus more abstract than multi-compartmental models. The single compartment is described by equations to model the membrane potential and spike generation. These neuron models are well suited for simulations of large networks. The most prominent member of this type of neuron models is the *integrate-and-fire neuron*, which integrates its inputs and fires a spike if the membrane potential crosses a certain threshold.

Population models are even more abstract than point neuron models. They treat whole populations of neurons with similar properties as one entity. Most commonly, these entities are described by the mean firing rate of the neurons inside. A transfer function is used to calculate the output rate of one population, given the input it gets from other populations.

Field models are another type of population models that do not only describe the activity of populations of neurons, but also take into account the spatial composition of the populations and cell assemblies and their spatial extent.

Classical neuroscience has produced huge amounts of data as a result of in vivo and in vitro experiments in different areas of the brain and in different species. This data allows a precise characterization of the internal dynamics and the spiking behavior of many different types of neurons. The measured electrical and chemical properties of these neurons provide a basis that allows to build models that replicate the behavior of a real neuron with great detail.

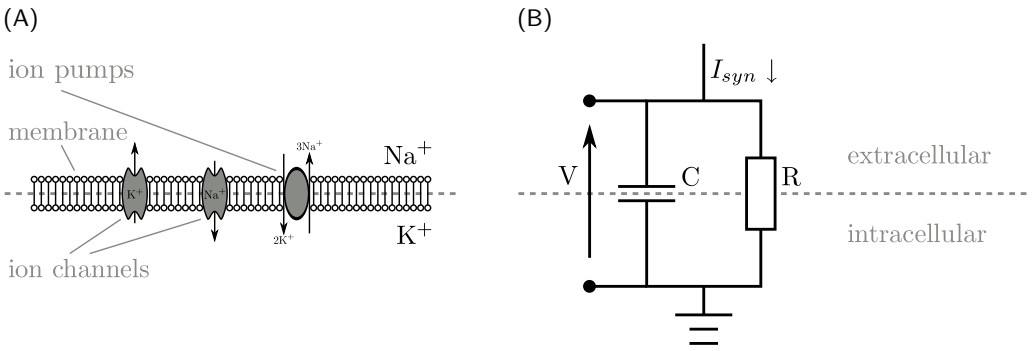


Figure 1.10: *Modeling single neurons: (A) Schematic of the cell membrane of a neuron. The membrane consists of a bi-lipid layer and is impermeable for ions and larger molecules. Ion pumps and channels are built into the membrane and allow the selective passage of certain ions to keep the membrane potential at its resting level. (B) Equivalent circuit. The membrane isolates the inside from the outside and thus functions as a capacitor. The channels in the membrane can be modeled as resistor.*

Figure 1.10 (A) shows a cross-section of a neuron's membrane, and the ion channels and pumps built into it. The semipermeable cell membrane separates the inside of the cell from the outside, and thus functions as a capacitor. The channels allow certain ions to pass the membrane. If the channels are lumped together, they can be modeled as a resistor. If an input current I is injected into the cell, it may add further charge on the capacitor, or leak through the channels in the cell membrane. Altogether, this can be seen as an electrical circuit as shown in Figure 1.10 (B). This circuit is a classical RC circuit and describes the charging of the membrane of the neuron without any spiking. Applying current preservation laws yields the following differential equation, which can be solved for the membrane potential V:

$$C\dot{V} = -\frac{1}{R}V + I_{syn}$$

To complete the model, a spike threshold is applied to the membrane potential. If it is crossed, a spike is emitted and the membrane potential is reset to its resting value. To account for the refractoriness of real neurons, the membrane potential is clamped to this value for a certain time after the spike. After this time, the neuron starts to integrate again. This simple point neuron model is known as the standard *integrate-and-fire neuron* (cf. Tuckwell, 1988).

The parameters for the neuron model (e.g. resistance, resting potential, etc.) can be extracted from real neurons in neurobiological experiments. Additional data from real neurobiological experiments can be integrated into the formal description to make the model more realistic. For example, one can split the single resistor that corresponds to the channels into multiple resistors, one for each type of ion channel. Another way to make the model more realistic is to replace the linear dynamics of the integrate-and-fire neuron by non-linear dynamics by adding conductances for the channels. The resulting model has been described as one of the first neuron models by Hodgkin & Huxley (1952). To build compartmental models, it is possible to combine several of the above circuits according to the three dimensional morphology of a real neuron.

Synapse models

Synapses do not only transmit the signals from one neuron to another, but play an important role in the information processing themselves. Synapses are not the same all over the brain, but differ from each other by the type of neurotransmitter they use, their strength (*weight*) and by the time they need to transmit the signal (*delay*, usually in the order of 1 ms). The strength of the synapse is not static, but can change due to the activity of the pre- and postsynaptic neurons. Heavily used synapses get stronger, while less used synapses get weaker. In addition to this *functional plasticity*, new synapses can grow in places where axons and dendrites are close enough, and die if they are not used (*structural plasticity*; Lendvai et al., 2000; Trachtenberg et al., 2002).

The simplest form of synaptic plasticity is the so-called short-term depression/potentiation (*STD*/*STP*). Here, the synapse gets weaker/stronger when many consecutive spikes are transmitted through it. STP and STD both happen on the time-scale of milliseconds, seconds, and minutes and allow the nervous system to adapt to ongoing stimulation. These changes are only temporary and do not change the molecular structure of the synapse. Once the stimulation ends, the synapse will return to its former state (Figure 1.11 (A); Tsodyks et al., 1998).

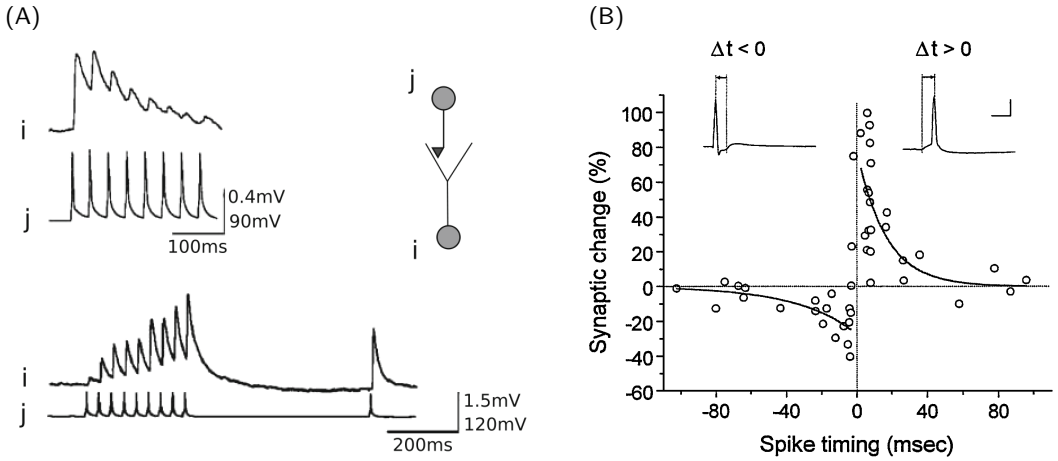


Figure 1.11: *Different forms of plasticity: (A) Experimental results from rat cortex in slice. The average amplitude of the evoked postsynaptic potential in neuron i varies with each successive spike of the presynaptic neuron j. Top panel: depression; bottom panel: facilitation (modified from Markram et al., 1998). (B) Weight modification in spike-timing dependent plasticity is a function of the relation of pre- and postsynaptic spikes (taken from Bi & Poo, 1998a).*

Other forms of plasticity are happening on a much slower time-scale, and go together with molecular and structural changes in the synapse. These slow changes are known as long-term depression/potentiation (*LTD*/*LTP*). They are happening in the range of hours and days and will stay for months and even years. Long term-facilitation and long-term depression are therefore the basis of learning and memory. These mechanisms are based on new receptors that are built into/removed from the cell membrane. One potential mechanism that underlies these long-term changes is spike-timing dependent plasticity (*STDP*; see Figure 1.11 (B); Bi & Poo, 1998a; Markram et al., 1997), which increases or decreases the weight of the synapse depending on the temporal relation of the spikes of pre- and postsynaptic neuron.

Apart from this *local* type of plasticity, some *global* forms are involved in the regulation of the dynamics of populations of neurons, and the activity in whole brain areas. This kind of plasticity is mediated by neuromodulators like acetylcholine, or certain hormones that are released into the tissue by arborizations of neurons that innervate a larger area. These modulating substances can activate or inactivate certain types of ion channels, or can modify the effect of certain neurotransmitters. As such, they play an important role in attention control, and enable learning and plasticity on a global scale.

In recent years, detailed models for synapses have been published. In the case of short-term plasticity they capture the relation of neurotransmitter synthesis, release, and uptake. Models for STDP are based on the temporal relation of the spikes of pre- and postsynaptic neurons (Gütig et al., 2003; Morrison et al., 2006).

Network models

Although there is a huge diversity of models for neurons and synapses, it is possible to find a common level of description for networks of spiking neurons: these networks can be described formally as graphs of *nodes* and *edges*. Each of the nodes contains a neuron model, while the edges can contain the different synaptic properties. As the communication in such a network is based on spikes, the type of model does not matter for the function of the network. Depending on the complexity of the neurons, these networks can be analyzed analytically or in computer simulations.

Simulations of neural systems

Having a formal description of neurons, synapses, and networks allows to set up a simulation that can be evaluated on a digital computer.

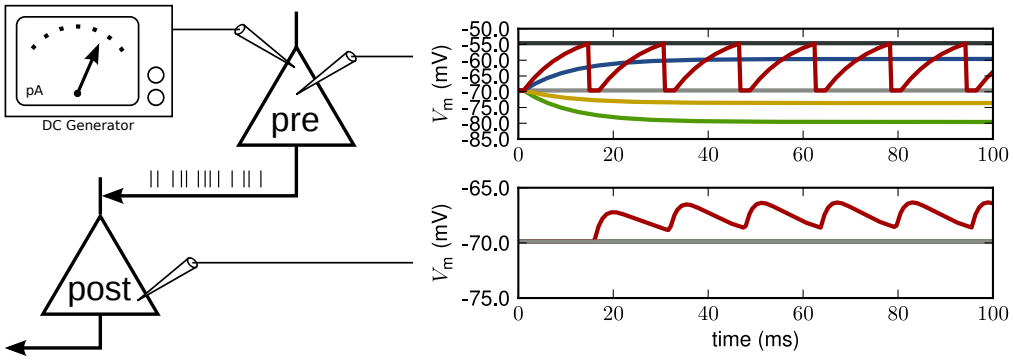


Figure 1.12: *Input summation in the integrate-and-fire neuron: Left: the triangles represent neurons (*pre *connects to* post*;* pre *is stimulated by direct currents of different amplitudes). Right: membrane potential traces of the two neurons for the stimulation with direct currents of different amplitudes (green: -250 pA; yellow: -100 pA; blue: 250 pA; red: 500 pA). Only the 500 pA simulation elicits a spike in* pre *(indicated by the reset of the membrane potential to the resting potential). Spikes are transferred to* post*, where they lead to a small rise of the membrane potential V_m. The spiking threshold is indicated by the horizontal dark gray line, the resting potential by the horizontal light gray line.*

The result of a small simulation with only two neurons and a current generator is shown in Figure 1.12. The first neuron (*pre*) is stimulated by a direct current with different amplitudes. The second neuron (post) only gets input by the first neuron, and thus only if *pre* fires a spike. If the membrane potential of the first neuron does not exceed the threshold (indicated by the dark gray line in the figure), the second neuron does not get any input and its membrane potential stays at the resting potential (indicated by the light gray line in the figure). Only the stimulation with 500 pA (red trace) is able to elicit spikes in the first neuron (at the times, when the red trace touches the dark gray line), and thus leads to an elevation of the membrane potential in the second neuron.

1.5 Simulators for biological neural networks

Since the early days of computational neuroscience, many different simulators for biological neural networks have been developed. Most of them are specialized for one specific level of detail. The following list contains the most important and most popular programs in the field (in alphabetical order) together with their main characteristics. In addition to these, many laboratories still use "home grown" simulators that are not available to the public. A more detailed list of simulators and their features can be found on the homepage of the user community of the Emergent simulator at `http://grey.colorado.edu/emergent/index.php/Comparison_of_Neural_Network_Simulators` and in Brette et al. (2007).

Brian (Goodman & Brette, 2008) is a simulator for spiking neural networks available on almost all platforms. The Brian package itself and simulations using it are all written in the Python programming language uses vectorized operations based on NumPy to achieve the necessary performance even for larger networks. Models in Brian are defined directly by their equations.

GENESIS (Bower & Beeman, 1997) is a general purpose simulation platform that was developed to support the simulation of neural systems ranging from sub-cellular components and bio-chemical reactions to complex models of single neurons, simulations of large networks, and system-level models. Simulations are set up using the built-in simulation language interpreter.

MCell (Kerr et al., 2008) is a modeling tool for realistic simulations of cellular signaling in the complex three-dimensional sub-cellular microenvironment in and around living cells. It simulates stochastic reaction-diffusion systems and allows the investigation of reactions between molecules and ions over time. MCell itself does not have a graphical user interface, but the companion project DReAMM can be used to create and visualize model descriptions for MCell.

MOOSE (Ray & Bhalla, 2008) is the Multiscale Object-Oriented Simulation Environment. It is the base and numerical core for large, detailed simulations in computational neuroscience and systems biology. Being based on GENESIS, it inherits a powerful framework to send messages between different components of the simulation, for example from one compartment to another or between different levels of the same model. The simulations are set up using an extension module for the Python programming language.

NEST (Gewaltig & Diesmann, 2007) is a simulator for spiking point neuron models. It is optimized for the simulation of large networks of relatively simple processing units that are connected over static or plastic synapses. NEST is controlled by a simulation language interpreter. Recent releases additionally provide a convenient Python interface to the simulator engine. A detailed description of NEST is contained in the following section.

NEURON (Hines & Carnevale, 1997) is a simulation environment, which is used in classrooms and laboratories around the world for building and using computational models of neurons and networks of such neurons. It supports compartmental models as well as simple point neurons and provides tools for conveniently building, managing, and using models in a way that is numerically sound and computationally efficient. The setup of the neurons and networks can either be done in a graphical user interface or by using a scripting engine based on Python or the custom programming language HOC.

PCSIM (Pecevski et al., 2009) is a tool for distributed simulation of heterogeneous networks composed of different model neurons and synapses. It is specialized for the simulation of point neuron models and biologically realistic synapse models, but also supports population models and algorithms for learning in artificial neural networks. PCSIM has a user interface based on the programming language Python.

SPLIT (Hammarlund & Ekeberg, 1998) is a simulator designed for efficient simulations of large networks on several architectures. SPLIT supports the simulation of compartmental neurons and point neuron models. However, SPLIT comes as a C++ library and the model description also has to be written in C++. This means that changes to the model require a recompilation of the simulation program.

STEPS (Wils & De Schutter, 2009) is a simulation platform for modeling and simulation of coupled reaction-diffusion systems with complex three-dimensional boundary conditions. The current version of STEPS has a Python interface for the interactive setup of model descriptions. STEPS does not have a graphical user interface. The user can use third party tools like Matlab, SciPy or Matplotlib for the visualization of the model.

1.5.1 The neural simulation tool NEST

NEST (Gewaltig & Diesmann, 2007) is a simulator for large networks of spiking neurons. Its development started in 1996 as an internal research tool. Since 2004, NEST is used at several international summer schools and advanced courses for computational neuroscience. The NEST Initiative frequently creates public releases of the source code that are available free of charge at its homepage at `http://www.nest-initiative.org/`.

NEST is written in C++ and runs on all POSIX compliant platforms. It has a built-in simulation language interpreter that allows the interactive definition and simulation of the network. The interpreter uses a stack-based programming language with a syntax based on PostScript (Adobe Systems Inc., 1999). NEST runs on a large range of architectures from ordinary desktop computers to computer clusters with thousands of processor cores (Plesser et al., 2007). This is achieved by using a hybrid strategy with threads on single computers and message passing across the cluster.

Most built-in neuron models in NEST are point neurons, although there is no principal restriction to this type of neuron models. However, as models have to be written directly in

C++, and because there is no tool support for the creation of new neuron models, the creation of multi-compartmental models is tedious compared to simulators like NEURON or GENESIS, which provide graphical user interfaces for this task, and allow to import complex geometries.

The network in NEST is represented as a weighted, directed graph of nodes and connections between them. The nodes represent either neurons, or devices used for the stimulation and observation of neurons. The connections can be either static, or they can change their weights according to synaptic plasticity rules like STDP (Morrison et al., 2008).

1.6 The software crisis in neuroscience

Until the beginning of the last decade, the computational neuroscience community was split into many different working groups, each of which specialized on a specific area of the brain and on a specific level of detail. The different laboratories did not collaborate much with researchers outside their own field. Little by little it became clear that this approach does not lead to an understanding of the brain as a whole, and that projects on the system level are required to conquer the complexity. This paradigm change lead to the foundation of many large-scale projects that integrate the work of the different projects that were formerly separate. The following paragraph contains a summary of the most important interdisciplinary projects of this kind:

The Blue Brain Project (`http://bluebrain.epfl.ch/`) is the first comprehensive attempt to reverse-engineer the mammalian brain, in order to understand brain function and dysfunction through detailed simulations. The project aims to simulate a cortical column with a high degree of biological realism by using multiple methods from classical neuroscience, through genetics to characterize the neuron types, to computer simulations. The main simulator used for the simulations is NEURON.

SyNAPSE (Systems of Neuromorphic Adaptive Plastic Scalable Electronics; `http://www.darpa.mil/dso/thrusts/bio/biologically/synapse/index.htm`) is a program by the Defense and Science Office of the DARPA to develop electronic neuromorphic machine technology that scales to biological levels. It is a multi-disciplinary approach, coordinating technology development in the areas of hardware design, architecture, simulation, and environment. IBM is one of the industrial partners of this project.

DAISY (`http://daisy.ini.unizh.ch/`) is a project that researches the *daisy architecture* of the neocortex, consisting of populations of pyramidal neurons within cortical areas, and their embedding within inter-areal connections. Their hypothesis is that the daisy architecture is found uniformly all over the cortex and supports self-organized, context-dependent processing. The project tries to reverse-engineer this architecture to find novel methods for scalable, distributed, autonomous computation in information technology.

FACETS (Fast Analog Computing with Emergent Transient States; `http://facets.kip.uni-heidelberg.de/`) is an attempt to create a theoretical and experimental foundation for the realization of novel computing paradigms, which exploit the concepts experimentally observed in biological nervous systems. The project is a collaboration between different laboratories for classical and computational neuroscience, as well as experts for the creation of analog hardware. NEST is one of the simulators in FACETS.

A direct consequence of the foundation of these large-scale projects in the field of computational neuroscience is the need for large-scale simulations to integrate the data from different domains. This means that the software in the field has to be able to handle large amounts of data (e.g. large data sets in databases, large numbers of neurons and synapses in simulations, and large amounts of data in analysis software) in an efficient way. A more indirect consequence is the need for neuroscientists to also collaborate in the software domain. The reason for this is that the new large-scale scientific endeavors require the establishment of work-flows consisting of many different tools and thus more complex software, which cannot be handled by a single laboratory anymore. In addition, the large interdisciplinary collaborations between several groups mean that more people depend on the software, and an easy way to exchange model descriptions and data. However, at the beginning of this period, most of the tools were incompatible with each other.

Researchers in neuroscience became increasingly aware that progress in neuroscience is slowed down by the absence of corresponding database and simulation technology in the field. In 2005 this lead to the foundation of the International Neuroinformatics Coordinating Facility (*INCF*, `http://incf.org/`) by the Global Science Forum of the OECD. In this context, neuroinformatics is understood as "computer and information science for neuroscience" and the mission of the INCF is to coordinate the transfer of computer science competence into neuroscience to cope with the complex multi-scale data and models created by brain research. Research on databases had a head start, but in 2007 and 2008 several studies were published that summarized the situation and identified the key problem areas of simulation technology (Djurfeldt & Lansner, 2007; Cannon et al., 2007; De Schutter, 2008; Brette et al., 2007):

- A lack of independent reproduction of simulations from published articles.
- A lack of a standards for the formulation of model descriptions.
- A lack of interoperability between different simulators.

According to the studies, the current diversity of simulators has advantages for the reproduction and independent validation of simulations and models. However, in most cases, where models are published, only a specification of the model, rather than the complete implementation is presented. Such specifications are frequently incomplete and a reproduction is often only possible with the direct help of the original authors. Even if the complete simulation code is published, it can still be tedious and error-prone to extract the model. Due to the variety in model description languages used by the different simulators, and due to the lack of interoperability between simulators, it is thus still a major problem to reproduce simulations from published papers. A realistic approach to ease these problems was proposed by Cannon et al. (2007), who suggested to facilitate interoperability in two ways: standardized model descriptions and run-time interoperability between simulators.

Standardized model descriptions based on the extensible markup language (*XML*; Bray et al., 2000) have a long tradition in systems biology (e.g. *SBML*; Hucka et al., 2002). However, a comparable standardization process in computational neuroscience has only just started.

Run-time interoperability between different simulators for the same problem domain can be implemented in different ways. The simplest is by using *named pipes* to transmit data and commands from one simulator to another. A more elegant way is to build on a library that couples the different applications and takes care of the low-level details of the communication.

Another problem of the field was the lack of a culture of sharing and re-use with respect to software and technology. Large communities, especially in neural network modeling, still use "homegrown" software that is not available electronically. This problem was further aggravated by the fact that it was hard or impossible to publish articles about the algorithms and data structures used in neuroscientific software in the classical neuroscience journals.

However, when this problem was realized, new journals for the field of neuroinformatics were founded. Springer's *Neuroinformatics* and the open-access journal *Frontiers in Neuroinformatics* are only two examples. In addition to the new journals, new conferences and developer meetings emerged, the most notable being the *INCF Congress on Neuroinformatics* and the *CodeJam* workshop series organized by the members of the FACETS project.

1.6.1 Implications of the software crisis for NEST

The large-scale projects and the resulting problems outlined in the previous section have direct consequences for the simulation tool NEST. To be a valuable tool for system-level projects like FACETS, NEST needs to be able to run neural network simulations that involve millions of neurons and billions of synapses efficiently. This includes the efficient simulation of neuron models and the efficient exchange of data between the neurons, as well as the efficient setup and control of the network. For the last point it is crucial to provide a convenient user interface that lowers the barrier for new users.

To improve the readability of model descriptions, it is important for NEST to support standard description languages that allow an easy exchange of models between researchers. This can be achieved in two ways: first, by supporting standard model descriptions, such as NeuroML (Crook & Howell, 2007; Crook et al., 2007), that allow a simulator-independent model specification, and second, by providing access to the simulator in a way that is more accessible to the user and thus easier to learn. As the standardization process for NeuroML is not finished, we decided not to invest our resources into developing support for it yet.

In general, interoperability between simulators can also be achieved in two ways: first, by supporting offline interoperability on the basis of common description languages that are supported by many simulators, and second, by support for run-time interoperability to couple NEST to other simulators and tools for stimulus generation and data analysis. The computational neuroscience community is currently working heavily to develop solutions and strategies in all these areas. The following section introduces our solutions to the problems to ensure NEST's aptitude for the neuroscientific challenges of the future.

1.7 Contributions of this thesis

At the beginning of the project for this thesis, NEST suffered from several drawbacks:

- The algorithms and data structures for multi-threaded and distributed simulation presented in Eppler (2006) were not optimal. Especially the event delivery mechanism and the algorithms for connection setup and management were only prototype implementations that were not ready for production use. However, to support large-scale simulations (as needed in the FACETS project for example) these new features are required.

- NEST's user interface (the simulation language interpreter SLI) is hard to learn and use by novices, because of its syntax. In addition, model descriptions written in SLI are hard to read. This is a problem in an environment were many researchers depend on readable model descriptions that allow an easy re-use of components.
- Coupling NEST to other software for stimulus generation and data analysis was only possible by using hand-crafted tools, which are complicated to use, because the user is responsible for the synchronization of the different processes. In the context of multi-scale simulations, this hinders progress and complicates the collaboration with other researchers.

These problems are united by the fact that all of them are are communication problems: the algorithms and data structures for event delivery and the storage of connections in the simulation engine comprise the central communication infrastructure of NEST. The user interface of NEST is the main communication architecture for the communication between the user and the simulation engine, and can be used by higher software layers to control NEST. These interfaces have to provide access to the elements of the simulation engine in a convenient way even in multi-threaded and distributed scenarios. The exchange of data with other applications at run-time can be implemented on top of a general communication library for neuroscientific software.

During the projects that were the basis for this thesis, we analyzed the different communication problems in NEST and developed solutions for them or extended the already existing solutions:

- We improved the event delivery algorithm and corresponding data structures that support event delivery between nodes that are distributed over multiple computers in a cluster. The scheduling algorithms and data structures were optimized to support both, the old thread based simulation of NEST 1 and a new distributed simulation scheme. Furthermore, we designed solutions for problems that were found in earlier versions of the NEST simulation engine connected to accessing node and synapse parameters in a multi-threaded setup.
- We designed a convenient new user interface for NEST (PyNEST) that allows the formulation of model descriptions in the programming language Python, which communicates with the simulation kernel through NEST's old simulation language interpreter SLI. This also provides a natural way of running legacy code that has been written throughout the last ten years.
- We co-developed a simulator independent description language for neural simulations (PyNN) that uses the PyNEST API, and provides a common programming interface for many different simulators and the neuromorphic hardware developed in the FACETS project.
- We designed an interface for the communication between different applications at run-time. It is based on the MUSIC library (Ekeberg & Djurfeldt, 2008), which was developed by the International Neuroinformatics Coordinating Facility (INCF). This interface allows to build multi-scale simulations, and to couple NEST to other simulators and tools for stimulus generation, visualization, and data analysis.

1.7.1 Communication inside the simulator

For many neuroscientific studies, the run-time of simulations directly translates into the number of experiments that can be performed on a model in a given time. Especially studies that involve plasticity and structural changes in the network often require the simulation of long periods of biological time (in the range of hours and days) to see the effects on the weights of the connections and on the structure of the network. On the other hand, the recent trend towards large-scale simulations of cortical structures requires larger networks than would fit into the memory available on a single computer.

Both requirements can be solved by recruiting more than one processor per machine or by using more than one machine. There are two basic possibilities to do so: *threads* or *processes*.

Threads allow the parallel execution of program code in one process. Most modern operating systems provide threads as a built-in feature. This means that the user does not have to set up special libraries or auxiliary programs in order to use multiple processors. Using threads, the different parts of the program can work on the same data without the need for special communication facilities. In principle, the same data structures can be used for single- and multi-threaded execution of the program. However, using threads, a program is constrained to a single machine and thus it is not possible to simulate networks exceeding the memory available on a single machine.

Processes only have access to their own memory. If multiple processes are used to solve a common problem, it is necessary to split and distribute the data structures of the problem and exchange the data between the processes using a special *message passing* library. As the processes of a message passing application can run on multiple machines, the possible network size is not restricted by the memory of a single machine, but by the sum of memory available on all participating machines.

To take advantage of both threads and processes, it is possible to use a hybrid scheme with processes on cluster nodes and threads locally. This results in the greatest possible flexibility for the user. Using threads is an easy way to support the many-core processors that are becoming more and more widespread even in laptops and ordinary desktop computers without extra work for the user. Large networks require the use of large computer clusters or facilities for high performance computing (*HPC*), where the message passing libraries are installed anyway.

Especially for large numbers of processors, scaling is more important than absolute performance. NEST 1 only supported thread-based simulations. The data structures were designed to allow concurrent access by multiple threads. However, production use revealed several problems with the performance of these data structures. During this work we analyzed the data structures and algorithms and developed a new network representation that exhibits much better scaling. The difference between the scaling behavior of NEST 1 and NEST 2 can be seen in Figure 1.13, which shows the run-time and speed-up for a simulation of a large random network model (Brunel, 2000). The results of this work are summarized in Plesser et al. (2007).

The work presented in Eppler (2006) was centered around performance improvements and combined thread-based and distributed simulation techniques. As such it constituted an important milestone in the development of NEST.

However, many of the improvements in the simulation engine were not easily accessible

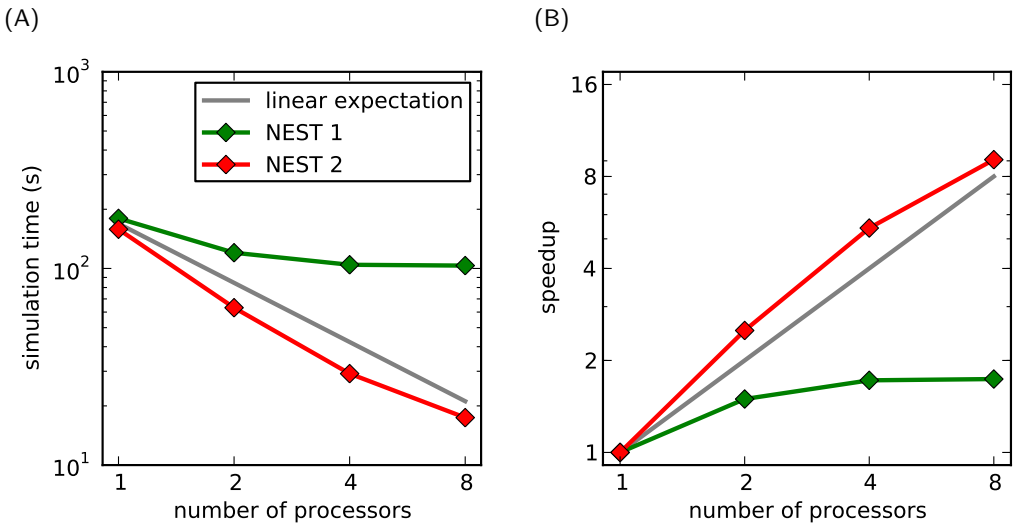


Figure 1.13: *Scalability of NEST 1 and NEST 2 with respect to number of processors: The simulated network contained* 10^4 *neurons, with 1000 random connections each; the exact model is described in Brunel (2000). Green line: NEST 1; red line: NEST 2; the gray line indicates linear speed-up. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speed-up* S_P *with* P *processors against number of processors, log-log representation (*$S_P = \frac{T_1}{T_P}$*;* T_1 *is the serial run-time,* T_P *the parallel run-time with* P *processors).*

for the user in a multi-threaded scenario. This was especially true for the access to nodes and connections in the network, due to the multi-threaded data structures of the network representation. During this work, we solved the remaining problems with these algorithms and data structures and improves the functionality and usability of NEST.

1.7.2 Communication between user and simulator

For the user, the most visible part of the software is the user interface. Because of the large number of elements in a typical neural network simulation in NEST, it is not sensible to use a graphical user interface, but instead we rely on a programmable scripting interface. NEST comes with a built-in simulation language interpreter (SLI), which has a syntax based on the PostScript programming language (Adobe Systems Inc., 1999). However, SLI was often criticized as hard to learn, because it uses *reverse polish notation* for all expressions. This means that functions expect their arguments on the operand stack and return their results to the stack, which is easy to parse for a computer, but makes the program code more complicated and especially math is hard to read. This made us think about a new user interface for NEST.

On the other side, SLI is used in many neuroscientific laboratories since over ten years. This means that a lot of code in SLI already exists and many published articles are based on simulations that were carried out with NEST and SLI. In addition, SLI is well tested and supports a multi-paradigm programming scheme that combines functional, object oriented, and procedural aspects in one language. A new user interface thus would have to be easier to

use than SLI and at the same time allow to run legacy SLI code.

A recent trend in computational neuroscience is to replace traditional tools like Matlab (MathWorks, 2002) and Mathematica (Wolfram, 2003) with Python. This happens for a number of reasons: Python is free software and is developed by a large and active community inside and outside science (Dubois, 2007). It provides a large number of extension modules for database access, graphical user interfaces, network services, scientific computing, etc., and can be used well as a glue language between different software components. However, Python is not available on some high-performance clusters that we still need to use, so a Python-only user interface is not an option.

Considering all this, we implemented a new user interface for NEST as an extension module for the Python programming language (PyNEST; Eppler et al., 2009). However, instead of using the traditional approach of creating direct bindings to the classes and functions of NEST, PyNEST provides a set of routines for data conversion from SLI to Python and back, and communicates with SLI to execute commands inside the NEST engine. PyNEST allows to specify stimulus generation, simulation, and data analysis in a single Python script, while still allowing the execution of legacy SLI code in a natural way.

A common interface for different simulators

Many other simulators are now also using Python as their primary user interface language or in addition to their old user interface (Kötter et al., 2009). This makes it easier to share model descriptions and code with others, and eases the task of porting models from one simulator to another. However, as most of the other simulators use different concepts for the definition of stimuli and models, this task can still be tedious.

The groups collaborating in the FACETS project are using many different simulators (currently NEURON, PCSIM, Brian and NEST). This diversity gave birth to the idea of a common user interface to all of them, based on their Python interfaces. To this end, we contributed to the design and development of PyNN (Davison et al., 2008), a tool which provides a simulator independent programming interface for multiple simulators by using adapter backends that take care of the setup and control of simulations, the translation of model names and physical units and the conversion of data files to a common format.

1.7.3 Communication between different simulators

As outlined above, models are created on different scales and by using different tools and simulators. One way of interoperability between different simulators is on the language level, PyNN is an example of this. However, PyNN does not allow the interaction of simulators at run-time, but only allows to control them with a common interface.

To get a broader view on the processes in the brain, it becomes more and more important to couple different tools and simulators in an on-line fashion: networks of point neurons can provide realistic input to models of single nerve cells, while functional models can provide input to networks of point neurons. Combining the models on different scales is complex, as the classical simulators only focus on a single level of detail and physical concepts such as time and units, or neuroscientific concepts such as neurons and synapses are represented differently in the different simulators.

Additionally, tools for stimulus generation and data visualization and analysis are currently tightly coupled to the simulators they are developed for, and depend on the exact data formats of the application that generated the data. It is desirable to have the ability of re-using these applications for other simulators to minimize the amount of duplicated work.

One approach to solving these problems is the Multi-Simulator Coordinator MUSIC (Ekeberg & Djurfeldt, 2008), which was developed by the International Neuroinformatics Coordinating Facility (INCF). It is a standard and an accompanying library that can be used by different simulators and tools to exchange data (spikes, continuous values and text messages) during run-time. MUSIC already comes with a set of tools for stimulus generation and recording of data, and simple data visualization programs. By providing an interface to MUSIC, it is possible to communicate with all other applications that have such an interface, without taking care of the low-level issues like synchronization and data transfer. During the work for this thesis, NEST was one of the first simulators that has been extended by an interface to the MUSIC library.

1.8 Character of this thesis

The solutions described in this thesis are a direct answer to research needs in computational neuroscience. As such, many of the presented solutions do not constitute new methods in computer science, but are a transfer and adaptation of computer science knowledge to the domain of neuroscience and neuroinformatics and important for the progress in these fields. This includes the analysis and refactoring of existing solutions to bring them to the level of state-of-the-art solutions in computer science.

In some cases new neuroscientific research directions also required new functionality. The work for this thesis was carried out in the public release of NEST (available from `http://www.nest-initiative.org/`), which imposed two important preconditions:

1. It was not possible to design the solutions from scratch, but rather the existing components had to be taken into account. Thus the changes had to be compatible with the algorithms and data structures already present.
2. The system had to stay usable throughout the development period of the new components and algorithms (Diesmann & Gewaltig, 2002).

The advantage of this approach was that the quality of the developed solutions was constantly evaluated and tested with real-world simulations and thus errors in the design or in the implementation could be found and fixed rapidly.

One implication of the rapid development in the field of computational neuroscience was the need to publish the new algorithms and methods as soon as they were tested internally and found working. The first way of publication was to create new beta releases of the software roughly once per month to give the users the possibility to use the new features for their research. The second way of publication was to write articles for high-ranked, peer-reviewed computer science and neuroinformatics journals and present the work at conferences for computational neuroscience and neuroinformatics to claim the authorship of the new methods. Each of the chapters in this thesis is accompanied by a journal publication in Appendix A, which is a distilled version of the work.

1.9 Structure of this thesis

In Chapter 2, we show how to represent biological neural networks for efficient simulation in multi-threaded scenarios, where multiple processors have to work on the same set of data, and in distributed scenarios, where the network has to be distributed onto many computers. We also describe the scheduling algorithm that allows an efficient delivery of events even with a large number of receivers. In addition we describe many improvements for problems with the multi-threaded use in earlier versions.

Chapter 3 describes PyNEST, the Python bindings for the NEST simulation kernel. PyNEST provides a shallow wrapper for NEST's simulation language interpreter SLI. Its API is modeled closely after the API of NEST itself.

To allow high-level descriptions of neural networks, we participated in the development of PyNN, which provides a common high-level API for multiple simulators. The basic design of PyNN is described in Chapter 4.

We designed and implemented an interface to the MUSIC library, which allows different applications (simulators, analysis tools, visualization programs, etc.) to exchange data at run-time. This interface is described in Chapter 5.

Chapter 6 contains a discussion of the work presented in this thesis and gives an outlook into the next steps of the development of NEST.

The publications that resulted from the work described in this thesis are contained in Appendix A for the convenience of the reader.

Chapter 2

Communication inside the simulator

The basic building blocks of information processing in the brain are nerve cells (*neurons*), which communicate by exchanging point events (*action potentials*, *spikes*) over their connections (*synapses*). Models of the nervous system are a valuable method to understand the working principles of these building blocks and to gain insight into the function of the neural networks in the brain. Simulations can be used to answer questions that cannot be addressed analytically and are not tractable in classical neurobiological experiments.

To understand the principles of information processing in the brain, we depend on network models with more than 10^5 neurons and 10^9 connections. These numbers follow from two statistical numbers that can be measured in the cortical tissue of vertebrates (Abeles, 1991):

- Each neuron receives approximately 10^4 connections from neurons nearby.
- Each neuron is connected to about 10 % of the neighboring neurons.

If we want to simulate networks which fulfill both of these criterions, we end up with at least 10^5 neurons and 10^9 connections, which corresponds approximately to the number of neurons and synapses found in one cubic millimeter of cortical tissue in a vertebrate (Figure 1.1; Braitenberg & Schüz, 1998). In general, it is desirable to use networks with this minimal size, as scaled down connectivity often leads to artifacts in the dynamics of the network and problems with its stability. From the computer science perspective, the key challenges for simulations of such networks are:

- The design of data structures that represent the neurons and connections succinctly.
- The mapping of inter-neuron communication onto efficient algorithms to transmit events.
- The update of neuron states with minimal effort.
- The convenient setup and manipulation of the network and its building blocks.

NEST (Plesser et al., 2007) is a simulator for networks of spiking neurons, which addresses all these requirements. To simulate very large networks in acceptable time and with acceptable memory requirements per machine, NEST uses a hybrid parallelization strategy, using multiple processes and message passing across the cluster and thread-based simulation on each compute node of the cluster.

Most large-scale scientific applications that use parallel computing just rely on *either* threads or message passing, because of the additional effort required to develop and maintain the data structures and algorithms for a hybrid scheme. The reason why we still use a hybrid strategy instead is twofold:

1. Threads allow parallel computing on multi-processor machines without the need for additional libraries. This is important to exploit multi-core processors out of the box, as these are becoming more and more widespread even in normal desktop computers and laptops. However, achieving good performance with thread-based parallelization is difficult, as the programmer has to take care of many low-level details, like cache efficiency and data locking in order to prevent concurrent accesses to the data. This is the reason why multi-threaded programming is often only used for applications where performance is not critical, like the decoupling of program logic and user interface.
2. Very large networks, however, do not fit into the memory of a single computer, which makes it necessary to distribute the network over a cluster of computers. On the downside, this means that the network representation has to be split and distributed onto different processes, running on different machines, and that the spikes of the neurons have to be communicated across a computer network.

The algorithms and data structures of NEST support parallel and distributed simulation of large-scale spiking neural networks with heterogeneous neuron and synapse types. We show that a single representation for the neural network can be designed, which is optimal for both, parallel simulation using threads, and distributed simulation using multiple processes and message passing for the transmission of events.

The communication architectures in NEST were optimized to minimize cache problems during simulation and to allow an efficient broadcast of the state of the neurons to other machines that participate in the simulation.

The user interface of NEST was adapted to the new simulation scheme and we designed solutions to inspect and modify the network elements in a unified way, although the elements may be distributed over different threads and machines.

2.1 The history of NEST

NEST has a long history of use in computational neuroscience. The roots of its first predecessors date back as early as 1994. Different branches of development were used to explore new solutions for novel problems faced in the neuroscience domain. This section contains the most important branches of the development that lead to the current version 2 of the NEST simulation system.

2.1.1 SYNOD

The first simulator developed by the NEST Initiative was SYNOD (Diesmann et al., 1995). It was implemented as a C++ library carrying out the simulation, and a simulation language interpreter (SLI) for the convenient setup of the network, and for the control of the library. The library supported different neuron models that could be used in the same network. Static

connections (i.e. synapses with static weight and delay) were used to connect the neurons, and simulated devices could be used to stimulate the network and to measure certain quantities (like membrane potential or spike output) from the neurons. Communication in the simulation kernel was not mediated by events, but rather by directly calling the respective function of the target neuron. This restricted the communication between the neurons to using spikes.

2.1.2 NEST 1

Based on the experience during the development of SYNOD, a new simulation engine called NEST 1 was developed. The name NEST is an abbreviation for NEural Simulation Tool. The simulation engine was completely re-written, while the simulation language interpreter was kept and extended to support the new features of the simulation engine library. The main improvements over SYNOD were:

Common base class for neurons and devices. In SYNOD, neurons and devices were separate entities. This meant that different sets of access functions were needed and that separate scheduling algorithms had to be used to update the elements. To remedy this situation, both element types in NEST are derived from the same base class (`Node`) and live together in a single network tree.

Support for structured networks. SYNOD only supported "flat" networks without structure. In NEST, networks could be organized hierarchically in sub-networks to reflect the structure in natural neural networks. Moreover, this allowed to structure networks into logical blocks to ease the work-flow of the researcher.

Communication via events. Neurons in SYNOD communicated by directly calling each other's functions. NEST introduced the concept of events, which allowed a more flexible communication using currents, rates and spikes. Using a central event delivery algorithm allowed to simplify the neuron model classes.

Support for multi-threaded simulations. This made it possible to use multiple processors in one machine. However, the data structures for the storage of nodes and connections were not optimized for thread-based simulations, which lead to a dissatisfying performance, if more than two threads were used.

Generic parameter interface for nodes. The use of dictionaries (*named parameters*; Finkel, 1996) together with the functions `GetStatus` and `SetStatus` allowed a minimal interface for the modification and inspection of node parameters and solved the problem of *fat interfaces* (Stroustrup, 1997): `SetStatus` gets a dictionary and a node id as argument and sets the parameters of the given node accordingly, while `GetStatus` gets a node id and returns a dictionary with the current parameters of the corresponding node.

2.1.3 Paranel

One of the first formal descriptions for the *distributed* simulation of neural networks in literature was published by Morrison et al. (2005). The simulator was called *Paranel* and is based on SYNOD. To support the simulation of large networks with heterogeneous synapse types on computer clusters, the authors made three major changes to the original design of SYNOD:

- Addition of message passing facilities on the basis of the *message passing interface* (*MPI*; Message Passing Interface Forum, 1994) and a network representation that distributes neurons and synapses onto multiple processes.
- Removal of the simulation language interpreter to reduce the overall complexity of the software and to simplify the network setup in a distributed scenario.
- Addition of support for plasticity and heterogeneous synapse types.

Paranel was very efficient and showed excellent scaling. Two improvements that are directly related to the distribution of the network were responsible for its good performance:

Communication in steps of the minimal connection delay. Each connection in the network has a delay δ. During the time Δ, given by the minimal delay in the network, no signals from other neurons can influence the state of receiving neurons, which means the network elements are actually decoupled during this time. The network is simulated on a fixed time grid with step size h. The decoupling now allows to send spike information to neurons on other processes in steps of Δ instead of h. This minimizes the communication overhead and thus increases performance.

Postsynaptic storage of synapse. Instead of storing the connection information of the neurons with the presynaptic neuron, it is stored on the process where the postsynaptic neuron is located. By only sending the ids of the neurons that fired a spike and recreating the full event on the receiving process, it is possible to minimize the data sent to each process, although this requires to broadcast all spikes to all processes.

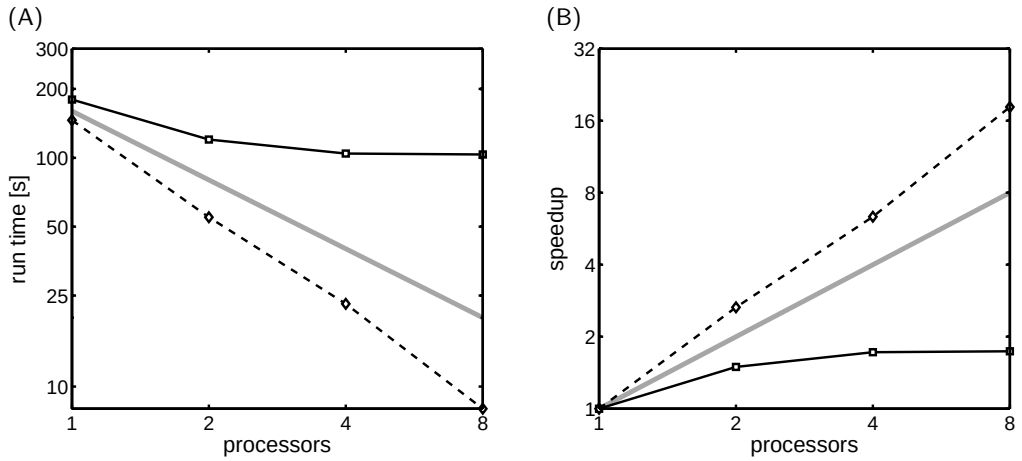


Figure 2.1: *Scalability of NEST 1 and Paranel with respect to number of processors (taken from Eppler, 2006, Figure 1.2): The simulated network contained 10^4 neurons, with 1000 random connections each; the exact model is described in Brunel (2000). Solid line: NEST 1; dashed line: Paranel; the gray line indicates linear speed-up. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speed-up S_P with P processors against number of processors, log-log representation ($S_P = \frac{T_1}{T_P}$; T_1 is the serial run-time, T_P the parallel run-time with P processors).*

Although NEST 1 and Paranel are both based on SYNOD, they show quite different scaling. The main reason is the different memory layout of the two simulators: Paranel uses multiple processes, each of which is assigned to one processor. NEST uses multiple processors within a single process, which can lead to performance problems (see Section 2.2).

Using multiple processors on a single machine only makes sense for applications that have good scaling. The comparison in Figure 2.1 clearly shows that the scaling of NEST 1 was insufficient and that Paranel was superior in this respect. However, the simulation language interpreter made the usage of NEST 1 more convenient than the use of Paranel, where changes in the network required a re-compilation of the whole simulation program. These insights lead to the exploration of more efficient update schemes for NEST.

2.1.4 Express

To improve the performance of the simulation engine of NEST 1, the update algorithm from the Paranel engine was adapted to the thread-based simulation scheme in 2005: in a development branch of NEST 1, called *Express*, we developed a first proof-of-concept implementation of the idea that not only communication between nodes, but also the update of nodes could be carried out in steps of the minimal connection delay Δ instead of the simulation time step h (Plesser, 2005). This means that more operations can be performed on the same data, which is already in the processor's cache memory, and that the cache has to be reloaded less often from the slow main memory (cf. Section 2.2). As a consequence this results in improved performance and better scaling.

2.1.5 NEST 1.9

The work presented in Eppler (2006) constituted a first prototype implementation for the release of NEST 2. It combined the features of multiple of the previous versions in a single simulation system. This included the algorithms for distributed simulations from Paranel, the improved update algorithm for multi-threaded simulations from the Express simulation engine, and the simulation language interpreter and the other features from NEST 1. The main features of the resulting NEST 1.9 simulation engine were:

Support for multi-threaded and distributed simulation allows to simulate very large networks on clusters of multi-processor machines with acceptable memory requirements on each machine. The underlying network representation is optimized for both simulation schemes with comparable performance and scaling.

Support for heterogeneous synapse types allows to implement plastic synapse types, and to use different synapse types in the same network.

Node update in steps of the minimal connection delay reduces cache reloads and thus improves performance and scaling.

Although NEST 1.9 was running stable, it could not be immediately used in the neuroscientific day-to-day work. First applications to real-world scientific tasks by several researchers revealed some severe shortcomings of the implemented algorithms and data structures with respect to their performance and functionality:

1. To carry out multiple simulations in a single session, the simulation engine provides a function to delete the network and reset all simulator parameters to their defaults. NEST 1.9 also allowed to customize existing synapse and neuron models, and to create new models by copying existing ones. However, the design was not able to recover the original model parameters upon a reset of the simulator. The solution to this problem is described in Section 2.4.1.
2. While porting the old models from NEST 1 to the new API for models of the NEST 2 simulation engine, we found that the round-robin distribution of nodes onto the different processes is not compatible with compound models, where a parent node needs direct access to the data structures and functions of its child nodes. See Section 2.4.2 for a solution to this problem.
3. The original design of the system for heterogeneous synapse types had a global preset for the synapse type to use for new connections and for the retrieval of synapse parameters of existing connections. This *synapse context* was only implicit and lead to many errors in user code that were hard to find and hard to debug. In NEST 2, the synapse type is an explicit parameter that has to be given to all functions that require a synapse type to carry out their task.
4. There was no mechanism to prevent non-spiking devices to send their events over dynamic synapses. This lead to wrong simulation results, if e.g. a current generator was connected to a neuron via a STDP synapse, which interpreted the events as spikes and changed the weight of the connection accordingly. The new algorithm for checking the compatibility of connections and events is explained in Section 2.4.3.
5. Without knowledge of the internal network representation, it was impossible to query the current state or modify the properties of synapses once they were established. The reason for this was the distributed storage of connections and the lack of proper access functions for them. Section 2.4.4 explains the design of such access functions.
6. In multi-threaded simulations, it was impossible to retrieve the data, which was collected by devices, without explicitly looping over the threads. This lead to errors, as a single call to `GetStatus` in a multi-threaded setup only revealed a subset of the collected data. Section 2.6 describes the solution to this problem.
7. When our users went to larger clusters or high performance computing (*HPC* facilities as e.g. the BlueGene, we found that the use of the CPEX algorithm (Tam & Wang, 2000) for the broadcast of spike information to all processes only provided insufficient scaling. For NEST 2, we replaced this with MPI's `Allgather()` function. The design of a new log-level communication system is explained in Section 2.5.1.

2.2 Cache efficient software design

Previous versions of NEST suffered from performance and scaling problems. One of the major performance bottlenecks in multi-threaded applications is a cache-inefficient memory layout of data structures that leads to cache-inefficient algorithms. If the working principles of the cache are known, several measures can be taken to avoid these problems.

Modern processors can process data much faster than the main memory can deliver it. To solve this problem, the processor contains a small working memory called *cache*, where it stores the data it currently needs. The main memory usually has a size in the range of a few Gigabytes and access times around 10 ns. The cache, however, runs at the full clock speed of the processor, but has a much lower capacity of up to only a few Megabytes (Tanenbaum, 1999). This means that data is frequently forced out of the cache and replaced with new data form main memory.

When the processor needs data, it first looks into its cache. If the data it needs can be found there, it can retrieve that data into its registers with little or no delay and work with it. If the data is not in the cache, it is fetched from main memory. Due to the high latency of this operation, the overall performance of the program decreases, because the processor has to wait for the data. In order to minimize the number of cache reloads from main memory, the processor does not only fetch the data it currently needs, but also the data stored in the direct vicinity of the data needed (*pre-fetching*).

To improve the speed of a program, this can be exploited by the design of data structures that store related data together and thus increase the chance of a successful cache look up. This strategy works well, if only a single processor needs to work on the data. The matter becomes more complicated in multi-threaded scenarios, where many processors work on the same data and have a copy of the same memory location in their cache. The reason why this is more complicated is that the data needs to be kept consistent with respect to the other processor's caches, by writing the data back to main memory and reload all caches from there.

The solution for this is to split the data structures according to the different processors that have to operate on it. Using multiple processes (as Paranel did), this work is done by the operating system, which assigns a private memory block to each process. Communication between the different processors is carried out by passing messages with the relevant data. In multi-threaded programs, however, communication happens by direct access to the data structures and the programmer has to take care that the data structures fulfill the separation requirement.

2.3 Network representation

To avoid the cache problems explained in the previous section and to improve the flexibility for the use of heterogeneous synapse types, we designed a network representation that stores connections separate from nodes and separates the memory of the different threads.

The networks of point neurons (see Section 1.4.2) that can be simulated in NEST can be described as graphs of threshold elements that exchange spikes over their connections. In this representation, neurons correspond to the nodes of the graph, while connections correspond to the edges. The connections between the elements are characterized mainly by their *weight* and a *delay*, and can (in the case of plastic synapses) contain a function that updates the weight based on the spiking behavior of the pre- and postsynaptic nodes.

The network graph can be transformed into an *adjacency list*, where each node stores the list of nodes it connects to (Gross & Yellen, 1999). However, to allow efficient multi-threaded and distributed simulations, this representation has to be split into chunks of approximately the same size, which can be simulated using POSIX threads (Lewis & Berg, 1997) and MPI processes (Message Passing Interface Forum, 1994).

Events

Different types of events are used for the different types of data that can be sent between the nodes in NEST. The `Event` base class contains pointers to the sending and receiving node, the weight, and the delay of the connection. This base class is extended by additional members depending on the concrete type of the event. The simplest type, called *SpikeEvent*, contains only the basic data fields as it does not carry any data other than the time stamp, while the *CurrentEvent* contains an additional field for the amplitude of the current. Several other event types exist for other types of data. *Request* events are answered by sending the requested data back to the sending node.

2.3.1 Nodes

All nodes in NEST are derived from the common base class `Node`, which provides virtual functions to initialize and update the node as well as to handle incoming events during simulation. Each model class has to provide an implementation of these functions. When a node is created, it is assigned a global identifier (*global id*, *gid*), which corresponds to the order of its creation. During the update of a node, it can send and receive events. The interface of the nodes is made in such a way that the developers of neuron and synapse models do not have to take care of the distribution of nodes.

Node types

Three basic types of nodes exist in NEST: neurons, devices, and sub-networks. Neurons represent neuron models and are implemented as C++ classes with corresponding update functions. This is the case for point neuron models as well as for compartmental neuron models and means that neurons in NEST cannot be split and distributed over multiple processes. Devices are nodes that are used to stimulate the network, or to measure certain quantities from nodes. The latter can be simple recorders, such as *voltmeter*, or carry out data analysis on the fly, such as *correlation_detector*, which calculates the correlation between neurons. Sub-networks can be used to create structured networks, but do not necessarily correspond to a functional partitioning of the networks. When NEST is started, a single empty sub-network exists (*root node*) and serves as container for the complete network. The root node has the global id 0.

2.3.2 Storage of nodes

The neuron models that exist in NEST are very diverse with respect to memory requirements and the computational load they cause. To split the network in smaller chunks of approximately the same size that can be distributed onto the threads and processes, each node is assigned to one of N_{VP} *virtual processes* using a simple round-robin algorithm (Morrison et al., 2005): the virtual process id id_{VP} of a node n is given by

$$id_{\mathsf{VP}}(n) = id_{\mathsf{N}}(n) \mod N_{\mathsf{VP}}$$

where $id_{\mathsf{N}}(n)$ is the global id of the node. A virtual process is a POSIX thread that lives in one of N_{MPI} MPI processes. Each of the processes contains the same number of threads.

After the creation of nodes, the number of threads cannot be changed anymore.

Device nodes are created for each virtual process to allow parallel data i/o and distribute the load. This is particularly important for device nodes that have to deliver large amounts of data to their targets, or devices that are CPU intensive, such as random spike sources, or devices that have to record a lot of data to disk. To balance the load of all virtual processes, neurons are only created on the virtual process they are assigned to. On all other virtual processes, they have light-weight proxies that only store administrative information, but do not carry out any computations.

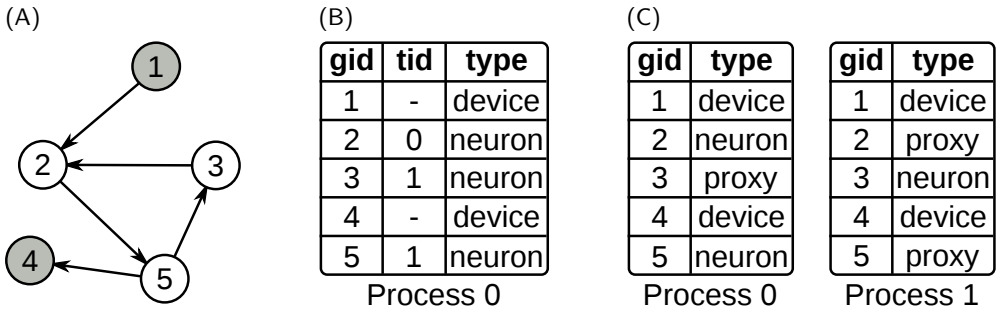


Figure 2.2: *Data structure for the storage of nodes: (A) The network as directed graph. The numbers correspond to the global id of the node. Grey nodes are devices; white nodes are neurons. (B) The network representation using a single process with two threads. The column labeled* gid *contains the global id of the nodes, the column labeled* tid *contains the thread the neuron is assigned to. (C) The network representation distributed onto two processes. The column labeled* gid *contains the global id of the nodes.*

The virtual processes can be distributed arbitrarily among threads and processes: a simulation with two processes and two threads each will yield the same results as a simulation with a single process having four threads, or a simulation with four processes with one thread each. In order to save memory with more than one thread, the network representations inside each process are merged, so that a single node list can be used. Each entry of the node list then contains an array with one entry for each thread. This means that the nodes for the different threads are stored in separate locations of the memory in order to reduce cache problems.

Figure 2.2 shows the network representation that was explained above applied to a network consisting of two devices and three neurons for a single process with two threads, and distributed onto two processes with a single thread each.

2.3.3 Connections

Synapses in NEST are objects that store the weight and the delay of the synapse, and optionally (depending on the type of connection) contain a method to update the weight according to a plasticity rule. All synapses are derived from the base class `Connection`.

During update, nodes can send events, which are delivered to their targets by the `Network` class. When created, these events only carry the time stamp of creation, and the data they shall transmit. During delivery, they are handed to the respective `Connection` object, which fills in the missing weight and delay and hands the event to the `handle()` function of the target node. This flow of events is depicted in Figure 2.3.

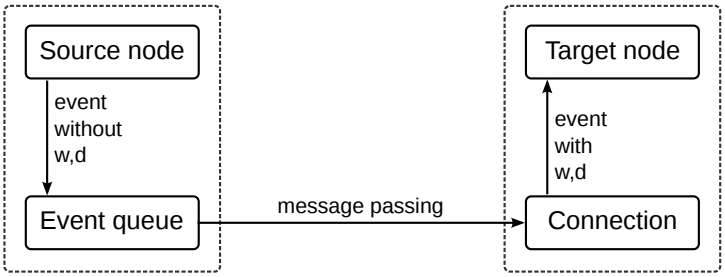


Figure 2.3: *Flow of events from source to target node: An event is created by a source node, but does not yet contain weight (*w*) and delay (*d*). This information is filled in by the Connection object after communication (message passing). The complete event is passed to the target node.*

2.3.4 Storage of connections

In simulations with only a single process, all connection information is available locally. With multiple processes, we have to decide if we want to store this information on the process of the presynaptic node or on the process of the postsynaptic node.

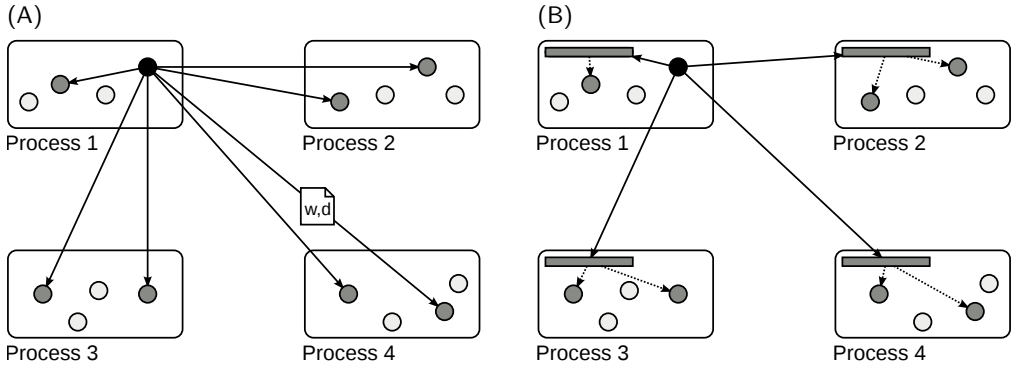


Figure 2.4: *Different possibilities for the storage of connections: (A) The naïve design, where connection data is stored on the process of the presynaptic node. Complete event data (containing the weight and delay of the connection, and the time stamp of event creation) have to be sent to all target nodes. Solid arrows depict inter-process communication. (B) Storage of connection information on the process of the postsynaptic node using a local event queue on each process. Each neuron sends its events to the local event queue, which is broadcasted to all participating machines. Only the id of the sending node has to be transmitted together with the time stamp of event creation. The event can be reconstructed on the receiving process. Solid arrows depict inter-process communication; dotted lines depict local event delivery.*

Figure 2.4 illustrates the consequences for the amount of transferred data for the two basic possibilities to store connection information. Storing the connection information on the process of the presynaptic node, the amount of transferred data scales with the number of targets, whilst it scales with the number of processes, when storing the connection information on the process of the postsynaptic node.

Presynaptic storage

If we used presynaptic storage of connection information, each node has to know its full target list and has to deliver events to all targets. For n nodes, each having a mean of k connections and a mean firing rate of λ, we need to send $\lambda \cdot n \cdot k$ events per process. For k approaching n, which is the case for biological networks below 10^4 nodes and realistic connectivity, this results in $\mathcal{O}(n^2)$ events to be sent.

Postsynaptic storage

If we used postsynaptic storage of connection information, each node sends event information to all processes. The events are stored in an event queue on the receiving process, and can then be recreated, as full connection information is available there. For m processes this results in $\lambda \cdot n \cdot m$ events. Assuming that m is usually much smaller than k, we only need to send $\mathcal{O}(n)$ events per process.

Distributed connection storage

As shown by the analysis above, postsynaptic storage is optimal, if the number of processes is smaller than the mean number of connections per neuron. For biological networks with realistic numbers of connections, this is true up to 10^4 processes. In addition, some types of plasticity (e.g. STDP) need to take into account the spiking behavior of the pre- *and* postsynaptic nodes, which is available naturally on the process of the postsynaptic neuron. Therefore, we decided for postsynaptic connection storage in NEST 2. To reduce the overall amount of communication we could store the id of the processes where a source node has targets on the machine of the presynaptic node and only send events there.

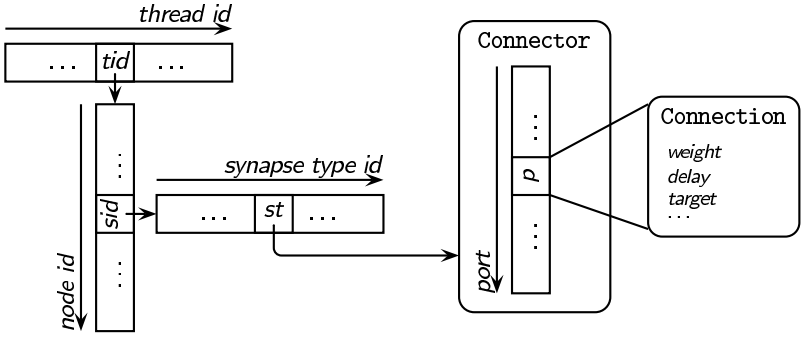


Figure 2.5: *The data structure for connection storage (modified from Eppler, 2006, Figure 4.3): Connections are stored inside a four-dimensional data structure. Each connection is uniquely identified by four numbers: the thread of the target node (tid), the global id of the source node (sid), the synapse type (st), and the port (p).*

Using postsynaptic storage, each node or proxy only stores the subset of connections that reach nodes (but not proxies) on the same virtual process. Thus, the connections are also distributed over all virtual processes. The data structure for the connections is separate from the node list, and has a separate dimension to store the connectivity of each thread, as to minimize cache problems. It is shown in Figure 2.5.

2.3.5 Memory requirement

For a network of n nodes, each requiring S_N bytes of memory, and proxy nodes requiring S_P bytes, the memory required by nodes on each process is given by:

$$M_{\text{nodes}} = \frac{n}{N_{\text{MPI}}} S_N + \left(n - \frac{n}{N_{\text{MPI}}} \right) S_P$$

Typical values are $S_N = 480$ bytes and $S_P = 56$ bytes. Thus, for 10 and more MPI processes, more than half of all memory occupied by nodes is occupied by proxies, and more than 90% for more than 80 MPI processes. In absolute numbers, however, M_{nodes} is only around 70 MB for a network of 10^6 nodes distributed across 100 processes, which is negligible compared to the memory required for the connections in the network. From a performance perspective, a node list filled with mostly proxies could become suboptimal if the node list became so large that it could no longer be held in cache memory efficiently. In this case, a fast hashing look-up may become more efficient (see Section 6.1).

Because the connections dominate the memory requirements of large networks, NEST splits them up such that each virtual process only stores the incoming connections to its own nodes. Each connection consists of at least a pointer to the local target node, along with the delay (integer), and weight (double) of the connection. The memory required for connections per MPI process is:

$$M_{\text{conn}} = \frac{n \times k \times S_C}{N_{\text{MPI}}}$$

where k is the number of outgoing connections per node and S_C the memory per connection. For connections with constant weight and delay, $S_C = 32$ bytes; plastic synapses require more memory. A network of 10^6 nodes with 10^4 connections each thus requires 32 GB connection memory per process if distributed across 10 MPI processes, but only 3.2 GB per process if distributed across 100 processes.

2.4 Network creation

The network in NEST is created by running a simulation script, written in either SLI (the language of NEST's built-in simulation language interpreter), or Python (using PyNEST; see Chapter 3). The structure of such a script depends on the actual aim of the study, but generally consists of the following steps:

1. Creation of nodes (sub-networks, neurons, and devices).
2. Connection of nodes.
3. Simulation.
4. Readout of the recorded data and data analysis.

In the distributed case (i.e. using multiple processes), each process executes the same simulation script. The user does not have to take care of the actual details of distribution, but only has to specify the number of processes to use.

2.4.1 Factories for nodes and connections

Using the function `SetDefaults` users can modify the defaults of the built-in models, and using `CopyModel`, they can create their own models from existing ones. NEST 1.9 supported the reset of the simulation engine. However, it was not possible to restore the original parameters of models in a running session, and NEST had to be restarted to do so.

The creation of nodes and synapses is based on *factories* (Gamma et al., 1994) that contain prototypes of the models. If the user creates a new node or synapse, a copy of the desired prototype model is placed in the node list, or in the connection storage system.

To allow restoring the original model parameters after a reset of the simulation engine, we designed a data structure for models that consists of two parts: the first part contains pristine models without any modifications by the user, while the second part contains copies of all pristine prototypes plus prototypes that were created by the user with `CopyModel`. When a new model is registered with the simulation engine, it is added to the first part of the data structure. Modifications of the parameters and the addition of new models by the user only happens in the second part. When the simulation engine is reset, the models of the second part are deleted and replaced by the ones from the first part. This data structure is used both for neuron and synapse models, which eases the maintenance of the two systems.

2.4.2 Distribution of nodes

In normal operation, the virtual process, a node is assigned to is determined by its global id as explained in Section 2.3.2. However, compound models that consist of a parent node and several child nodes, require the direct manipulation of child nodes by their parent node. An examples for this kind of model is a retina model, which takes responsibility for all the receptor models that are inside of it in order to define global properties like sensitivity, or receptive fields. To support the non-modulo distribution of nodes, sub-networks in NEST 2 have been extended by the new boolean property `children_on_same_vp`, which can be used to force the assignment of child nodes to the same virtual process as the parent node. This means that all algorithms had to be changed to ask nodes for the id of their virtual process instead of assuming a modulo assignment. Another application of this feature is manual load balancing: if activity hot-spots in the network are known, the communication volume can be be minimized by grouping nodes with strong inter-group connectivity.

2.4.3 Compatibility checks for connections

During connection setup, the sending node checks if the receiving node is able to handle the type of event it will send during simulation by calling the receiver's `connect_sender()` function with an event of the respective type. The receiver answers by either assigning a *receptor port* to the connection or throwing an exception. The reason to perform this handshake at setup time is that we do not want to check this for each event during the simulation, as this would considerably slow down the simulation.

In the test phase of NEST 1.9, we observed wrong simulation results if non-spiking devices (e.g. `dc_generator`) were connected to neurons. After an investigation of the problem, we found that the problem was related to the handshake explained above, as it only checked compatibility of source and target neuron, but did not check the compatibility of the `Connection`.

Many plasticity rules (e.g. STDP; see Morrison et al., 2006) update the weight of the connection if a presynaptic neuron fires a spike based on the time of the spike. The rules expect that spikes are sparse and carry all information in their time stamp. However, non-spiking devices transmit an analog value in each time step of the simulation to their targets. If a non-spiking device is connected to a target through a plastic synapse, the synapse erroneously updates the weight in each time step, because it assumes spikes as the means of communication. This kind of error was not prohibited by the handshake of NEST 1.9.

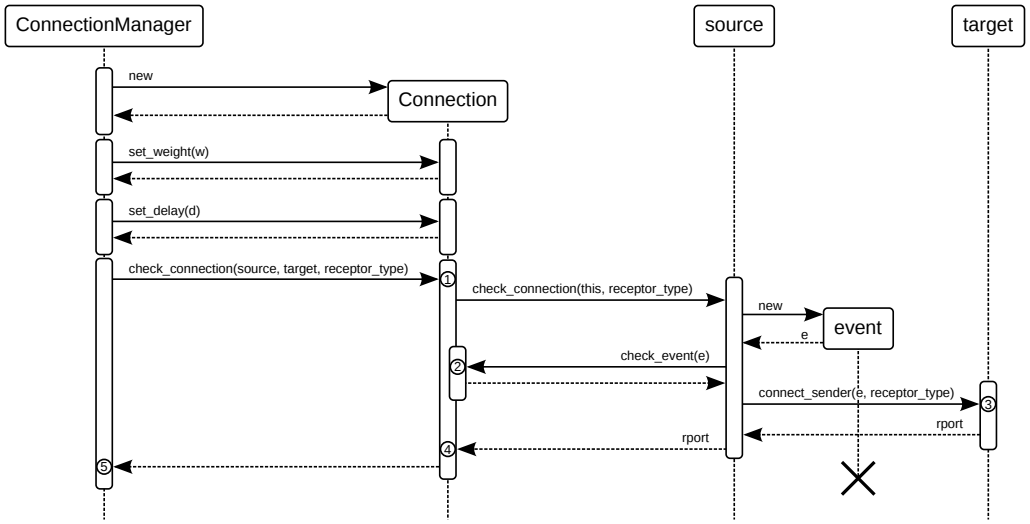


Figure 2.6: *Sequence diagram of the handshake to check node and event compatibility:* ① *Set target for the connection object.* ② *Check if the event type is supported by the connection, otherwise throw* `UnsupportedEvent`*.* ③ *Check if receptor type and event type are supported by the receiver, otherwise throw* `UnknownReceptorType` *or* `IllegalConnection`*.* ④ *Set the receptor port for the connection.* ⑤ *The Connection is now fully established.*

To solve the problem, we designed a new handshake algorithm, which includes a check for the compatibility of the connection with the used event.

The base class for connections was extended by the function `check_event()`, which is present once for each event type in NEST. The implementation in the base class just throws an `UnsupportedEvent` exception. If a synapse type derived from `Connection` wants to support a certain type of event, it can overload the variant of the function for the corresponding event with an empty definition. The sequence of function calls and object creations in the handshake is shown in (Figure 2.6).

2.4.4 Inspection and manipulation of connections

In NEST 1.9, it was complicated to inspect and manipulate connections once they were set up. The main reason for this was the complex data structure for connection storage, which provided no easy way for the user to access connections. However, in studies of synaptic plasticity and learning it is important to have access to the state variables of the connections, and to be able to manually modify the connections according to learning rules.

NEST 1 returned handles to new connections after their creation. Because of the higher memory requirements due to the complexity of the data structure for synapse storage (see Section 2.3.4), we dropped this feature during the development of NEST 1.9. However, a connection could still be uniquely identified by four integer values:

1. The global id of the source neuron.
2. The thread id of the target neuron.
3. The synapse type id of the connection.
4. The port of the connection.

The *port* of a connection is the index of a `Connection` object in the connection list of the presynaptic neuron on a specific thread with a specific synapse type. This means that the port is only meaningful together with all three other values.

NEST 1.9 provides the functions `GetConnection` and `SetConnection` to retrieve and modify connection information. Both functions got the four numbers for the identification of a connection as arguments, while `SetConnection` additionally got a dictionary with the new parameters for the connection. However, this meant that the user had to track the order of connection creation manually in order to know the port of a connection. The following listing contains an short example that illustrates this method to retrieve synapse parameters:

```
1 /iaf_neuron Create /n1 Set
2 /iaf_neuron Create /n2 Set
3 /iaf_neuron Create /n3 Set
4 n1 n2 Connect
5 n1 n3 Connect
6 n1 0 /static_synapse 1 GetConnection /conndata Set
```

Lines 1 to 3 create three integrate-and-fire neurons, `n1` to `n3`. The neuron `n1` is connected to neuron `n2` and `n3` in line 4 and 5, respectively. To query the connection parameters of the connection between `n1` and `n3`, we use the function `GetConnection`. It expects the neuron's global id (`n1`), the thread of the target node (`0`), the synapse type (`static_synapse`) and the port of the connection (`1`). Knowing the port of the connection requires to track the order of connection creation. In simulations with random connectivity this is often impossible.

For NEST 2, we designed a mechanism for inspecting and modifying connections that fits naturally into the already existing framework to inspect and modify the parameters of nodes. The status of nodes in NEST can be retrieved using the function `GetStatus`, which gets a global id as argument and returns a dictionary with the current values of the model variables. `SetStatus` expects a global id and a parameter dictionary as arguments and sets the model variables on the node.

The solution for connections consists of two new language elements for NEST's simulation language interpreter: first, a new data type called `ConnectionDatum`, and second, a new function `FindConnections`. `ConnectionDatums` can be used as arguments for the functions `GetStatus` and `SetStatus` (like global ids for nodes) to inspect and modify the properties of a connection. They encapsulate the four numbers that identify a synapse. `FindConnections` gets a dictionary that contains the source gid, and optionally a synapse type and a target gid as arguments, and returns a list of `ConnectionDatum` objects for the connections that match the given criteria. It does this by performing a complete linear search through the synapse storage

system, constrained by the given arguments. The new solution is much more convenient than the old one and at the same time improves the consistency between the interface functions for nodes and connections, which eases the maintenance of both NEST's code and the model specification of the user. The following listing functionally contains the same example as above, but is using the new mechanism:

```
1 /iaf_neuron Create /n1 Set
2 /iaf_neuron Create /n2 Set
3 /iaf_neuron Create /n3 Set
4 n1 n2 Connect
5 n1 n3 Connect
6 << \source n1 \target n3 >> FindConnections /conn Set
7 conn { GetStatus } Map 0 get /conndata Set
```

Lines 1 to 3 create three integrate-and-fire neurons, `n1` to `n3`. The neuron `n1` is connected to neuron `n2` and `n3` in line 4 and 5, respectively. To query the connection parameters of the connection between `n1` and `n3`, we now use the function `FindConnections` to obtain handles for all connections between `n1` and `n3` in line 6 and retrieve the parameters of the connections in line 7 by applying `GetStatus` to all elements of the list of handles.

2.5 Network update

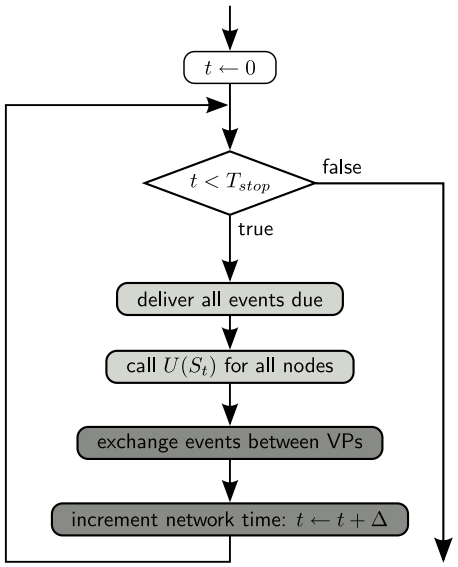


Figure 2.7: *Flow chart of the scheduling algorithm in NEST: The light gray boxes indicate actions that are executed in parallel by all threads, the dark light boxes indicate actions that are executed in parallel by all processes. The simulation starts at t=0 and is executed in a loop that runs until the time specified by the user (T_{stop}) has elapsed. During the simulation, we first deliver all events from the previous update cycle and then update all nodes. After this step, we exchange the event buffers that contain spikes for remote machines. The last step is to advance time by the smallest delay in the network (Δ) and start the loop anew.*

For the simulation of time discrete systems, two basic update schemes exist: *time based* and *event based*. In time based simulations, the system is updated in regular intervals, known as the simulation *time step*. In event based simulations, the elements are only updated when they receive events (Zeigler et al., 2000).

NEST evaluates the network model using a hybrid scheme between event based and time based update. The network elements are updated on an evenly spaced time-grid $t_i := i \cdot \Delta$, where Δ is determined by the shortest transmission delay in the network. This is possible, because the elements are effectively decoupled during this time span and events from other elements cannot influence the state of their targets. At each point in time, the network is in a well-defined state S_i. Starting at an initial state S_0, a global state transfer function $U(S)$ propagates the system from one state to the next, such that $S_{t+\Delta} \leftarrow U(S_t)$.

The network model in NEST is evaluated by executing the loop shown in Figure 2.7. During the execution of $U(S_t)$, nodes may create events that must be delivered to the target nodes after a delay that depends on the connection. The nodes have a local event queue to store the events until they are due.

2.5.1 Event buffering and delivery

Events by devices that are created during the update cycle only need to be delivered to local targets, because devices are replicated in each virtual process (see Section 2.3.2). The replicas are responsible for providing events to their local targets. In contrast to this, the spikes that are created by neurons must be delivered to local *and* remote targets. To make this distinction, the global `send()` function of the network class has to distinguish between events for remote nodes and events for local nodes based on the identity of the sender and the type of event.

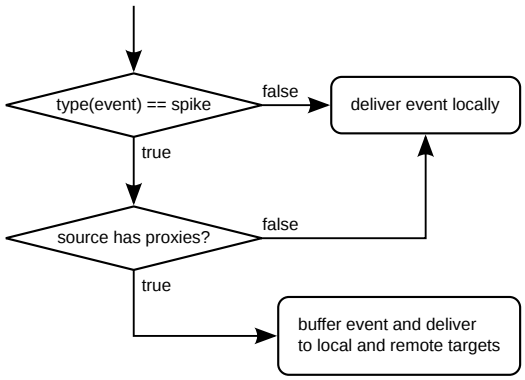


Figure 2.8: *Flow chart of the logic for sending events: If the event is not a spike, it is delivered directly. If the event is a spike sent by a device (a node without proxies), it is also delivered directly. Spikes by neurons are buffered and sent to local and remote targets at the end of the time slice.*

The `send()` function is implemented as a C++ template with two specializations: one variant only accepts non-spike events and delivers directly to local targets. This function is used by non-spiking devices, e.g. current generators. The other variant only accepts spike events and switches between buffering for remote delivery and local delivery. If the source has proxies, this means that it is a neuron and the spikes are buffered locally until the end of the

time slice. If it does not have proxies, it is a device and the spikes can be delivered to local targets directly. The logic for sending an event is shown in Figure 2.8

Note that this design and its current implementation does not support non-spike events to be sent to remote targets, which prohibits the modeling of neurobiological concepts like gap junctions (direct electrical couplings between neurons that require a current flow from one neuron to another) in a distributed scenario.

At the end of the time slice we ensure that all processes are informed about the content of all spike buffers that have been filled during the update phase. To minimize the amount of data sent to remote processes, we only send the id of the sender and markers, that separate spikes originating from different time steps within a time slice of length Δ. After the exchange of the spike buffers, the events are reconstructed on the machines, where the spiking neuron has targets. This reconstruction is possible, because the connection information is stored completely on the process where the receiving node is located. This method for event buffering is described in more detail in Morrison et al. (2005) and Eppler (2006).

Improved low-level communication

Paranel and NEST 1.9 used a custom implementation of the CPEX algorithm (Tam & Wang, 2000; Morrison et al., 2005) on top of the message passing interface (MPI) to perform the pairwise exchange of spike buffers. Because the CPEX algorithm did not scale well for large numbers of processes, we tried to find an alternative algorithm for NEST 2.

As we already used MPI, two obvious candidates were the functions `Allgather()` and `AllgatherV()` provided by the library (Message Passing Interface Forum, 1994). The latter is very efficient, but requires an additional exchange of the buffer sizes before the actual communication of data. Our benchmark simulations showed that the communication bottleneck for our applications is the frequency and the latency of communication, rather than the size of the transmitted packets. This finding lead us to favor `Allgather()`.

However, `Allgather()` requires that all send buffers must have the same size, and the question of how to determine this size arises. The optimal size of the buffers depends on the network size and its activity, which might not be stationary. This means, we need an algorithm that adapts the buffer sizes progressively until they are big enough.

At the beginning of the simulation, the buffers are set to their smallest possible size (enough for all the time-slice markers). In each simulation cycle, each machine collects its spikes in a separate buffer. If the send buffer available is smaller than the size that is needed, it writes an error flag to the first position of the send buffer and the size it requires to the second position, instead of copying its spikes to the send buffer. After this step, `Allgather()` is carried out. After the communication is finished, each machine checks the start of each section of the receive buffer for error flags. If there are any, the size of the send buffer is set to the largest size requested by the machine which reported the error, and the size of the receive buffer is set to the number of machines times the size of the new send buffer. Then each machine copies its spikes to the new send buffer and `Allgather()` is repeated. The flow of the algorithm is shown in Figure 2.9.

Typically, the buffer sizes reach an equilibrium after only a few cycles. No difference in simulation time can be observed between such a simulation, and one in which the buffer size was set to the (experimentally determined) equilibrium size at the beginning of the simulation.

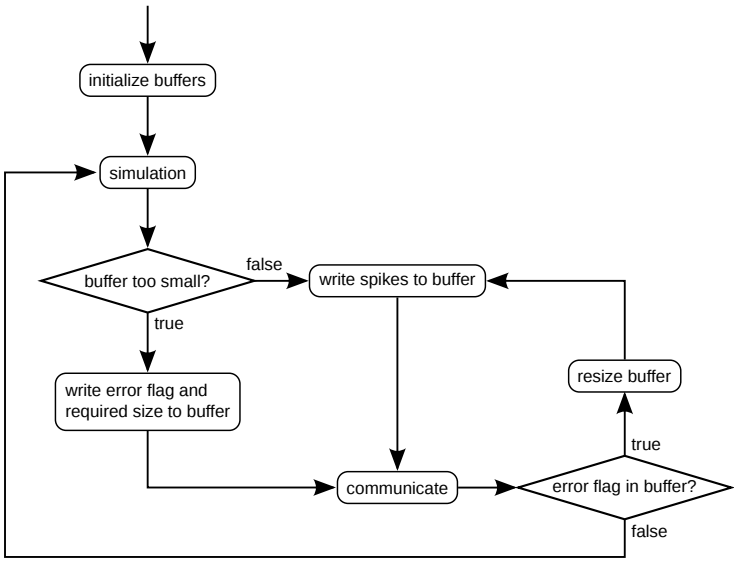


Figure 2.9: *Flow chart of the resizing algorithm for communication buffers: The communication buffers are initialized to provide space for all markers. After simulation each process checks if the buffers are too small. If yes, an error flag and the required size is written to the buffer. Otherwise, the spikes are written to the buffer. After the communication of the buffers, the receiving process checks whether one of the received buffers contains an error flag. If yes, the buffer is re-sized to the maximal requested size and the spikes are written to the buffer and communication is repeated. If no, the simulation continues.*

Besides scaling, another advantage of using a standard function of the communication library is that we do not have to maintain its implementation.

2.6 Readout of data

In NEST 1.9, it was not possible to obtain the data collected by the device replicas with a single call to `GetStatus`. The user had to query each device replica separately by looping over the threads and calling `GetStatus` for each replica. This made it complicated to transfer simulations to machines with a different number of processors, because the simulation scripts had to be adapted for each machine. Moreover, this meant that all functions concerned with devices had to take the thread as additional argument, which made the simulation code more complex and dependent on the number of threads and thus on the details of parallelization. The following listing illustrates this problem with the example of data retrieval from a spike detector in a simulation with two threads:

```
1 /spike_times [] def
2 [0 1] {
3   sd GetAddress exch append GetStatus
4   /events get /times get spike_times exch cva join /spike_times Set
5 } forall
```

Line 1 defines an empty array `spike_times` for storing the spike times received from the spike detector. The list of loop variables corresponds to the thread ids (`[0 1]`), and is defined in line 2. The values are put onto the stack one after the other. Line 3 puts the global id of the spike detector (`sd`) onto the stack and converts it to the address representation using the command `GetAddress`. Addresses are an alternative way to identify nodes and allow the hierarchical addressing of nodes. The command `exch` exchanges the two topmost elements on the stack, so that the thread id comes after the address. The thread id is appended to the address as to obtain an *extended address* that uniquely identifies the device replica. Next, `GetStatus` is called on the extended address in order to obtain the parameter dictionary of the spike detector. Line 4 extracts the event times array (`times`) from the dictionary, converts it to a SLI array (using `cva`), concatenates it with the content of the array `spike_times` and assigns the resulting array again to the variable `spike_times`. The loop is run for each element in the list of loop variables (defined in line 1) by calling `forall` in line 5.

In order to retrieve data from all threads more conveniently, we developed a mechanism for the data collection for NEST 2, which does not require loops in the user code.

The detailed layout of the internal data structures for the storage of recorded data is only known in the concrete implementation of a certain device class. This means that the devices themselves have to collect the data from the replicas on other threads. As the devices do not know the number of threads a priori, the mechanism for data collection has to be dynamic and take into account the real number of threads in a simulation.

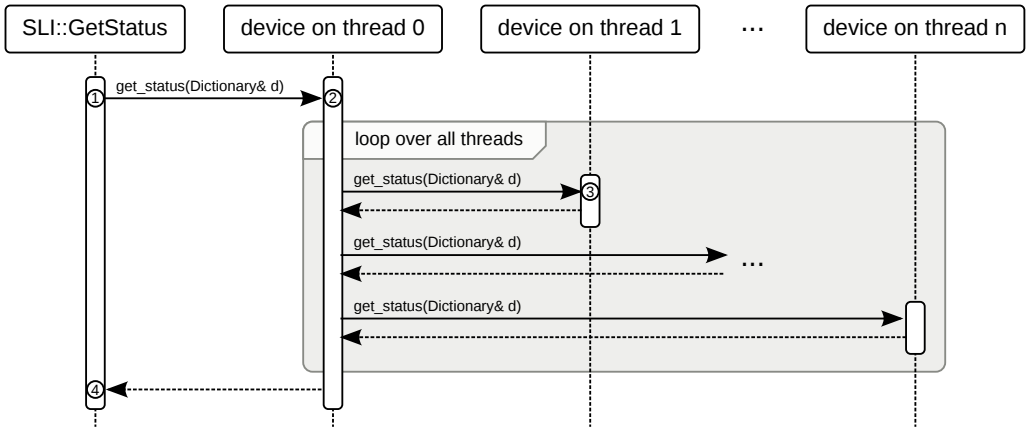


Figure 2.10: *Sequence diagram of the collection of data across threads: ① The SLI function `GetStatus` creates an empty dictionary, `d`. This is handed to the device on thread 0 in the call to `get_status()`. ② The device on thread 0 creates an empty dictionary, called `events`, and adds its own data. It then loops over all other threads. ③ The devices on all other threads just add their data to the `events` dictionary. ④ The `events` dictionary contains the data of all threads and is returned to the user.*

All recording devices in NEST are derived from the common base class `RecordingDevice`. It takes care of the recording interval and provides mechanisms to record the sender and the time stamp of the events it records. A spike detector, for example, only needs to record the data the base class already registers. Recorders for analog data additionally record other variables from the connected neurons (e.g. the membrane potential, or inhibitory or excitatory

conductances). This means that the new mechanism for data collection has to allow an incremental addition of new data fields to the dictionary returned by `GetStatus`.

The previous design of `GetStatus` only returned the data of the device replica on thread 0. The data on other threads had to be retrieved by explicidly passing the thread to `GetStatus`. In the new implementation, a call to the SLI function `GetStatus` calls the C++ function `get_status()` of the respective node on thread 0 with an empty dictionary as argument, which creates an `events` dictionary in the empty status dictionary and populates it with its own data. After that, it iterates over all siblings on other threads and calls `get_status()` on them. These nodes only append their data to the already existing data arrays in the `events` dictionary. At the end of the loop, the data of all replicas is contained in the `events` dictionary in the status dictionary, which is returned to the user. The new algorithm for data collection is shown in Figure 2.10.

Using the new mechanism, the example above can be written in a much more compact way, as the loop is carried out by `GetStatus` internally. A single line of code (compared to five lines with the old mechanism) is now enough to retrieve the event times from the spike detector `sd`:

```
1 sd GetStatus /events get /times get /spike_times Set
```

It is important to note that the temporal sequence of the returned data is not globally maintained. The data of each device replica is sorted by the time stamp of arrival, but the data arrays of all replicas are just concatenated. However, sorting the data after the simulation can be carried out much more efficiently by the user, and additionally provides greater flexibility with respect to the analysis that follows simulation and data collection.

2.7 Benchmark results

We performed benchmark simulations with a random network based on the work of Brunel (2000) to assess the quality of our new algorithms and data structures for different network sizes and different numbers and types of connections. Depending on the computing power of the tested machine, we modified the total run-time of the simulation, and the type of synapses used in the network (static or STDP). Many of the benchmarks in the following sections exhibit super-linear scaling. A theoretical analysis of cache effects explains this and can be found in Section 6.3 of Eppler (2006).

2.7.1 Performance on multi-processor machines

One of the design goals for NEST 2 was to achieve a performance and scaling comparable to the Paranel simulation engine, which showed super-linear scaling in certain situations. The benchmarks to assess the performance and scaling on small multi-processor machines were run on a Sun Fire V40z with 4 Dual Core AMD Opteron 875 processors, each having 1 MB of cache memory and a clock speed of 2.2 GHz.

Figure 2.11 shows the simulation run-time and speed-up of NEST 1 and NEST 2 on a small multi-processor machine. In general, NEST 2 exhibits much better scaling, although the absolute performance of both versions is approximately the same as in the serial case. The improved scaling can be explained by the use of the cache-optimized network representation in NEST 2 (see Section 2.3).

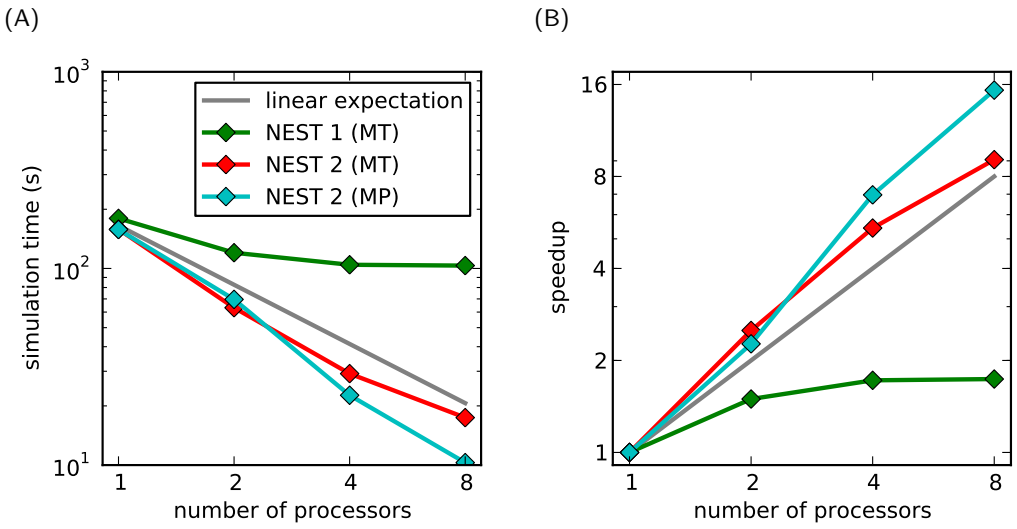


Figure 2.11: *Scalability of NEST 1 and NEST 2 on multi-processor machines (modified from Eppler, 2006, Figure 6.1): The simulated network contained* 10^4 *neurons, with 1000 random connections each; the exact model is described in Brunel (2000). Simulation time is 1 biological second. Green line: NEST 1 using multi-threading; red line: NEST 2 using multi-threading; cyan line: NEST 2 using message passing; the gray line indicates linear speed-up. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speed-up* S_P *with* P *processors against number of processors, log-log representation (*$S_P = \frac{T_1}{T_P}$*;* T_1 *is the serial run-time,* T_P *the parallel run-time with* P *processors).*

An important thing to note here is that the speed-up of the multi-threaded version of NEST 2 is worse than that of the message passing version. On first sight, this contradicts the intuition that a program, which lives in a single memory space and where objects can interact by calling each other's member functions directly ought to be faster than if message passing over a special communication library is used.

However, the explanation for the different scaling is again the processor's cache: multi-threaded programs suffer from much more cache problems (see Section 2.2), because the data structures for the different processors are never completely separated, as they are when using different processes.

2.7.2 Performance on small clusters

To determine the performance and scalability of the new algorithms and data structures on medium-sized clusters, we ran a simulation of a large random network on a cluster with 20 compute nodes, each equipped with 2 AMD Opteron 250 processors with 1 MB cache memory and a clock speed of 2.4 GHz. The computers were connected with a Dolphin/Scali interconnect, and we used the custom MPI implementation by Scali.

Figure 2.12 shows that we achieved our design goal that NEST 2 provides good scaling comparable to the scaling of Paranel. The reason for the difference in absolute performance is due to the simpler design of Paranel's simulation engine compared to NEST.

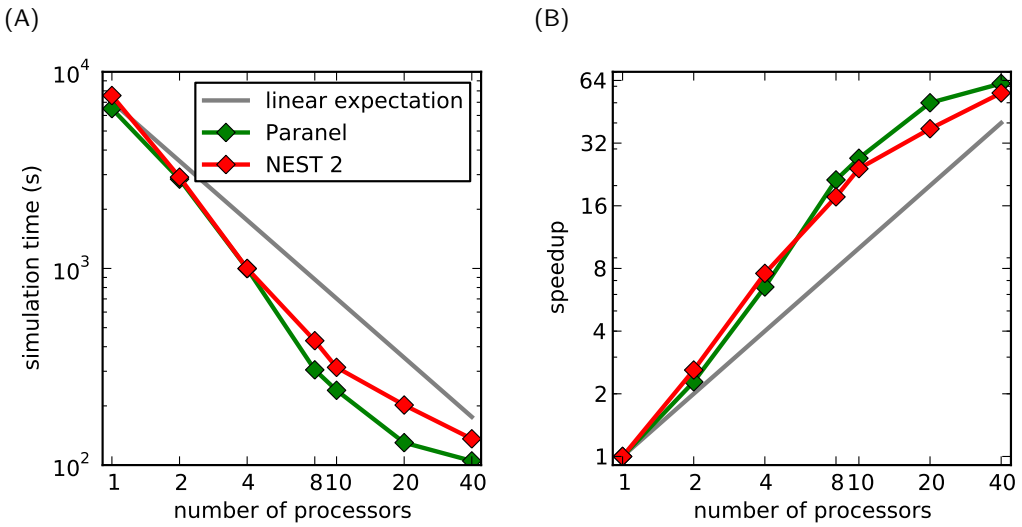


Figure 2.12: *Scalability of NEST 2 and Paranel on small computer clusters (modified from Eppler, 2006, Figure 6.2): The simulated network contained 10^4 neurons, with 1000 random connections each; the exact model is described in Brunel (2000). Simulation time is 50 biological second. Green line: Paranel; red line: NEST 2; the gray line indicates linear speed-up. (A) Simulation time against number of processors, log-log representation. (B) Corresponding speed-up S_P with P processors against number of processors, log-log representation ($S_P = \frac{T_1}{T_P}$; T_1 is the serial run-time, T_P the parallel run-time with P processors).*

2.7.3 Performance on HPC facilities

The algorithms and data structures of NEST 2 were designed for the use on small multi-processor machines and small and medium-sized computer clusters ($<$ 100 compute nodes). However, during the writing of this thesis, much larger machines became available and we tested the implementations on them. In general, our algorithms also show good performance on larger machines. To assess the scaling of the algorithms on large clusters, we performed benchmark simulations on two different HPC clusters:

The BlueGene/L architecture combines two PowerPC 440 processor cores, each with a clock speed of 700 MHz, together with main memory, a cache system and a communication subsystem on a board. The boards can be combined without introducing bottlenecks. The machine we used is located at the Riken Brain Science Institute in Wako-shi, Japan.

The BlueGene/P architecture is similar to the architecture of the BlueGene/L, but uses four PowerPC 450 cores at 850 MHz per board. The machine we used (JUGENE) is located at the research center in Jülich, Germany and ranks number 4 in the November 2009 issue of the TOP500 supercomputer list (`http://www.top500.org/list/2009/11/100`).

Figure 2.13 shows that the algorithms of NEST 2 also scale linearly up to 1024 processors on very large clusters, although the algorithms were not designed for this kind of machines.

However, the performance drops when going to even more cores: in order to reduce the computing time by one half we have to use four times as many processors if we go beyond 1024

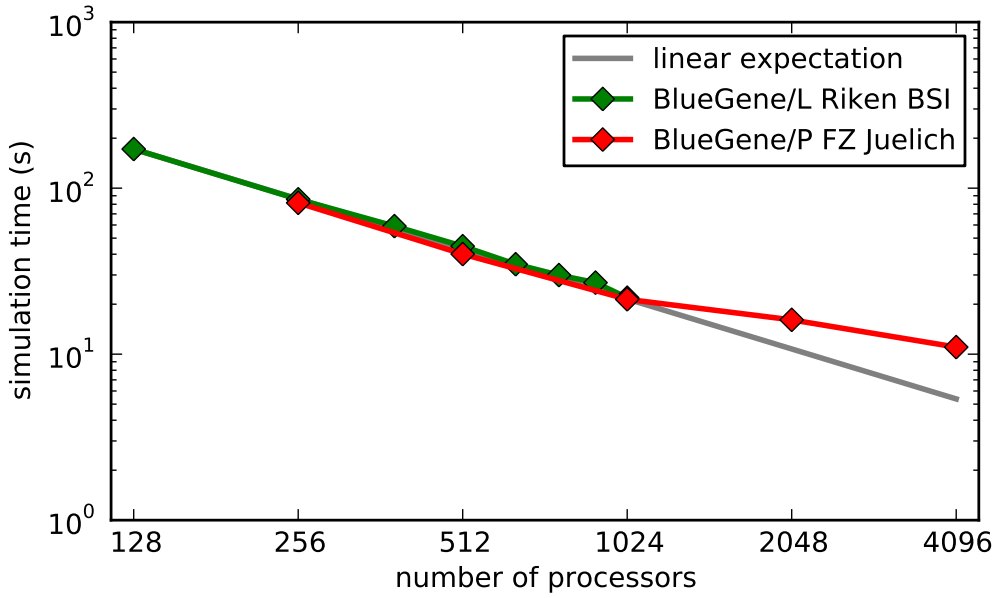


Figure 2.13: *Scalability of NEST 2 on large computer clusters: The simulated network contained 112,500 neurons, with 11,250 random STDP connections each; the exact model is described in Morrison et al. (2007). Green line: BlueGene/L; red line: BlueGene/P; the gray line indicates linear speed-up. Simulation time against number of processors, log-log representation.*

cores. This can be understood by considering that 1024 cores means that each processor core is assigned less than 100 neurons, and only about one million connections. As a consequence, the computational cost for updating the neurons is dropping below the cost for communication and the positive effects of the cache on the overall simulation time and scaling disappears.

2.8 Summary

This chapter summarized the design of the algorithms and data structures for the simulation engine of NEST 2. Although the described methods are tailored to the problems and preconditions found in this simulator, they are general in the sense that they solve problems that occur in the design of all simulators for spiking neural networks.

NEST 2 provides the possibility to run simulations of large biological neural networks on multi-processor machines and on computer clusters. It supports the use of heterogeneous neuron and synapse models in the same network. Using cache-optimized data structures for the network representation and a hybrid simulation kernel with threads locally and message passing on the cluster, we observe excellent scaling even on very large facilities for high-performance computing, e.g. IBM's BlueGene.

Many neuroscientific studies depend on the run-time and the scaling of the simulations in a crucial way. This means that many studies would not have been possible without the improvements presented here. Examples are the investigation of plasticity and learning (Mor-

rison et al., 2006; Potjans et al., 2009), the analysis of signal flow in large structured networks (Schrader et al., 2009; Potjans & Diesmann, 2008b), the cross-validation of the neuromorphic hardware developed in the FACETS project against simulation results (Brüderle et al., 2007), models of working memory (Gewaltig, 2009) and many more. For a detailed list of publications carried out with NEST, see the homepage of the NEST Initiative at `http://nest-initiative.org/index.php/Publications`.

The use of factories for the creation of nodes and connections allows to create custom models from existing ones. With the improved design of the factory classes in NEST 2, the usability of these classes became easier for the user, while at the same time reducing the maintenance effort for the developers.

The removal of the implicit synapse context for all functions associated with handling connections and the improved handshake during connection creation make it easier to write correct code and contribute to the elimination of many errors in the simulation code that are hard to find.

With NEST 2, we succeeded to hide the implementation details of multi-threading from the user, who only has to set the desired number of threads before setting up the network. This constitutes a major improvement over previous versions of NEST, as it makes it easier to run the same simulation on different machines. In the current release of NEST, it is possible to inspect and modify synapses after they have been established using an interface that is consistent with the interface to inspect and modify neurons and devices, and to retrieve the data from devices without the need to think about threads. This improves the readability of simulation descriptions, which eases the independent reproduction of simulations by others.

Chapter 3

Communication between user and simulator

Chapter 2 introduced the simulation engine and the technology behind NEST, and explained how to efficiently simulate large-scale biological neural networks with a great number of neurons and realistic connectivity. However, pure simulation performance is only one component of an efficient simulator. The second component is nicely illustrated by the following quote from Wilson (2006):

> *Increasingly, the real limit on what computational scientists can accomplish is how quickly and reliably they can translate their ideas into working code.*

This quote highlights the importance of a usable and convenient user interface to ease the work of the researchers who use the software. Since its first version, NEST provides a built-in simulation language interpreter (*SLI*) for the interactive setup of the network and the control of the simulation engine. Because of the large number of elements involved in network simulations of spiking point neurons, this has proven advantageous over a graphical user interface. However, SLI is a stack machine, which means that functions expect their arguments on an operand stack and return their results back to this stack. As a consequence, all operations have to be formulated in reverse polish notation (Burks et al., 1954). This makes SLI hard to learn and use for novices. Moreover, SLI uses a custom language tailored to the concepts of NEST. New users have to learn both, the concepts of NEST and the language of SLI in order to use NEST, and they asked for a more convenient user interface.

In addition to lowering the initial barrier for new users, a new and more convenient user interface for NEST could provide the chance to simplify the task of writing model descriptions, and make it easier to share these descriptions with other researchers. On the other side, a new interface has to support legacy code that has been written for SLI over the last years.

During the development of NEST, different alternative interfaces for the simulation engine have been designed and implemented. Examples of a successful coupling to other programming languages are bindings for Tcl/Tk (Diesmann & Gewaltig, 2002), Mathematica, Matlab, IDL and Java. However, none of these interfaces was supported by a large user community and their development was discontinued often already after a short time. One decisive advantage of SLI is that it does not depend on third-party software and allows NEST to be completely self-contained and independent.

3.1 Languages for neural simulation specification

In the early days of computational neuroscience, standard simulators were not available to the researchers in the field. The question of user interface design and the choice of an appropriate simulation language did thus not arise. Each researcher wrote custom code for the problem at hand either in a general purpose language like C/C++ or Java, or in a scientific programming environment like Matlab (MathWorks, 2002) or Mathematica (Wolfram, 2003). This approach, however, had several drawbacks:

- The simulation code was written to answer one scientific question, and thus could not be used for other studies.
- The code was often hard to read and hard to maintain by other people, as these goals were not the aim of the authors. As a result, many simulators were abandoned after the end of the scientific project they were built for.
- The independent reproduction of simulations was often complicated as the simulation code was only rarely published with the model.
- The correctness of the simulator code was unknown, because an independent review was not possible.

This approach to writing simulators did not scale well beyond the current problem, as the simulators of that time were not designed to be sustainable software products, but rather were written to address one specific scientific question. A central problem of this approach was code quality and maintainability. Many of the authors were neuroscientists with basic programming skills, but without a proper computer science background. This means that standard methods and techniques for common problems were not used and brute-force solutions were often favored over a careful design of the software.

When the field matured, some simulators emerged as the de-facto standard tools in specific problem domains and were used by a larger community. Prominent examples are NEURON (Hines & Carnevale, 1997) and GENESIS (Bower & Beeman, 1997) for the simulation of detailed morphologically realistic compartmental models, and NEST (Gewaltig & Diesmann, 2007) and CSIM (Natschläger, 2003) for the simulation of large networks of spiking point neurons. The source code of all these simulators is available electronically, which allows an independent control of quality and of the correctness of the algorithms used.

Although this second phase in the development of computational neuroscience had many advantages over the previous era, the different simulators still used different user interfaces. The simulators that were written in scientific programming environments like Matlab and Mathematica used the user interface components of these environments as their primary interface, while the other simulators had a custom graphical user interface based on an external library, or used a custom programming interface.

Introducing a dependency on third-party software for the user interface has the disadvantage that the simulator can only be used on platforms where the user interface library runs and that the fate of the simulator is coupled to the fate of the library.

A summary of the history of simulation languages in computational neuroscience is contained in Davison et al. (2010). The following section contains a detailed description of NEST's user interface.

3.1.1 The simulation language interpreter of NEST

NEST's built-in simulation language interpreter (SLI) allows the interactive control of NEST's simulation engine. Originally it was designed as an intermediate language layer between the simulation kernel and a new language, which supports the formulation of model descriptions using neuroscientific concepts and terminology. However, this new language was never implemented, and SLI itself was used for the specification of simulations.

SLI is a stack machine and uses a syntax based on PostScript (Adobe Systems Inc., 1999). The term SLI is used both for the interpreter and for its programming language. SLI is a multi-paradigm language that allows to use object-oriented, procedural, and functional language constructs in a single program. Flexible data structures like heterogeneous arrays and dictionaries (*named parameters*; Finkel, 1996) together with powerful operators like `Map` enable a compact formulation of algorithms. However, due to the reverse polish notation SLI uses, the language is hard to read. The syntax of SLI is exemplified by the following listing, which defines a procedure to calculate the alpha function given by $\alpha(t, \tau) = t \cdot e^{-t/\tau}$:

```
1 /alpha
2 {
3   /tau Set
4   /t Set
5   t neg tau div exp t mul
6 } def
```

The first line contains the name of the procedure. All literal names in SLI start with the token `/`. Procedures are enclosed in curly brackets (line 2 and 6). Line 3 and 4 store the arguments to the function in variables called `tau` and `t`, respectively. Please note the reverse order of assignment due to the stack semantics. The actual value of the alpha function is calculated in line 5. The procedure is assigned to the name `alpha` using the keyword `def` in line 6. A call to the function takes two (numeric) arguments from the stack and returns the result back to the stack. If the type of the arguments are incompatible with the functions used inside the procedure, an error is raised.

In principle, SLI can be used as a general purpose scripting language like Bash (`http://www.gnu.org/software/bash`) or Python (`http://www.python.org/`). It provides modules for random number generation, string processing, file system access, and has a comprehensive library of mathematical functions and operators, which makes SLI a powerful tool also outside the simulation of neural systems.

SLI has a modular architecture, where dynamically loaded extension modules can add new data types and functions by registering them with the simulation engine. Fundamentally, the NEST simulation engine is only one of many modules, which adds functions for the simulation of neural networks to SLI. This includes functions and data types for the creation and manipulation of neurons, devices and connections, and for the control of the simulation engine. The following example shows the creation, parametrization, connection, and simulation of two neurons in SLI:

```
1 /iaf_neuron Create /n1 Set
2 n1 << /I_e 1500.0 >> SetStatus
3 /iaf_neuron Create /n2 Set
4 n1 n2 Connect
5 100.0 Simulate
```

Line 1 and 3 each create a neuron of type `iaf_neuron` (a simple integrate-and-fire neuron) and assign its global id to a variable for later reference. Line 2 sets the unspecific background current of neuron `n1` to 1500 pA using the function `SetStatus`. The tokens << and >> are used to define a dictionary, which contains the parameters as key-value pairs. The neurons are connected in line 4. Line 5 simulates the network for 100 ms.

Implementation

The SLI interpreter uses two stacks to operate: the *execution stack* and the *operand stack*. The first is used as internal data structure for the interpreter only, and is hidden from the user. The operand stack holds user data.

The execution of commands in SLI is based on a main execution loop. In each cycle, the interpreter fetches one token from the execution stack and calls a function, which depends on the token's type. For most token-types, e.g. numbers, strings and other data, the function will simply move the token from the execution stack to the operand stack. Other tokens are executable and the function will execute them appropriately.

Two types of executable tokens exist: SLI procedures and C++ functions. The SLI procedures consists of a collections of SLI expressions delimited by the tokens { and }. An example is the procedure `alpha` shown above. The second type of executable tokens are functions written in C++. They have full access to the data structures and functions of NEST and SLI and can retrieve data from the operand stack using member functions of the interpreter.

If the next token on the execution stack is a C++ function, it will be called directly. If the token is a SLI procedure, it is executed, by executing each token of the procedure in sequence. If all tokens of the procedure are executed, the procedure token is removed from the execution stack and the interpreter continues with the next token.

Normal functions and procedures take their arguments from the operand stack and also return their results back to this stack. In addition there are internal functions which operate on the execution stack. These are used to execute procedure, loops and conditionals.

At start-up, the execution stack contains only one token. It is the parser which reads characters from the standard input, translates them into SLI tokens and pushes them on the execution stack. This token is then the next to be handled by the interpreter. After it has been executed, the interpreter will again find the parser on its stack, and the cycle continues. Once the execution stack runs empty, the interpreter will return.

An extensive description of the design and use of SLI is contained in Diesmann et al. (1995). More sophisticated examples for neural simulations with NEST can be found along with the source code of NEST and on the homepage of the NEST Initiative at `http://www.nest-initiative.org/`.

3.1.2 Towards a general language for computational neuroscience

Many simulators for spiking neural networks provide a programmable interface to allow the script-driven setup of the neural network and the convenient exploration of parameter spaces. However, each of them uses a different programming and configuration language. For the large-scale projects introduced in Section 1.6, this is a major problem, because they often rely on multiple simulators, and a lot of effort is put into the development of tools to integrate the data from different simulators and recordings from real neuroscientific experiments.

The members of FACETS (http://facets.kip.uni-heidelberg.de/) promoted the use of Python as a general glue language to solve this problem and researchers started to create tools for stimulus generation, the control of experiments and simulations, databases for the results, and tools for data analysis in this programming language.

This development was sparked by the need for a consistent code base and inspired by a strong trend towards Python in the scientific community in general (Dubois, 2007), although Python was almost unknown in computational neuroscience at that time.

NEST was one of the main simulators used in the FACETS project, and we evaluated the language based on the experience of other researchers in the project. Python has a number of advantages over commercial programming environments like Matlab or Mathematica:

- It is free software, developed and supported by an active community with members inside and outside of science.
- It is installed by default on almost all Linux and MacOS based computers.
- It has a large user base outside of science and thus also provides packages for other purposes such as databases, multimedia, graphics, and networking.

On the other side, the availability of packages for scientific computing (http://www.scipy.org/) and plotting (http://matplotlib.sourceforge.net/) make Python also a good alternative to other free and open source programming languages like Tcl/Tk (Ousterhout, 1994) or Perl (Wall et al., 1996). Compared to many other languages, Python has a very readable and concise syntax and provides a higher expressiveness (Prechelt, 2000), which helps researchers to go from an idea to working code in less time. This means that a Python interface for NEST could improve the productivity of its users, and at the same time ease the readability of model descriptions.

Considering the advantages of Python above other languages and taking into account the current trend towards this language in the computational neuroscience community, we decided to explore the possibilities of a new user interface for NEST based on the Python programming language.

3.2 A Python based user interface for NEST

The usual approach to creating Python bindings for existing software is to create a wrapper library that exposes all data structures and functions of the application to Python. This can either be done manually, using Python's C-API (van Rossum, 2008), or automatically using an interface generation tool like the Simplified Wrapper and Interface Generator (*SWIG*; http://www.swig.org/) or an external library like Boost.Python (http://www.boost.org/).

However, this approach restricts the flexibility of the interface as it only allows to call functions as they are and thus leads to problems regarding error handling and access control for the functions and objects. This also means that it is not possible to extend the functions in an easy way, or to use high-level definitions like the ones already present in SLI.

Another drawback of the common approach is that data conversion is carried out using the standard mechanisms of the wrapper generator. As the generator does not know anything about the data types in SLI, it cannot optimize the data conversion.

With SLI, NEST already has a well-tested and stable interpreter for the setup and control of neural simulations. Instead of exposing the classes and functions of NEST directly to Python, we decided to deviate from the usual approach and instead keep the existing interpreter as an intermediate layer between the Python based user interface and the simulation engine. The reason for this is threefold:

1. A lot of SLI code has already been written, and we did not want to render this code useless with newer versions of NEST by abandoning SLI. Moreover, many of NEST's built-in functions are written in SLI. This also includes the testsuite and the basic library components.
2. Python is not yet available on some "exotic" hardware platforms, which we still need to use. As SLI is completely self-contained and thus runs on all of these platforms, it still allows the usage of NEST on them.
3. As a long-term project, NEST needs to remain independent of third party software to guarantee its sustainability. This can be achieved by keeping NEST's source code separate from Python code, and avoid hard dependencies on Python.

Based on these observations, we implemented a first version of Python bindings for NEST. The prototype implementation, called *PyNEST*, that resulted from this work, had a minimal interface consisting of only three main functions:

`sli_push()` pushed a Python data type as SLI data type onto the SLI stack. Initially, the function only supported the conversion of the basic data types `int`, `float`, `double`, `bool`, and `string`.

`sli_pop()` returns a SLI data type from the stack as a Python data type to the user. The function is thus the inverse of `sli_push()` and supported the same set of basic data types.

`sli_run()` executes a command in SLI. The command has to be given to the function as a string. As SLI's parser was invoked with the string as argument, the string could contain all tokens supported by the stand-alone version of SLI.

These functions allow the complete control of the NEST simulation engine from within the Python interpreter. However, the commands still had to be formulated in SLI's own language, and given as arguments to the function `sli_run()`. This meant that the users still had to learn and use this language in order to use PyNEST. To solve this problem, the low-level API described above was accompanied by a set of high-level functions to provide Python wrappers for the most important functions in SLI and NEST.

The main advantage of using an interface generator like SWIG is that it eases the task of creating the bindings. However, the low-level API of PyNEST only consists of three functions that also contained the code for the conversion of data. Moreover, these functions do not have to be modified when changes in NEST occur, so the effort for developing and maintaining this level of the PyNEST API is minimal. In addition, we know the implementation of the data types in SLI and can thus provide optimized routines for the data translation between SLI and Python.

3.2.1 Problems in the prototype

The prototype implementation of PyNEST demonstrated that it is, at least in principle, possible to build a Python interface for NEST on top of the already existing infrastructure. However, its use during a comprehensive test phase revealed several shortcomings that resulted mainly from the fact that it was created in a very short time. The following paragraphs contain a detailed analysis of the shortcomings.

The performance of the Python bindings was not good enough for the use in the day-to-day work of researchers. This weakness especially showed when large amounts of data were transferred from Python to SLI or vice versa. According to application run-time profiling and other benchmarks, two main reasons were responsible for the low performance of the interface:

- The transfer of data from SLI to Python allowed only basic data types to be sent. Complex data types had to be sent element-wise and re-assembled by the receiving interpreter, which resulted in many function calls and thus too much overhead. To eliminate this problem, a special data type for SLI, called `PyDatum`, was implemented as a wrapper around an ordinary Python object. Neuron models that had to use complex data types from Python or that wanted to return complex data types to Python had to support this new data type, which introduced a hard dependency on Python in NEST. Moreover, the SLI functions themselves were not able to work with this data type, which severely limited the use of the interface.
- The data conversion routines were based on cascades of type checks and dynamic casts to find the correct type of Python and SLI objects and carry out the conversion. This solution was expensive and made the performance of type conversions depend on the order of the checks.

The high-level API that contains Python wrappers for SLI functions was not complete and many tasks still required the use of SLI code in the simulation description scripts of the users. This meant that although users of PyNEST could use Python to run their simulations, they still had to learn SLI in order to take full advantage of the capabilities of NEST. Moreover, the semantics of the functions was not consistent over the set of functions in the high-level API. The `Create()` function for example returned a complete list of identifiers to the new nodes, while `Connect()` expected single node identifiers as arguments.

Another major problem of the prototype was that error conditions in NEST and SLI were not propagated to the Python level. This meant that errors often went unnoticed, which resulted in wrong simulation results and errors in the model description that were hard to find. On the other side, errors in the implementation of the PyNEST APIs themselves were not easily detected, as support for formal unit tests was missing from the prototype version of the Python bindings.

The Python extension had to be compiled and linked to NEST manually. This was a problem, as the build-time settings of NEST were not taken into account in PyNEST, or they had to be transferred to the build process of the Python bindings manually.

In spite of all problems and restrictions, the Python bindings were used in the FACETS project by several researchers. The reaction to them was mainly positive, and we decided that we wanted to use them as the basis for a new user interface of NEST.

3.2.2 Requirements for a Python based user interface

Based on the prototype implementation explained above, we went back one step in the design, and made a formal analysis of the requirements for Python bindings for NEST. The following requirements are crucial for the long-term sustainability of NEST and for the acceptance of the new interface by a broad user community.

Independence

The NEST code base has to be kept independent of Python. This is important for machines where Python is not available, and to keep NEST's fate independent of the fate of Python.

This was violated by the introduction of `PyDatum` in PyNEST 1. A proper solution would use SLI's own data types for all communication between the two interpreters. This, however, requires a way for the efficient transfer of large amounts of data (e.g. random neuron parameters) from Python to SLI and back (e.g. connectivity data).

Freedom of choice for the interface

It has to be easy for the user to choose between the interfaces SLI and PyNEST for simulations. This is important to provide the users with greater flexibility and to allow them to chose the interface that best suits their research needs.

This requirement is closely related to the requirement for independence. However, in addition it requires an integration of the build process of PyNEST with the build process of NEST in a way that a deactivation of PyNEST is easily possible.

Backwards compatibility

It has to be possible to run legacy SLI code without porting it to Python. This is important, because a lot of published models were already in written in SLI.

This requirement is already fulfilled by the prototype implementation of PyNEST. Using the function `sli_run()` to call the SLI function `run` with the appropriate script as an argument allows to use legacy code without further work in the interface. The only restriction of this method is that neuron ids are not available in Python if the simulation is fully set up by an external SLI script.

Extensibility

The extension of the high-level interface of the Python bindings has to be easy. This is important to keep the maintenance complexity of the bindings to a minimum. At the same time we must allow extension modules for NEST to integrate their functions into the interface.

The separation of PyNEST into a low-level API (containing the functions `sli_push()`, `sli_pop()`, `sli_run()`, and the routines for data conversion) on one side, and a high-level API with Python wrappers for SLI function on the other side, allows an easy addition and modification of high-level functions without the need for a recompilation of NEST or PyNEST. This means that this requirement is also satisfied by the prototype. However, as the prototype had to be built and installed manually, it was not possible to automatically install the Python components of extension modules for NEST.

Testability of correctness

NEST and SLI themselves come with an extensive suit of unit tests. They provide an important means to guarantee that the simulator and its interface are working as expected and that the results obtained with it can be trusted. The prototype implementation of PyNEST did not have any unit tests.

To test the correctness of the high-level functions and to check if they carry out the same operations as their SLI equivalents, a testsuite for PyNEST is indispensable. Such a testsuite should use Python's mechanisms to write unit test, but it should also be integrated with the testsuite of NEST and SLI to allow a convenient check of all software components involved in a simulation.

Error handling

It is important to propagate errors and exceptions from the C++ and SLI level of NEST and SLI to Python, in order to inform the user about problems in the simulation description. However, it is not enough to print out the error messages, as simulations are often run in loops and thus errors would still go unnoticed. Proper error handling has to convert the errors in NEST and SLI to Python exceptions, which the user can check for and handle appropriately. This requires the design of an error handler in SLI, which catches errors and translates them into the respective Python exceptions. These can then be handled without knowledge of the error handling mechanisms of NEST and SLI.

Performance

It is clear that the performance of PyNEST is lower than that of SLI, as it constitutes another software layer that involves function calls and additional parsing. Building the networks often has a major share in the total run-time of the simulation of a neural network, so the overhead should be sufficiently small.

However, the simulation time is only a minor part in the development of the model. The major part is taken by the conception of the model and we can tolerate longer simulation time, if the translation of an idea is faster by providing a more convenient interface.

3.3 The architecture of PyNEST

Although the current version of PyNEST is based on the original prototype explained above, it constitutes a major improvement over earlier versions in that it solves all of the problems explained above. The following sections contain a detailed description of the design and implementation of the interface.

Just as the prototype implementation, the current version of PyNEST is only a light-weight wrapper around SLI rather than a complete wrapper around NEST's classes and functions. The basic architecture of PyNEST is shown in Figure 3.1. PyNEST consists of two separate layers: The low-level API is responsible for the basic interaction between Python and SLI, and provides functions for the data conversion from SLI to Python and back. The high-level API provides Python wrappers for the most important functions in SLI and NEST.

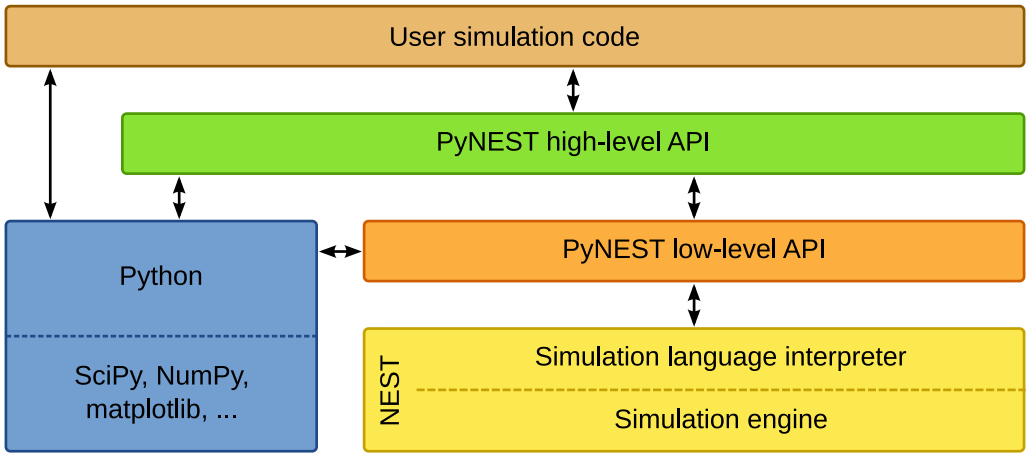


Figure 3.1: *The architecture of PyNEST: The lowest level is the simulation engine. It is used by the simulation language interpreter and by the PyNEST low-level API. The PyNEST high-level API uses the low-level API to communicate with the simulation engine. The user's simulation code can use functions from PyNEST, from Python, and from its extension modules.*

3.3.1 The low-level API

The low-level API of PyNEST is implemented using the Python C API (van Rossum, 2008), and calls the respective functions of SLI to push data onto the stack (function `sli_push()`), to retrieve data from the stack (function `sli_pop()`), and to invoke SLI's parser on strings to execute commands (function `sli_run()`). In addition it contains routines for the conversion of Python objects to SLI data types (used by `sli_push()`) and for the conversion of SLI data types to Python objects (used by `sli_pop()`).

This interface allows to run NEST simulations from Python by calling the respective functions in SLI. The following listing repeats the example from Section 3.1.1 using this approach:

```
1  sli_run("/iaf_neuron Create")
2  n1 = sli_pop()
3  sli_run("/iaf_neuron Create")
4  n2 = sli_pop()
5  sli_push(n1)
6  sli_push({"I_e": 1500.0})
7  sli_run("SetStatus")
8  sli_push(n1)
9  sli_push(n2)
10 sli_run("Connect")
11 sli_push(100.0)
12 sli_run("Simulate")
```

The neurons are created in line 1 and line 3 of the listing. Their identifiers are assigned to the Python variables `n1` and `n2` in line 2 and 4. Line 5 and 6 push the arguments for the `SetStatus` function to SLI, which is called in line 7. The neurons are connected, by first pushing their global ids to the stack in SLI, and then calling the function `Connect` in line 8. Finally, the network is simulated for 100 ms by pushing the time in line 9 and calling the function `Simulate` in line 10.

Although this method allows to carry out a complete simulation in NEST from within the Python interpreter, it is neither convenient, nor compact (twelve lines of code versus only five lines using pure SLI). To remedy this situation, PyNEST contains a set of high-level functions for common tasks.

3.3.2 The high-level API

SLI already provides all necessary commands to build and simulate a neural network. As demonstrated above, calling the SLI functions directly, using the low-level API, is not very convenient. However, we can create a more convenient Python interface to NEST by creating wrapper functions that call the respective functions in SLI. The high-level API uses the functions of the low-level API to provide Python versions of all important SLI commands. The functions of the high-level API can then be used by the user's simulation code. In general, each function wrapper that is part of the high-level API stereotypically consists of three basic parts:

1. Push the arguments onto the SLI stack with `sli_push()`.
2. Execute one or more SLI commands to perform the desired action inside of NEST using `sli_run()`.
3. Retrieve the results from the stack via `sli_pop()`.

An example for a function that realizes this structure is the high-level implementation of `Create()`. A simplified version of the function is shown in the following listing:

```
1  def Create(model, n=1, params=None):
2      sli_push(n)
3      sli_run("/%s exch Create" % model)
4      lastid = sli_pop()
5      ids = range(lastid - n + 1, lastid + 1)
6      if params:
7          sli_push(params)
8          sli_run("/params Set")
9          sli_push(ids)
10         sli_run("{ params SetStatus } forall")
11     return ids
```

Line 1 contains the function's signature with the mandatory parameter `model` and the optional parameters `n`, the number of nodes to create, and `params`, a parameter dictionary to use for the new nodes. The number of nodes is pushed to the SLI stack in line 2. The command that is executed in line 3 first creates a literal name from the given model name, exchanges the two topmost elements on the stack as to create the right order of arguments, and finally calls SLI's `Create` function. Line 4 then retrieves the global id of the last created node. Using this id, the full list of global ids is created in line 5. If a parameter dictionary is given, it is used to set the parameters of the newly created nodes by calling `SetStatus` on each of them (lines 7-10). Finally, the list of global ids is returned to the user in line 11.

Note that the call to SLI's `SetStatus` function in line 10 could also be replaced by a corresponding high-level function of PyNEST.

If we now assume that we have high-level functions for the creation and connection of nodes and for the simulation of the network, we can re-formulate the example from above as follows:

```
1 n1 = Create("iaf_neuron", params=[{"I_e": 1500.0}])
2 n2 = Create("iaf_neuron")
3 Connect(n1, n2)
4 Simulate(100.0)
```

This formulation is much more compact than the one we had previously, and which only used calls to the low-level API, but also more compact than the original formulation in SLI.

Array semantics of PyNEST

The implementation of the high-level function `Create()` shows a basic principle of the PyNEST API: all functions that operate on node handles expect lists of global ids, although the underlying functions in SLI may only operate on single node ids. Further examples for this are the functions `Connect()`, `SetStatus()`, `GetStatus()`, and many others. The reason for this approach is that it allows to minimize the number of data transfers from Python to SLI and thus to improve the performance of PyNEST.

The implementation of the `SetStatus()` function is a good example for the application of this technique:

```
1 def SetStatus(nodes, params):
2     sli_push(nodes)
3     sli_push(params)
4     if params == types.DictType:
5         sli_run("/params Set")
6         sli_run("{ params SetStatus } forall")
7     else:
8         sli_run('2 arraystore')
9         sli_run('Transpose { arrayload pop SetStatus } forall')
```

Line 1 contains the function's signature. The argument `nodes` is a list of one and more global ids as returned by `Create()`, while `params` can be either a single parameter dictionary, or a list that contains one parameter dictionary for each node and thus has to be the same size as `nodes`. The lines 2 and 3 push the arguments to SLI. Line 4 checks if `params` is of type `DictType`, in which case the dictionary is assigned to the SLI variable `params` and set for each of the given nodes. Else, we assume `params` to be a list and create an array of the two arguments `nodes` and `params` in SLI using the function 2 `arraystore` in line 8. Line 9 first transposes the array and then calls the procedure {`arrayload pop SetStatus`} for each element of the new array. The function `arrayload` in SLI resolves an array and puts the elements onto the stack. It then puts the number of elements onto the stack. This number is discarded by `pop`, which removes the topmost element of the stack. The function `SetStatus` takes the global id and the dictionary from the stack and sets the properties of the node.

Compared to the naïve design, this version has a better performance. The reason is that it only needs to push the parameter dictionary once, where the naïve version needs to push it n times for n nodes. Even in the case of n different dictionaries, the expected performance is better, as less function calls are involved.

The `Connect()` function is implemented in a similar way. The PyNEST version allows to make arbitrarily many one-to-one connections with a single function call, where SLI's basic `Connect` routine only can connect two nodes with each other:

```
1 group1 = Create("iaf_neuron", 10)
2 group2 = Create("iaf_neuron", 10)
3 Connect(group1, group2)
```

Extending the functionality of SLI

The availability of powerful functions for plotting in Python through Matplotlib (`http://matplotlib.sourceforge.net/`) allows us to extend the functionality of PyNEST over that of SLI. PyNEST contains two modules for plotting the data that results from the simulation: the modules `voltage_trace` can be used to plot membrane potential traces of arbitrarily many neurons, `raster_plot` allows to plot spike data with or without a temporal histogram. The following listing contains an example usage of the `voltage_trace` module.

```
1 from nest import *
2 from nest import voltage_trace as plot
3 n = Create("iaf_neuron", params={"I_e": 500.0})
4 vm = Create("voltmeter", params={"withgid": True})
5 Connect(vm, n)
6 Simulate(100.0)
7 plot.from_device(vm)
```

Line 1 imports all functions from the NEST high-level API to the global namespace. The `voltage_trace` module is imported under the name `plot` in line 2. The simulated network consists of a single neuron with an unspecific background current of 500 pA (created in line 3) and a `voltmeter` to measure the membrane potential of the neuron (created in line 4). Line 5 connects the device to the neuron. In line 6, the network is simulated for 100 ms. The data collected by the `voltmeter` is plotted in line 7. The result of the simulation is shown in Figure 3.2:

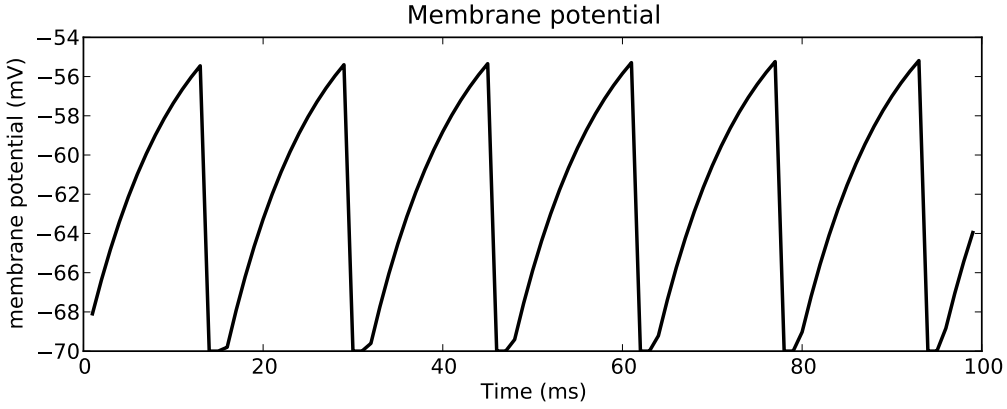


Figure 3.2: *Example of plotting with the `voltage_trace` module: The blue line shows the membrane potential of a single neuron stimulated by a direct current input with 500 pA.*

3.4 Data conversion

PyNEST needs efficient routines for the data conversion from Python to SLI, and from SLI to Python. This conversion is possible, because each data type in SLI has a corresponding data type in Python. However, the data types for both interpreters are implemented in different ways and in different programming languages (Python uses C, while SLI uses C++), so we need different approaches for the two directions of the conversion.

The data conversion of the prototype implementation suffered from severe performance problems and introduced a hard dependency on Python in the NEST source code (see Section 3.2.1). Our new design of the data conversion routines effectively solves both problems.

3.4.1 From Python to SLI

The function `sli_push()` has to convert a given Python object to the corresponding SLI data type. In general, all Python objects are available in C as pointers to objects of type `PyObject`. To distinguish between the types, the C-API of Python provides type-check functions to test if a Python object is of a specific type. These functions return `true` if the object is of the queried type, and `false` otherwise. Based on these functions, we designed the function `PyObject_ToSLIDatum()` for the conversion of a Python object `PyObj` to the corresponding SLI data type. It first determines the type of the Python object and then instantiates a new SLI datum of the correct type, and with the correct value. This is shown in the following listing using the example of the conversion of integers and booleans:

```
1  if (PyInt_Check(PyObj))
2  {
3    if (PyObj == Py_True)
4      return new BoolDatum(true);
5    else if (PyObj == Py_False)
6      return new BoolDatum(false);
7    else
8      return new IntegerDatum(PyInt_AsLong(PyObj));
9  }
```

The `if` statement in line 1 checks if the given Python object `PyObj` is of type integer. As booleans in Python are also implemented as integer values, we first check if `PyObj` has the value `Py_True` or `Py_False` in line 3 and 5, in which case we return a `BoolDatum` with the corresponding value. For ordinary integer numbers, we return an `IntegerDatum` with the value of the Python object in line 8.

Similar conversions are available for the conversion of floating point numbers and strings. The conversion of the compound data types dictionary, list, and tuple are carried out element-wise by using the function `PyObj_ToDatum` recursively.

Conversion of NumPy arrays

NumPy (`http://numpy.scipy.org/`) is an extension module for Python, which provides efficient multi-dimensional data types for representing arrays and matrices in Python. As most of SciPy's routines for scientific computing are based on NumPy, it is important to support these data types in PyNEST.

When the PyNEST prototype was written, SLI did not have any efficient numeric array data types. Because of this, NumPy arrays had to be translated element-wise to the SLI data type `ArrayDatum`, which is implemented as a linked list of tokens. However, this was very inefficient and lead to the development of `PyDatum` for SLI, which was a wrapper around NumPy arrays that could be directly transferred to SLI. The major drawback of this approach was that SLI itself was not able to operate on these data types, which hampered the interaction between the two interpreters.

The design of an efficient conversion for NumPy array required the addition of two new data types for SLI: `IntVectorDatum` and `DoubleVectorDatum`. These are implemented as wrappers around C++ `vectors` of the corresponding types and are thus very efficient compared to the linked lists that are the basis for the `ArrayDatum` data type. Using the new data types allows to implement the conversion of the NumPy array types by directly copying the memory content of the NumPy data type to that of the corresponding SLI vector data type.

NumPy arrays support a technique called *slicing*, which allows to efficiently define subsets of arrays without modifying the internal representation of the data in memory. An examples for the usage of this technique is given in the following listing:

```
1 a = numpy.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
2 print a[:5]
3 numpy.array([1, 2, 3, 4, 5]
```

Instead of copying the requested subset of the data, slicing only changes the indexing of the NumPy array internally, and, although the array looks as if only the selected elements were present from within Python, the complete data is visible if the array is accessed from the C-API of NumPy.

In order to support "sliced" arrays in addition to "normal" NumPy arrays in PyNEST, we first have to check if the array is a sliced array or not. If it is, we have to move through it with the correct step size (given by `array->strides[0]` in the C-API of NumPy) and copy the elements one by one to the SLI vector of the corresponding type. Otherwise, we can proceed as explained above.

To improve the integration of the types `IntVectorDatum` and `DoubleVectorDatum` in SLI, we have modified all synapse and neuron models to accept these types in addition to the old list-based types. However, most of SLI's mathematical functions still expect the old `ArrayDatum`, and a conversion is needed.

3.4.2 From SLI to Python

The function `sli_pop()` has to convert a SLI datum to a Python object. This conversion has a more elegant implementation than the conversion from Python to SLI, because it can exploit the fact that each SLI datum knows its own type. A SLI datum thus can convert itself to a Python object of the right type. However, this would introduce a dependency on Python in NEST and violate the requirement of independence stated in Section 3.2.2. To avoid this dependency, we use the acyclic visitor pattern (Alexandrescu, 2001).

The data conversion framework consists of two parts: first, a Python-unspecific part, which is implemented in the source code of NEST, and second, a Python-specific part, implemented in the source code of PyNEST.

Python-unspecific classes and functions

All data types of SLI are derived from a common base class, called `Datum`. For the conversion of SLI datums to Python objects, this class was extended by the function `use_converter()`, which takes a reference to a `DatumConverter` and is inherited by each of the derived data types. The implementation of the function `use_converter()` is shown in the following listing:

```
1  void Datum::use_converter(DatumConverter &converter)
2  {
3    converter.convert_me(*this);
4  }
```

If the function is called on one of the derived data types, it calls the `convert_me()` function of `converter` that matches its own type with itself as an argument.

The class `DatumConverter` serves as the base class for the implementation of a concrete type converter. It contains one pure virtual `convert_me(T&)` function for each SLI data type `T`. The following listing shows this for some of the types in the original implementation:

```
1  class DatumConverter
2  {
3    public:
4      virtual void convert_me(DoubleDatum&)=0;
5      virtual void convert_me(IntegerDatum&)=0;
6      virtual void convert_me(BoolDatum&)=0;
7      // convert_me() functions for all other data types of SLI
8  }
```

The functions and classes for the conversion of SLI types that were described until now, are not specific for a conversion to Python objects. They can be used for any type conversion from a SLI data type to some arbitrary type not known to NEST. The following section describes the part of the conversion framework that is Python-specific and resides in the source code of PyNEST.

Python-specific classes and functions

The Python-specific part of the conversion is implemented in `DatumToPythonConverter`, a class which is derived from the base class `DatumConverter`. It has only a single data member, called `py_object`, to store the result of the conversion. In addition it contains the function `convert()`, which converts a SLI datum `d` to a corresponding Python object by calling `d`'s `use_converter()` function with itself as argument. The implementation of `convert()` is shown in the following listing:

```
1  PyObject* DatumToPythonConverter::convert(Datum &d)
2  {
3    d.use_converter(*this);
4    return py_object;
5  }
```

To carry out the conversion of a concrete SLI data type, the `DatumToPythonConverter` contains implementations of the `convert_me()` functions for all supported SLI data types. These implementations store the result of the conversion in the variable `py_object`, which is

returned by the `convert()` function. The following listing illustrates the conversion with the example of SLI's data type for floating point numbers, `DoubleDatum`:

```
1  void DatumToPythonConverter::convert_me(DoubleDatum &d)
2  {
3    py_object = PyFloat_FromDouble(d.get());
4  }
```

To convert a SLI data type `d` to the corresponding Python object, `sli_pop()` creates a new instance of the class `DatumToPythonConverter` and calls its `convert()` function with `d` as argument. The result of the conversion is returned to the user. A complete example for the order of events during the conversion of a `DoubleDatum` to the corresponding Python object is shown in Figure 3.3.

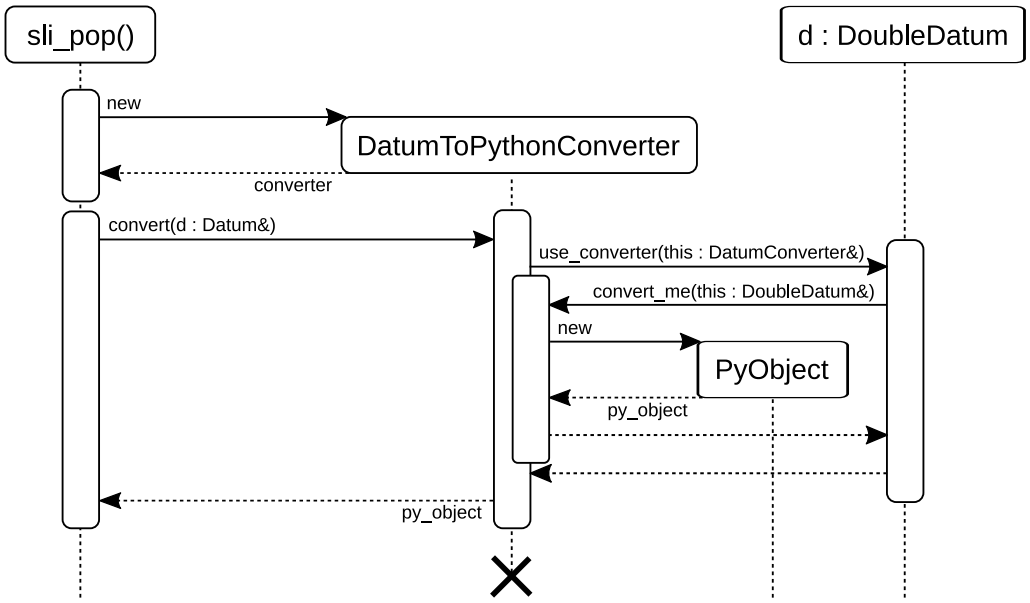


Figure 3.3: *Conversion of a `DoubleDatum` to a Python object: For the conversion of a SLI datum `d`, `sli_pop()` creates an instance of `DatumToPythonConverter`. It then calls the `DatumToPythonConverter`'s `convert()` function, which passes itself as a visitor to the `use_converter()` function of `d`. `Datum::use_converter()` calls the `DatumToPythonConverter`'s `convert_me()` function that matches `d`'s type. `convert_me()` creates a new Python object from the data contained in `d`. The new Python object is returned to `sli_pop()`.*

3.5 Error handling

NEST uses two different error handling strategies: *assertions* to check the correctness of state variables in the program flow during execution (implemented using the C macro `assert()`), and *exceptions* to catch user errors, like wrong arguments to SLI functions and the use of invalid object identifiers. The user errors have to be translated into Python errors, as to inform the user of PyNEST about problems in the simulation description.

Exceptions are generated at the location where an error is detected. Upon an exception, the normal execution of the function is aborted and the exception is propagated up in the calling hierarchy until an appropriate statement catches and handles it. If no such statement is in place, the general error handler of SLI catches it and signals the error to the user.

SLI commands executed from within Python via `sli_run()` are run in a protected environment that uses SLI's mechanisms to catch errors in order to create Python exceptions in the case of errors. The protected environment consists of two parts: First, the function `sli_run()` in the high-level API of PyNEST, and second, the SLI function `runprotected`. The implementation of `sli_run()` is shown in the following listing:

```
1  def sli_run(cmd):
2      kernel_sli_run("{%s} runprotected" % cmd)
3      if not sli_pop():
4          errname = sli_pop()
5          message = sli_pop()
6          cmdname = sli_pop()
7          raise NESTError(errname + " in " + cmdname + message)
```

Line 1 contains the function's signature. In line 2, a SLI procedure is created from the given command `cmd` and executed by `runprotected` using the function `kernel_sli_run`, which is the implementation of the low-level API and provides direct access to SLI's interpreter. The function `runprotected` either returns `True` if the command was executed without errors, or `False` otherwise. If an error occurred, `runprotected` puts the name of the error, the name of the failed command and the error message on the stack, which is retrieved by Python in line 4 to 6 and converted to a Python exception of type `NESTError` in line 7.

The function `runprotected` provides a mechanism for the protected execution of SLI commands and to catch errors. Its implementation is shown in the following listing:

```
1  /runprotected
2  {
3    stopped dup
4    {
5      errordict /newerror get
6      {
7        errordict /message known
8        { (: ) errordict /message get join errordict /message undef}
9        { () } ifelse
10       errordict /errorname get cvs
11       3 2 roll
12       errordict /newerror false put
13     }
14     {
15       (Software Interrupt)
16     }
17     ifelse
18   } if
19   not
20 } def
```

Line 3 executes the given SLI procedure in a `stopped` context (PostScript; Adobe Systems Inc., 1999). In case of an error, `stopped` leaves the name of the failed command on the stack

and returns `true`, and `false` otherwise. The return value is duplicated using the command `dup` to be returned to Python later. If an error occurred, `runprotected` checks if the `newerror` flag in the `errordict` is set in line 5. If yes, the error is caused by a failed command and the information about the error is extracted from the `errordict` in line 7 to 12. Otherwise, the error is caused by a software interrupt. Finally, the return value of `stopped`, which is on top of the stack, is inverted and used by `sli_run()` to check if an error occurred or not.

Using this error handling strategy allows to leave error handling to SLI. The functions of PyNEST's high-level API do not have to check their arguments and can thus be written in a very compact form. An additional advantage of this strategy is that the error messages of pure NEST and PyNEST are consistent without requiring additional effort by the developers.

3.6 Installation and build process

PyNEST uses a build process based on the `distutils` package (`http://docs.python.org/distutils/`). This package allows to build and install extension modules for Python on a large variety of platforms. NEST and SLI, on the other side, use the GNU Build System (`http://www.gnu.org/software/hello/manual/automake/GNU-Build-System.html`) for the build and installation process.

To integrate both systems, we designed a custom `setup.py` script, which is processed by the build mechanism of NEST to set the correct paths (as set during the configuration of NEST), compiler flags, library options and such. This build system guarantees a tight integration of PyNEST with NEST, while at the same time allowing to disable PyNEST on machines where Python is not available or when it is not needed.

3.6.1 Support for NEST extension modules

NEST can be extended by dynamically linked modules, which can add new neuron, device, and synapse models to the simulation engine, and new functions to SLI. To make the functionality of these modules also available in PyNEST, they have to provide function wrappers for their functions.

The installation procedure of NEST has been modified such that the content of the module's sub-directory `pynest` is installed as a sub-module to PyNEST. This means that modules in NEST can be used naturally as sub-modules in PyNEST. For example, to use the *topology module* from within PyNEST, it suffices to execute the command `from nest import topology` in Python.

3.7 Unit tests

NEST itself provides a extensive suite of unit tests in order to guarantee the absence of errors in its implementation (Eppler et al., 2009). This is an important prerequisite for scientific software, and allows the researchers that use NEST to rely on the correctness of its algorithms. During the design of PyNEST, we used simple unit tests for the high-level API to test if the results are the same as that obtained with the underlying functions in SLI and NEST. The testsuite of PyNEST is based on the `unittest` module and is automatically run by NEST's testsuite.

3.8 Performance

One of the critical problems in the prototype of PyNEST was its performance. During the re-design process for the new version of PyNEST, we found that the overhead for function calls can be neglected. Therefore we do not need full function wrappers in the low-level API and the functions `sli_push()`, `sli_pop()`, and `sli_run()` are sufficient. Our benchmarks showed that the real bottleneck is the data conversion.

Partially, this problem could be solved by the two new data types `IntVectorDatum` and `DoubleVectorDatum` for SLI that allow the efficient translation of NumPy arrays into the corresponding vector type of SLI and vice versa by abandoning the element-wise conversion of array elements, which is only needed for lists, tuples, and dictionaries.

Another important prerequisite to achieve good performance in the PyNEST interface is to minimize the number of data conversions. This can be achieved by using collective operations that push data only once to the stack and use SLI operations to work with it. For an illustration of the technique, see the implementation of `Create()` and `SetStatus()` in Section 3.3.2. The magnitude of the saving is shown by the examples in the following two listings that both add up a sequence of numbers in SLI. The first creates the sequence of numbers in Python, pushes them to SLI one after the other and lets SLI add them. Executing it takes approximately 15 seconds on a laptop with an Intel Core Duo processor at 1.83 GHz.

```
1 sli_push(0)
2 for i in range(1, 100001):
3     sli_push(i)
4     sli_run("add")
```

The second version computes the same result, but instead of creating the sequence in Python, it is created in SLI:

```
1 sli_run("0 1 1 100000 { add } for")
```

Although Python loops are about twice as fast as SLI loops, this version takes only 0.6 seconds, because of the reduced number of data conversions and, to a minor extent, the repeated parsing of the command string and the larger number of function calls in the first version.

3.9 Summary

We have shown an alternative approach for creating Python bindings for an application by using a generic interpreter-interpreter interaction interface instead of directly wrapping of the underlying functions and data structures. The complete API reference of PyNEST is contained in Eppler et al. (2009).

One of the design goals for PyNEST was to keep NEST independent of Python in order to allow the user to select between the two user interfaces SLI and PyNEST, and to keep the fate of NEST independent of the fate of Python. This goal was achieved, by providing an easy way to disable PyNEST during the configuration of NEST.

The design of the data conversion routines explained above does not introduce any dependencies on Python in the NEST code base. Moreover, it provides a general framework to implement conversions of SLI data types to arbitrary types that NEST does not know. Being

based on efficient arrays, the performance problems of the conversion from Python to SLI in the prototype could be solved.

Using SLI as the basis for the new user interface allows to run code that has already been written in the language of the old user interface seamlessly from the new user interface. Moreover, the library components of NEST and SLI can still be used from the new interface. SLI' ability to handle errors in the user code relieves the author of high-level functions from the necessity to check the arguments of functions, because they are checked by SLI and reported to the user of PyNEST without additional effort.

The integration of the testsuites of both interfaces improves the trust of users into the correctness of both the functions in SLI and the implementation of the high-level interface of PyNEST. This is an important prerequisite for the correctness of the scientific results obtained with NEST.

With the separation of PyNEST into a stable low-level API and a flexible high-level API we reduce the effort for the maintenance of the binary interface to the minimum. At the same time, it is simple to extend the high-level API by new functions and by new modules for plotting, data analysis, and for the support of extension modules for NEST.

Our choice of Python as programming language for the new user interface of NEST is supported by the fact that a growing number of other neuroscience laboratories are using Python as well (Kötter et al., 2009). Moreover, many researchers in the field are developing tools for stimulus generation and data analysis in this language, and we can benefit from the growing ecosystem that is only just starting to grow around Python.

One example for the synergy achieved by this development is the common interface for different simulators (PyNN), which is explained in the next chapter.

Chapter 4

A common interface for different simulators

In recent years, several simulators for biological neural networks have been developed independently by a number of neuroscience laboratories to address their specific research interests. Each of these simulators is specialized on a certain problem domain, such as spiking neuron and network models (Gewaltig & Diesmann, 2007; Pecevski et al., 2009), detailed compartmental models with rich morphology (Hines & Carnevale, 1997; Bower & Beeman, 1997; Ray & Bhalla, 2008), or reaction-diffusion models (Kerr et al., 2008; Wils & De Schutter, 2009). The availability of multiple simulators has several advantages compared to a software monoculture:

- As each simulator is specialized on a different problem domain, it is possible to chose the most appropriate simulator for a given problem.
- Simulation results can be compared with those obtained with other simulators. This allows a validation of models, which results in a greater confidence in the simulation results.
- The implementations of the simulators themselves can be compared in order to cross-validate their correctness.
- The competition between developers leads to faster progress in the field as more ideas are developed and tested in parallel. The increasing number of publications in the field of neuroinformatics is clear evidence for this.

However, the diversity also leads to problems, as each simulator uses its own programming or configuration language (cf. Section 3.1). The major problem is related to the communication between researchers using different simulators, in particular the exchange of formal model descriptions, which has hindered the independent reproduction of results and the re-use of model components developed by others for a long time. Considerable time has been spent porting models from one simulator to another.

In addition, the diversity in the simulator landscape makes it more difficult to understand and compare model written descriptions for different simulators and by different people. This complicates the independent reproduction of results by others and to build on their work, which is an important prerequisite for scientific progress and for the credibility of the results.

4.1 PyNN: An abstraction layer for simulators

One consequence of the large-scale neuroscience projects introduced in Chapter 1 is the need to use multiple simulators and tools. The reason for this is that only large system-level models of the brain allow to understand brain function as a whole. One example is the EU project FACETS (Fast Analog Computing with Emergent Transient States; `http://facets.kip.uni-heidelberg.de/`), in which a variety of simulators have been used. To ease the cooperation between the different working groups, we developed a common interface to the different simulators (*PyNN*; Davison et al., 2008, available from `http://neuralensemble.org/trac/PyNN/`), which allows the same model description to be run on all supported simulators as well as on the neural VLSI hardware created in the project.

As explained in Chapter 3, the Python programming language is becoming the de-facto standard in computational neuroscience and is used as a general purpose language in the field of neuroinformatics. Most of the current simulators now support Python, either as their primary interface or as an alternative in addition to their original interface (Kötter et al., 2009). This provides an unprecedented opportunity to define a common interface to multiple simulators using Python. PyNN (pronounced "pine") is both a specification of such a common simulator interface and an implementation of the interface for several simulators.

With PyNN it is possible to write a simulation script once and run it without modification on any supported simulator. Thus, we keep the advantages of having multiple simulators (for cross-validation, etc.) but lower the translation barrier. PyNN is now in world-wide use and plays an important role in the INCF program "Multi-scale modeling".

4.2 The architecture of PyNN

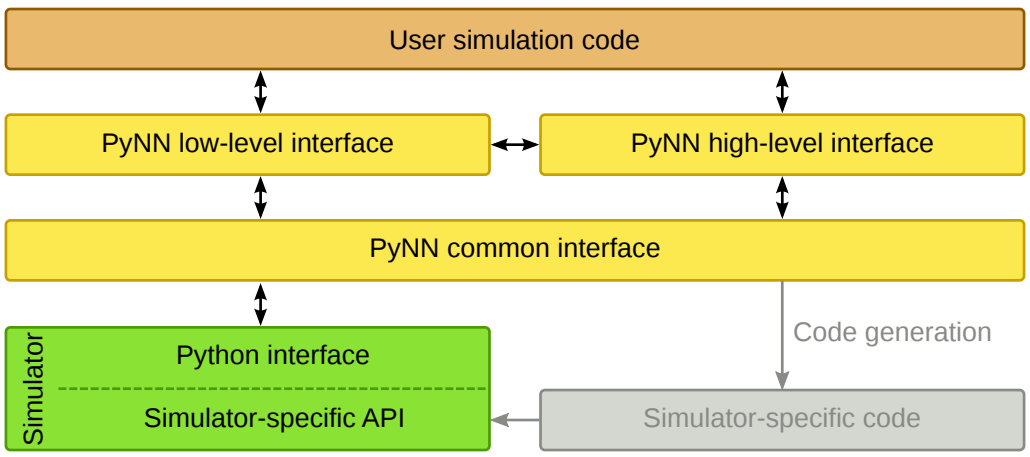


Figure 4.1: *The architecture of PyNN: The user's simulation code can use the high-level and/or the low-level interface of PyNN. They both use the common interface of PyNN, which contains a simulator-specific part (backend) for the simulators that provide a Python interface. The gray parts showing code generation to generate native code for the simulator-specific API of the simulator are planned, but not yet implemented.*

PyNN provides two APIs for the simulation of spiking networks. The low-level interface is implemented in a procedural manner and provides functions to create, connect, modify, and record from single neurons. This interface can be used for porting a simulation script from a simulator-specific language to PyNN, as most of the simulators use a procedural approach. The high-level interface is object-oriented and provides two classes of objects (`Populations` and `Projections`) to build network models using the terms and concepts from neurobiology. The elements within `Populations` can be accessed individually and the functions of the low-level interface can be applied to them. This allows a very flexible combination of the two interfaces in a single simulation script. The basic architecture of PyNN is shown in Figure 4.1.

To support simulators with a Python interface, PyNN uses simulator-specific backends for implementing the simulator-nonspecific part of its interface. In the simplest case, the backend only consists of conversion tables for the translation of function names and physical units to PyNN' standard names and units. This approach works well if the concepts of the simulator are close to the ones in PyNN. In other cases, however, the translation to the simulator-specific interface is more complicated, because the concepts of the simulators differ considerably from the ones in PyNN. For simulators without a Python interface, PyNN supports the possibility to create a model description in the native language of the simulator.

As of writing, the simulators NEURON, NEST, PCSIM, Brian, and the neuromorphic VLSI hardware of the FACETS project have fully supported backends in PyNN. Support for MOOSE (a successor to GENESIS 2) is currently under development.

4.2.1 Simulator-specific backends

The backends for PyNN basically consist of two parts: first a set of functions to initialize the simulator, to run the simulation, and the classes `Populations` and `Projections` of the high-level API. These classes allow the formulation of model descriptions in terms of neuroscientific concepts, which makes PyNN accessible to neuroscientists without a computer science background. Second, Python modules for the different parts of the PyNN API:

cells contains a mapping of PyNN standard cell names to simulator specific names and translations of PyNN parameter names to the parameter names used in the simulator. The cell module is used by the high-level and the low-level API.

connectors contains `Connector` classes that define the parameters of specific connectivity patterns (e.g. all-to-all, small world, or distance dependent). These `Connectors` are used by the `Projection` class of the high-level API.

electrodes contains the classes to simulate current injections into neurons, and methods to connect them to neurons. The electrodes correspond to the electrodes used in real neuroscientific experiments.

recording contains classes and methods for managing recordings of spikes, potentials, etc. and maps the recording API of PyNN to the simulator-specific way of reading out model variables.

simulator contains the definition of the low-level API specific to the simulator. This includes the state of the simulator as well as functions to create and access individual cells in the low-level API.

synapses contains the mapping of PyNN standard synapse names to the synapses of the simulator and conversion tables for the translation of parameter names. This module is to synapses, what the module *cells* is to neuron models.

4.2.2 Unified data format

Each of the different simulators uses a specific data format for the output of recorded data. In large-scale projects like FACETS, this poses an additional problem, as this leads to the use of different tools for the analysis of data depending on the simulator used, or the tools have to be adapted to provide import filters to read all of the data formats.

To ease this problem, PyNN supports a common output format for spikes, conductances and membrane potentials that is created after the simulation by reading the output files of the simulator and rewriting them in the common format. This common format also bridges the gap between real neuroscientific experiments and simulations by allowing to use the same tools for data analysis.

4.2.3 Random number generators

Many neuroscientific studies rely on random connectivity for some parts of the network, or on random input to emulate input from an external population that is not modeled explicitly. Random number generators (*RNG*) thus are an important part of a common interface for different simulators. In order to guarantee reproducible results even if different simulators are used, PyNN provides RNGs that can be used for the setup of random connections, or to initialize random spike trains for stimulating the network.

However, drawing random numbers in Python and transferring them to the simulator is often not very efficient. Moreover, most simulators support random connectivity and random input as a built-in feature. Therefore, PyNN provides a flag to switch between PyNN's RNG, which is slow, but allows to obtain reproducible results, and the one of the simulator, which is fast, but leads to non-reproducible results.

4.3 Benefits of using PyNN

PyNN aims to increase the productivity of researchers in the area of neuronal network modeling in two ways:

1. By providing the capability to model at a high level of abstraction using concepts commonly used in neurobiology, while still allowing access to low-level details of the simulation where necessary. This allows to develop a simulation from an idea faster than by only using low-level concepts such as neurons and single connections. It also improves the maintainability of the simulation code, as PyNN often allows a more compact formulation of ideas.
2. By promoting code sharing and re-use across simulator communities, PyNN simplifies the process of porting models between simulators, which helps to validate the results of others and build on their work.

PyNN changes the process of porting a model from one simulator to another from an all-or-nothing task, in which the validity of the translated model can only be tested when the entire translation is complete, to an incremental approach, in which the native code is gradually replaced by simulator-independent code. At each stage, the hybrid code remains runnable, and so it is straightforward to verify that the model behavior has not been changed by the translation process.

By providing support for populations and projections directly in the language interface, PyNN allows to formulate simulation descriptions in terms of neuroscientific principles. This is in contrast to many of the simulator-specific APIs, which require to formulate the studies in computer scientific terms like objects, classes, and control structures.

Through the multitude of third-party modules for the Python programming language, PyNN can directly benefit rapidly from the straightforward integration with other components such as graphical interfaces, databases, stimulus generation, and data visualization and analysis tools.

4.4 Performance

In general, PyNN follows the philosophy of maximizing compatibility and reproducibility between simulators. This is always the default, but it is usually possible to exchange reproducibility for performance by setting a flag. One example is the `compatible_output` flag, which activates or deactivates re-writing the output of the simulator into the unified data format (see Section 4.2.2). Another example is the `parallel_safe` flag, which is used for turning on or off PyNN's random number generators in a distributed setup.

It is clear that adding multiple layers of user interfaces on top of each other will slow down the network setup compared to the native interfaces of the simulators. However, the improved readability and the possibility to write simulation descriptions in less time still justifies the performance penalty. Moreover, this penalty can be kept small by using the high-level functions of the simulators for creating nodes and connections in the simulator-specific backends of PyNN.

4.5 Community driven development

It is important to note that the development of PyNN is actively supported by its users and the respective simulator developers. This means that PyNN has become a central point for the exchange of concepts and techniques for simulator development in general.

Driven by the FACETS project, these collaborative efforts lead to the foundation of the open-source sharing platform Neural Ensemble (`http://neuralensemble.org/`), which plays an important role in the joint efforts to create an open-source tool chain for computational neuroscience. The founders of the platform organize an annual event (*CodeJam*; `https://neuralensemble.org/meetings.html`, where developers of tools and simulators share their knowledge and experience and present new developments in their domain. The newly emerging culture of sharing and re-use lead to the development of tools for data analysis (NeuroTools; `http://neuralensemble.org/trac/NeuroTools/`) that are actively maintained by the user community.

4.6 Summary

PyNN allows to write model descriptions in a simulator-independent way by using backends for the simulators that support Python and by allowing code generation for the native simulator interfaces where no Python interface is available.

The use of PyNN eases the task of porting models from one simulator to another and improves the communication between the users of different simulators and from different laboratories.

Another advantage of PyNN over the simulator-specific APIs is its support for NeuroML (Crook & Howell, 2007; Crook et al., 2007), a standard for model descriptions based on XML. Using PyNN, it is already possible to use models, which are written in NeuroML, even in simulators that do not support it currently.

During the work for this thesis, we contributed to the overall design of the PyNN API. NEST was one of the first simulators to be supported by PyNN, and the two projects influenced each other mutually to make them more useful for the neuroscientific day-to-day work.

Chapter 5

Communication between different simulators

Throughout the history of neuroscience, the view of the brain and the recognition of its relevant parts has changed many times. This refinement was based on new insights that became possible either due to advancements in technology, or because of a paradigm shift. On the technical side this includes aids such as optical microscopy or functional magnetic resonance tomography (*fMRI*) to enhance the visibility of certain structures, but also new chemical methods, like dyes to selectively stain certain components of brain tissue.

Today, the *neuron doctrine* is a generally accepted theory. It states that the nervous system (just like all other living tissue) is made up of single nerve cells (Barlow, 1972; Shepherd, 1991). These cells (*neurons*) communicate by means of electric pulses (*spikes*) mediated by the flow of ions, over connections that are called *synapses*. Ultimately, this communication brings about the behavior of an animal on all levels, from simple reflexes to complex cognitive tasks such as reasoning and consciousness. Sensory organs like the eyes, or the heat and pressure sensors in the skin, encode their measurements of physical quantities in spike trains that are communicated over nerve tracts to the brain. The brain processes these inputs and finally generates a behavior, again by using spike trains, which are sent to the muscles and to the organs of the body of the animal.

The brain exhibits interesting dynamics and phenomenons on a large range of spatial and temporal levels. In neuroscience, a multitude of methods is used to characterize these levels, and to measure their electrical and chemical properties. The resolution of the different methods, however, varies considerably: electroencephalography (*EEG*) for example has a spatial resolution of several centimeters. This means that the activity of many millions of neurons is seen as one accumulated signal with this method, whereas other methods such as electrophysiology allow to measure the activity of individual neurons with high precision.

To understand the function of the brain, it is important to understand the dynamics on all levels. However, the majority of researchers in neurobiology are working only on one specific level. The main reason for this is that most of the methods used for the investigation are restricted to a specific level of the organization, and that it is thus not possible to investigate all levels at once. Working in this way, neuroscientists accumulated a considerable amount of knowledge about the working principles and the mechanisms that underlie the function of the different levels over the past years.

More and more it becomes clear that a single level of description is not sufficient to explain the function of the brain as a whole. The brain has many different subsystems that are organized in hierarchies. The hierarchies interact with each other in order to bring about what is often referred to as "the function of the brain". However, the notion of *function* is over-simplified as each level has its own task and therefore its own function that adds its share to the function of the entire system.

As long as the knowledge about the different levels is not embedded into the context of the whole system, and by looking at the single achievements alone, it is hardly possible to get an overall picture of the working principles in the brain, and to understand how complex phenomenas like consciousness and mind come about.

5.1 Levels of organization in the brain

The brain is hierarchically organized on many different levels and on many different spatial scales (see Figure 5.1). The different levels can be distinguished by the elements that play a role in their function. For example the highest of them is given by the behavioral level of cognitive functions that are created by the activity of the complete central nervous system (*CNS*). Many of the lower levels cannot be defined as precisely, as not all building blocks and mechanisms are fully known yet. To understand the function of the brain as a whole, it is necessary to integrate the knowledge about the functions from all of the different levels.

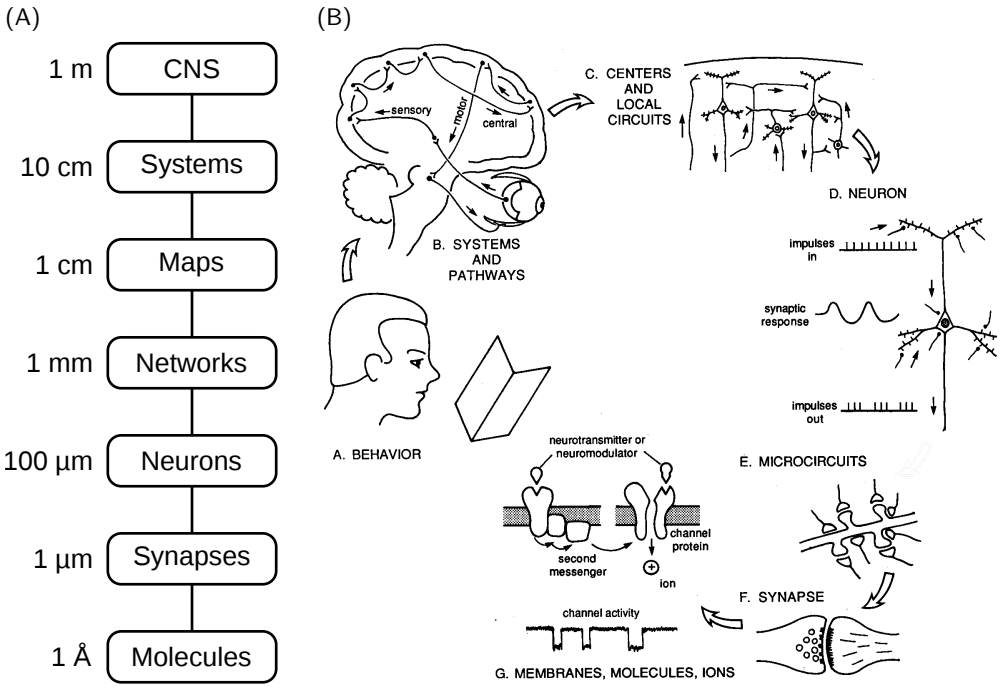


Figure 5.1: *Different levels of organization in the nervous system: (A) The spatial scales at which anatomical organizations can be identified varies over many orders of magnitude (redrawn after Churchland & Sejnowski, 1988). (B) Levels of organization in the nervous system (taken from Shepherd, 1988).*

Not all parts and functions of the nervous system are accessible easily to neurobiological experiments. The behavior of an organism for example is visible to an observer in the environment. Likewise, the general organization of pathways in the early sensory and motor systems can be identified in anatomical studies without the need for an overly elaborate laboratory setup. Therefore it is not surprising that a lot of knowledge is available for these specific levels of detail. On the other side, relatively little is known about the properties at the network level in comparison with the detailed knowledge we have of synapses. The reason for this is that synapses are distinct entities that can be characterized by their bio-chemical properties, while networks are often highly recurrent, and their function cannot be tested easily in an experimental setup due to the strong interactions in the system.

Even today, neuroscientists argue about the right level of description that has to be used in order to understand the brain. Moreover, a clear separation of the levels is not always possible due to the strong interactions between them. However, it is also clear that brain function emerges from this interaction, and that information on all levels of the hierarchy provides valuable knowledge.

5.2 Multi-scale models of the brain

Similar to the current style of research in classical neuroscience, the models produced by computational neuroscience often stay on one specific level of detail. The reason for this is that models are often created to reproduce and understand one certain study from classical neurobiology, or the function of one specific subsystem. A selection of the different types of models that are used currently is contained in the following list (see also Section 1.4.2):

- Hypothesis about the function of whole populations of cells and about whole brain areas can be tested using population or field models that do not explicitly contain models of single neurons.
- Effects on the network level are examined in large networks consisting of point neuron models, or of compartmental neuron models that are based on three dimensional reconstructions of real neurons.
- Signal transduction in single cells is investigated in detailed compartmental models of whole neurons or patches of cell membrane that are based on detailed electrophysiological measurements.
- Chemical processes and the molecular basis for the function of cells and synapses are studied using reaction-diffusion models, which allow to understand the interactions of molecules and ions in the cell and at the cell's membrane.

The variety of the models in above list makes clear that we depend on all these models in order to understand and describe the different levels appropriately. However, the models on the different scales are not necessarily unrelated. They often describe the same aspect of the system using a different level of abstraction in order to permit an easier analysis or simulation of the system. Furthermore it is not even possible to develop a single model that covers all levels with the same precision, because too much detailed knowledge on the system would be necessary to do so.

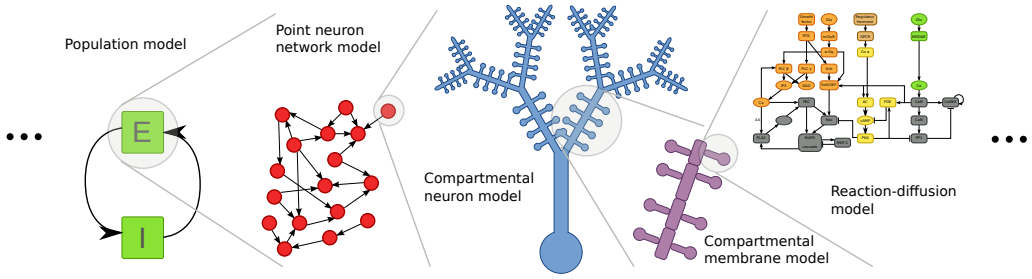


Figure 5.2: *Sketch showing the relation of different modeling scales (redrawn and extended from Dudani et al., 2009): The coarsest end of the scale is given by population models. On a finer level, populations can be modeled by networks of point neurons. Point neuron models are abstractions of detailed compartmental models of neurons and membranes. The chemical reactions in the cell or at the cell's membrane can be modeled using reaction-diffusion systems. The scale extends in both directions.*

Figure 5.2 shows the relation of the different models explained in the list above. Each of the elements in one of the higher levels can be replaced or refined by a model of the lower level. However, the the variety of models is also reflected in the development of simulators. Different simulators exist and are also specialized on a specific level of detail. The main reason for this is complexity: it is hardly possible to develop and maintain a simulator that works reliably on all levels of detail, and provides all researchers with the required features for their work. In addition, the multitude of simulators has a lot of advantages for the exploration of new algorithms and for the validation of simulator implementations. Therefore it is not desirable to only have a single simulator for all scales (see also Chapter 3).

The following list contains a selection of well known simulators for the different levels of detail that are shown in Figure 5.2:

Population models: Nengo (Stewart et al., 2009), Topographica (Bednar, 2009)

Point neuron models: Brian (Goodman & Brette, 2008), NEST (Gewaltig & Diesmann, 2007), PCSIM (Pecevski et al., 2009)

Compartmental models: GENESIS (Bower & Beeman, 1997), MOOSE (Ray & Bhalla, 2008), NEURON (Hines & Carnevale, 1997), SPLIT (Hammarlund & Ekeberg, 1998)

Reaction-diffusion models: MCell (Kerr et al., 2008), STEPS (Wils & De Schutter, 2009)

Only in the recent years, researchers in computational neuroscience started to integrate the models on the different levels that were created independently in the past. This is a promising approach to understand the brain as a whole, and it is highly probable that we learn more about the brain on a system level by integrating different models, than we learned by looking at the levels separately.

This integration marks a completely new direction in the research of neural systems, and the technology for this kind of multi-scale models is only just starting to become available. Moreover multi-scale modeling is not very common in other fields of science, either. This means that we cannot learn from the experience of others, but have to develop our own methods instead.

5.3 Interoperability between simulators

The rising complexity of the models, especially in large-scale neuroscientific projects such as FACETS, requires the integration of small models on multiple scales into larger models.

In order to study the dynamics of a single compartmental neuron model, for example, it is important to provide the model neuron with realistic input from a large number of presynaptic cells. However, it is often not possible to model these cells with the same attention to detail as the studied neuron because of the requirements for and computing power this would entail. One possibility to overcome this limitations is to provide input to the neuron by using a network of point neurons modeled in another simulator. The information that is gained from such a study with a detailed neuron model can then be used to refine the point neuron model.

Such an integration can happen in two basic ways: *offline*, on the level of a common language, or on the level of data files, and *online*, allowing different applications to talk to each other at run time.

5.3.1 Offline interoperability

One example for a tool that allows offline interoperability between different simulators has been introduced in Chapter 4. PyNN is a common interface for different simulators, which allows the researcher to use different simulators with a common description language. This makes it easier to port models from one simulator to another, and minimizes the effort for porting between different programming and configuration languages.

Other approaches for offline interoperability are based on the data format of the applications and provide common interfaces to read and write data, or to use the same analysis and visualization tools for the data originating from different simulators.

However, an offline approach only allows to implement *open loop* interactions between applications, where the data flows from one application to the next, but without recurrent connections between the applications. The simulation of *closed loop* interactions requires the transmission of data between the application at run time.

5.3.2 Online interoperability

An example of a model that requires the interaction of simulators in an online fashion is a network of point neurons (for example simulated in NEST) where each neuron model has biologically realistic synapses, based on a reaction-diffusion system (for example simulated in STEPS). Because of the bidirectional interactions between neurons and synapses, this model cannot be simulated in an offline fashion.

An extension to this is to include hardware into the setup: a camera (e.g. the dynamic vision sensor developed at the ETH Zürich, `http://siliconretina.ini.uzh.ch/wiki/index.php`) could provide input to a model of the visual system. To close the loop, this model could be coupled to a system that controls camera movement and thus decides what the model sees next.

The setup of such a closed-loop simulation, however, was often tedious in the past. Most solutions were based on *named pipes* or *sockets* to transmit the data between different applications. This required a manual synchronization of the applications, and the development of custom data protocols understood by both simulators.

5.4 The multi-simulator coordinator MUSIC

To bridge the gap between models on different scales, and to allow a model in one simulator to interact with a model in another simulator, it is necessary to couple the different simulators at run time. In addition to coupling different simulators, this approach can be used to couple programs for stimulus generation, data analysis, and data visualization with simulators, and with each other.

On the first *INCF workshop on large-scale modeling of the nervous system* (Djurfeldt & Lansner, 2007), a library for the exchange of data between simulators for spiking neurons at run time was first discussed. In 2008, the International Neuroinformatics Coordinating Facility (INCF) commissioned a standard to allow this. The Multi Simulator Coordinator (*MUSIC*; Ekeberg & Djurfeldt, 2008) is a library based on MPI (Message Passing Interface Forum, 1994), which lets different applications exchange spikes, continuous data, and arbitrary text messages at run time. MUSIC coordinates the timing of the data exchange, sets up communication routes between the different applications, and makes sure that the data from each process of one application arrives at the correct process in another application at the right time.

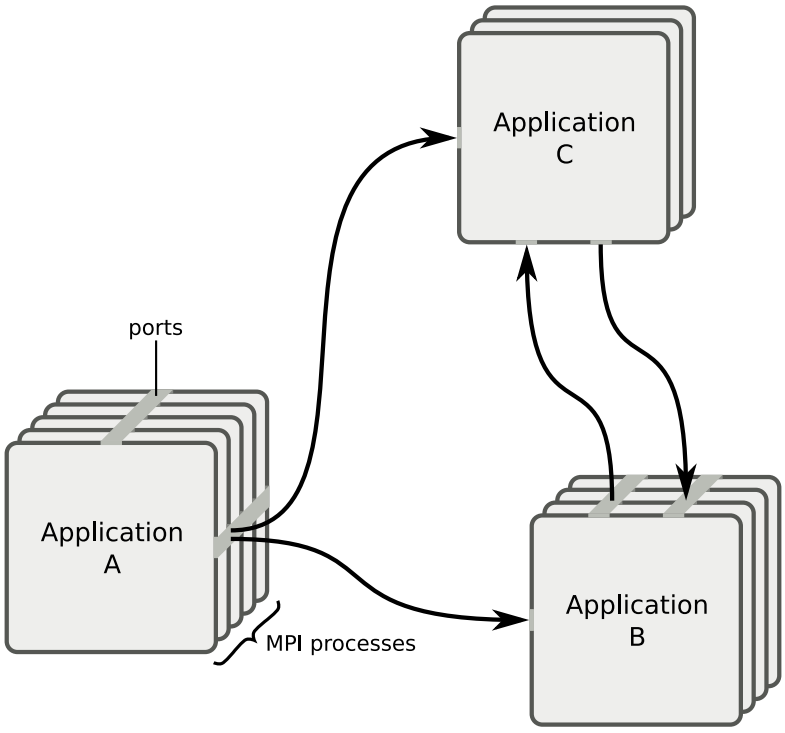


Figure 5.3: *Illustration of a typical multi-simulation using MUSIC (taken from Djurfeldt et al., 2010): Illustration of a typical multi-simulation using MUSIC. Three applications, A, B, and C, are exchanging data during run-time. Each application runs in a set of MPI processes. Data flows exit and enter* ports, *each spanning the set of processes of the application.*

The basic setup of a multi-simulation that uses MUSIC is shown in Figure 5.3. Applications that use the MUSIC library can register so-called *ports* with the library. A port is a named connection point that provides an arbitrary number of communication channels. The number of

channels is called the port's *width*. Different types of ports are available for spikes, continuous data, and text messages. An output port of one application can be connected to an input port provided by another application, given the ports have the same width and type. The communication graph is set up in a configuration file that is read by MUSIC when the simulation is started.

A multi-simulation using MUSIC is run in three distinct phases: during the *launch* phase, the applications are started with the number of processes that is specified in the MUSIC configuration file. Technically, the launch phase starts with the call of `mpirun` and ends when the constructor of the `Setup` object is called. The creation of the `Setup` object marks the begin of the *setup* phase, in which applications can publish the ports they can receive and send data on. The `Setup` object is responsible for calling `MPI::Init()`. The last phase is the *run-time* phase, which starts with the destruction of the `Setup` object by the constructor of the `Runtime` object. At the beginning of this phase, MUSIC sets up the communication graph between the applications. During the run-time phase, the applications can send and receive data and have to call the `tick()` function of the `Runtime` object in regular intervals. The interval is specified during the creation of the `Runtime` object.

5.4.1 Requirements for using MUSIC

To use MUSIC, an application has to fulfill several requirements, which are described in their temporal order during the multi-simulation in the following list:

Initialize MUSIC by creating a `Setup` object.

Publish ports to inform MUSIC about the ports on which the application can send and receive data during the run-time phase.

Map ports to inform MUSIC about the processor, on which the data for a specific channel on a port is made available.

Initiate the run-time phase by creating a `Runtime` object and inform MUSIC about the desired time step.

Advance time by calling `tick()` in regular intervals as specified during the creation of the `Runtime` object. During this call, MUSIC transfers the events collected in the previous simulation cycle.

Send and receive events during the simulation by using the communication facilities of MUSIC. Received events have to be delivered to the targets in the receiving process.

Finalize MUSIC by calling `finalize()` on the `Runtime` object. This shuts down all communication and allows the applications to quit safely.

5.4.2 Auxiliary tools for MUSIC

In addition to the library, MUSIC comes with a number of tools that provide event sources for stimulus creation, and event sinks for the visualization and recording of data, independently of the specific simulator used. These applications use the same mechanisms as the simulators to connect to MUSIC and therefore comprise minimal examples of how to use the MUSIC library. The output of the visualization application is shown in Figure 5.4.

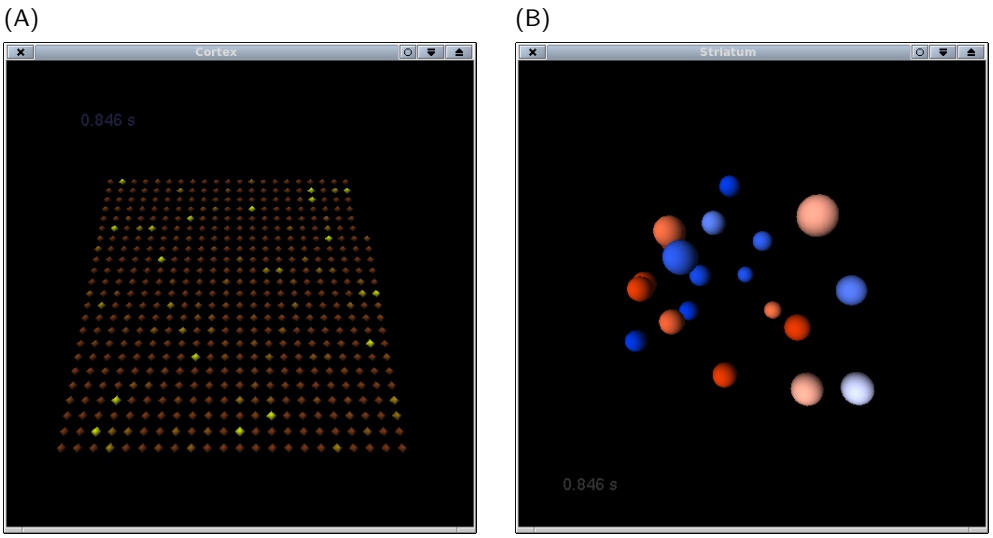


Figure 5.4: *Visualization tools to be used with MUSIC (taken from Djurfeldt et al., 2010): Two window captures from 3D visualizations of the cortex and striatum model (Djurfeldt et al., 2010). (A) 500 outputs from the cortex model in NEST are visualized on a planar grid, the radii and intensity of the color of the neurons increase when they spike. (B) 10 medium spiking (red) and 10 fast spiking (blue) neurons in the striatal network in MOOSE are visualized in the same manner.*

5.4.3 Simulator support

MOOSE and NEST were the first two simulators to be extended by an interface to MUSIC. These interfaces demonstrated the implementability of the standard, and enabled the first multi-scale simulation, which combined two models that had been developed independently before (Djurfeldt et al., 2010).

MOOSE (Multiscale Object Oriented Simulation Environment, `http://moose.ncbs.res.in/`) is a simulator for detailed biological models of neuronal and biochemical networks. It is a multi-scale simulator in the sense that models can be built by coupling components from different levels of detail, from single molecules to whole neurons. Although MOOSE already supports multi-scale simulations itself, the combination of existing models within MOOSE would still require to port the models to MOOSE. In order to relieve the researcher from this task, the addition of a MUSIC interface is a valuable contribution to strengthen MOOSE's position in the field of computational neuroscience.

Objects in MOOSE communicate with each other by sending *messages*. Here, a message is a connection between two objects which allows them to exchange information during the simulation. For the implementation of the MUSIC interface, two new classes of objects were developed, one for receiving spike information from MUSIC and relaying it to other MOOSE objects, and one for receiving messages from MOOSE objects and forwarding them to MUSIC. The details of the MUSIC interface for MOOSE are described in Djurfeldt et al. (2010).

The following section contains a detailed description of the MUSIC interface for NEST, which we designed and implemented during this thesis.

5.5 The MUSIC interface for NEST

NEST is a simulator specialized on large networks of point neuron models, or neuron models with a small number of electrical compartments (see Chapter 2). It is used to study phenomenons on the network level rather than the dynamics of single neurons. The use of MUSIC in NEST is optional. By using the preprocessor macro `HAVE_MUSIC`, it is possible to compile NEST without requiring the MUSIC library.

Neurons and other sources or targets for events that are located in remote MUSIC applications are represented by input or output *proxies* in NEST. These proxies are derived from the base class `Node` just like all other nodes in NEST. This means that the functions `GetStatus` and `SetStatus` can be used to retrieve and modify their parameters, and that they can be integrated into the network like neurons and devices. This design only required three new classes, and minimal changes to the existing classes of NEST.

The creation of `Setup` and `Runtime` objects and the transition from setup to run-time phase are handled by NEST's `Communicator` class. In addition, this class was changed to use a MPI communicator created by MUSIC instead of `MPI_COMM_WORLD`, which is used by MUSIC itself. This change is necessary to let MUSIC optimize the data transfer between the different applications.

The scheduler of NEST was changed to execute the MUSIC function `tick()` once per update cycle. This function delivers queued events to their target application, which then forwards them to the respective target neurons.

5.5.1 Sending events to MUSIC

The class `music_out_proxy` is responsible for sending data from nodes in NEST to remote MUSIC applications. For each output port, one `music_out_proxy` is created in each process. However, one output proxy can handle all channels that are associated with the corresponding output port. The following listing shows the creation of such a proxy, and the setup of connections from neurons to the proxy using PyNEST (see Chapter 3 and Eppler et al., 2009).

```
1  outproxy = Create("music_out_proxy")
2  SetStatus(outproxy, {"port_name": "spikes_out"})
3  neuron1 = Create("iaf_neuron")
4  Connect(neuron1, outproxy, {"music_channel": 0})
5  neuron2 = Create("iaf_neuron")
6  Connect(neuron2, outproxy, {"music_channel": 1})
7  neuron3 = Create("iaf_neuron")
8  Connect(neuron3, outproxy, {"music_channel": 2})
9  neuron4 = Create("iaf_neuron")
10 Connect(neuron4, outproxy, {"music_channel": 3})
11 neuron5 = Create("iaf_neuron")
12 Connect(neuron5, outproxy, {"music_channel": 4})
```

Line 1 creates an instance of a `music_out_proxy`, which will send the spikes it receives to the port called `spikes_out`, which is set using `SetStatus` in line 2. Line 3 creates a neuron of type `iaf_neuron`, which is connected to channel 0 of the output port in line 4. The remaining lines create more neurons and connect them to the channels 1 to 4. The width of the port is set implicitly by the highest channel number that is used in the model description.

The MUSIC channel is set during connection setup and thus stored in the `Connection` object. In addition, the channel is stored by the proxy in an index map (called `indexmap`) during the connection handshake (see Section 2.4.3) to be mapped with MUSIC before the start of the first simulation. During the simulation, the channel will be transmitted to the proxy with each spike, so that the proxy knows on which channel the spike has to be sent.

Before the simulation, NEST's scheduler calls `calibrate()` on each node. In this function, the `music_out_proxy` checks if the used channels are compatible with the width of the port, and the content of the `indexmap` is used to map the port with MUSIC. This happens in three steps:

1. Create a `MUSIC::EventOutputPort`, `outport`. This triggers an exception if the port already exists
2. Create a `MUSIC::PermutationIndex` and initialize it with the data from the `indexmap`. The `PermutationIndex` informs MUSIC about the local channels in a process.
3. Call `map()` on `outport` with the `PermutationIndex` argument to map the port with MUSIC.

Note that the connections to the `music_out_proxy` bypass NEST's system for handling synaptic interactions. Incoming events are directly forwarded to the corresponding channel on the MUSIC port `outport` by the `music_out_proxy`. Synaptic interactions have to be implemented in the receiving application.

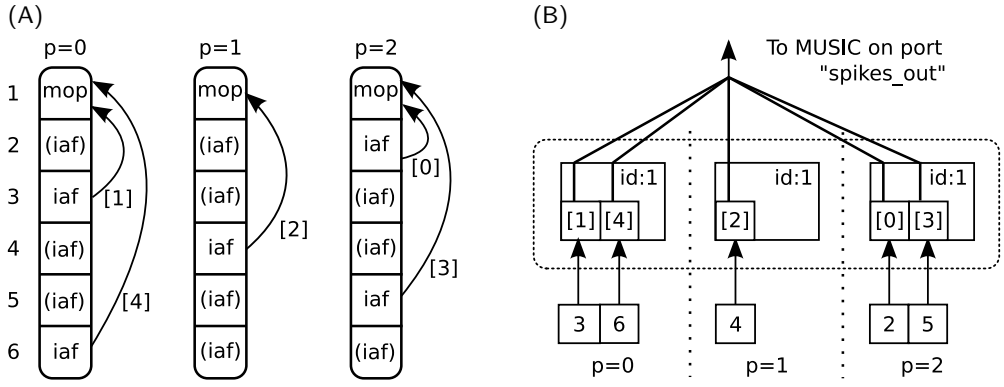


Figure 5.5: *Network representation for the `music_out_proxy` (taken from Djurfeldt et al., 2010): (A) Nodes in NEST are distributed over the processes ($p = 0, 1, 2$). `iaf` denotes an integrate-and-fire neuron, `(iaf)` denotes a NEST proxy. `mop` denotes a `music_out_proxy`. MUSIC channels are indicated in square brackets for each connection (arrows). (B) A sketch of the complete connectivity from the nodes (lower squares) over the different channels (numbers in square brackets) to MUSIC. The dashed box encloses all proxies that belong to one MUSIC output port.*

Figure 5.5 shows the network in NEST after the above commands were executed using three NEST processes. The proxy is created in all three processes, while connections are only established on the process, where the presynaptic neuron exists. This setup is consistent with the ordinary setup of a network in NEST without using MUSIC (see Section 2.3.2).

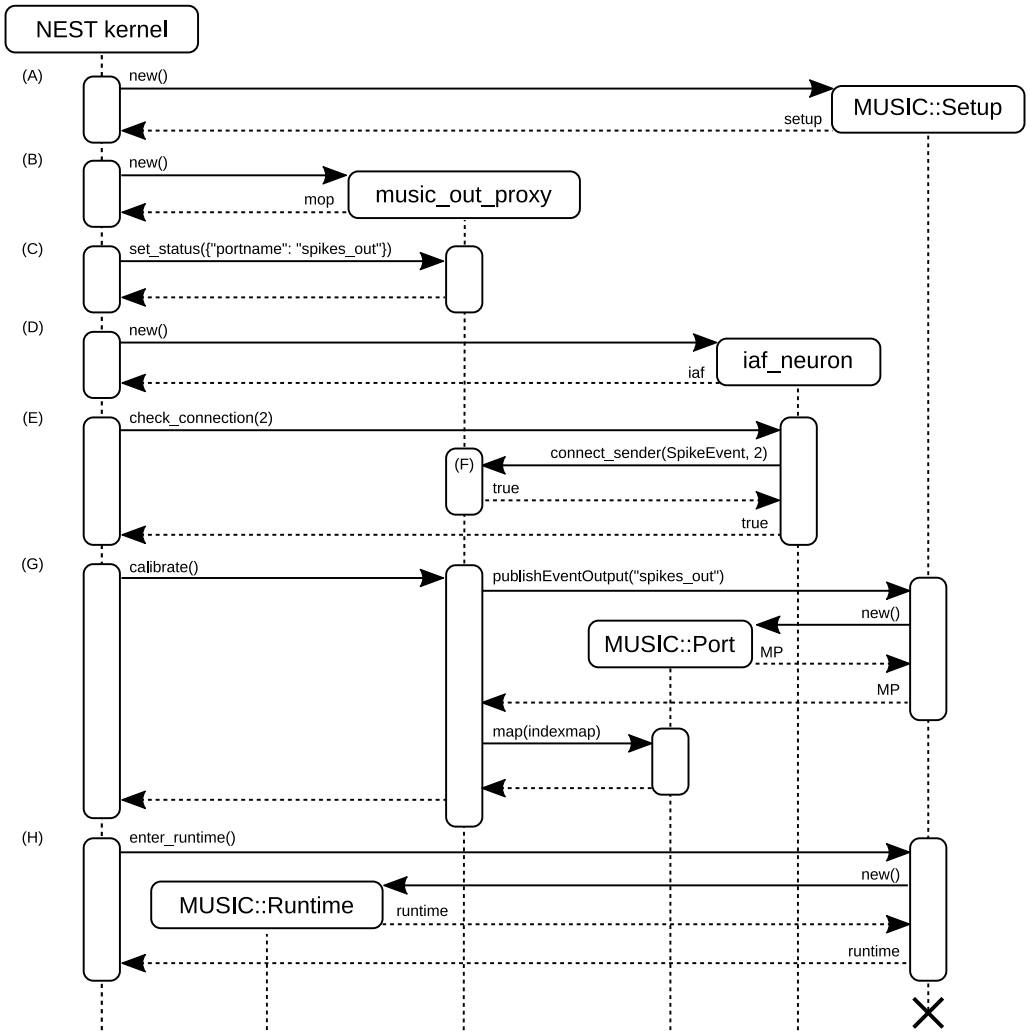


Figure 5.6: *Sequence diagram for the NEST-MUSIC interaction using a* `music_out_proxy`*: Sequence diagram showing the events from setup phase to run-time phase of a network containing a* `music_out_proxy`*. (A) On start-up, NEST creates a MUSIC* `Setup` *object. (B) The proxy is created by the command* `mop = Create("music_out_proxy")`*. (C) The port name for the proxy is set using* `SetStatus(mop, "portname", "spikes_out")`*. (D) A neuron is created by calling* `iaf = Create("iaf_neuron")`*. (E) The neuron is connected to the MUSIC output channel 2 of the port the proxy represents using* `Connect(n, mop, {"music_channel": 2})`*. (F) The proxy stores the connection in its* `indexmap`*, which contains the indices of all local channels of the port. (G) Calling* `Simulate()` *first calls* `calibrate()` *on the proxy, where it registers itself with MUSIC using the* `indexmap`*, built previously. (H) NEST then enters the MUSIC run-time phase by calling* `enter_runtime()` *on the* `Setup` *object, which returns a* `Runtime` *object and destroys itself thereafter.*

Figure 5.6 shows the internal order of events for a complete example of an `iaf_neuron` that sends its events via a `music_out_proxy` to targets in remote MUSIC applications.

5.5.2 Receiving events from MUSIC

For sending events to MUSIC, we could set the channel during the setup of the connection. This was possible, because connection setup in NEST includes a handshake, in which the receiver is informed about the connection and can therefore store the mapping of connection number to desired channel (cf. Section 2.4.3) in order to send the events to the right MUSIC channel during the simulation. This mechanism was originally designed to select the compartment or receptor type a neuron connects to. For the selection of a sending compartment, we do not have such an infrastructure. This means that we cannot specify the MUSIC channel during connection setup to a single proxy.

The setup for handling incoming events from MUSIC in NEST is thus more complicated than the setup for outgoing connections, and we need two classes instead of one: the class `MusicEventHandler` is used to receive the events from MUSIC and forwards them to the `music_in_proxy` that corresponds to the channel the event is addressed to. The `MusicEventHandler` maintains a map `channelmap`, which maps the channel to the proxy for the channel. The proxy itself can be connected to other nodes in the network. In contrast to the setup for sending events, these connections use the synapse system and can thus use all synapse types available in NEST.

The creation of `MusicEventHandlers` for the MUSIC input ports is carried automatically before the simulation. The following listing contains an example that illustrates the use of `music_in_proxys` in a small network:

```
1 inproxy1 = Create("music_in_proxy")
2 SetStatus(inproxy1, {"port_name": "spikes_in", "music_channel": 0)
3 inproxy2 = Create("music_in_proxy")
4 SetStatus(inproxy2, {"port_name": "spikes_in", "music_channel": 1)
5 neurons = Create ("iaf_neuron", 4)
6 DivergentConnect(inproxy1, [neurons[0], neurons[1]])
7 DivergentConnect(inproxy2, [neurons[1], neurons[2]])
8 Connect(inproxy1, [neurons[3]], model="stdp_synapse")
```

Line 1 and 2 create two `music_in_proxys`. The `port_name` of both is set to `spikes_in`, the `music_channel` to 0 and 1 in line 2 and 4, respectively. Line 5 creates four neurons of type `iaf_neuron`. The first and second neuron receives input from the first `music_in_proxy` (line 6), the second and third receives input from the second `music_in_proxy` (line 7). In addition, the second proxy is connected to the fourth neuron using a STDP connection in line 8.

When the `calibrate()` function of a `music_in_proxy` is called by the scheduler before the start of the simulation, it registers itself with the simulation engine, by calling the `register_music_in_proxy()` function of the `Network` class. This class maintains a list of `MusicEventHandlers` for the different MUSIC input ports and forwards the registration to the one corresponding to the port the proxy is assigned to. If the handler does not exist yet, it is created by `register_music_in_proxy()`.

After the calibration of nodes, the channel maps of all MUSIC event handlers contain all local proxies that are able to handle incoming spikes. At this time, the `MusicEventHandler` can use the channel map to map the channels with MUSIC.

For each incoming spike, MUSIC calls `operator()` on the event handler with the time of the spike and the target channel as arguments. The function `operator()` only stores the time of the spike in the event queue for the corresponding channel and returns.

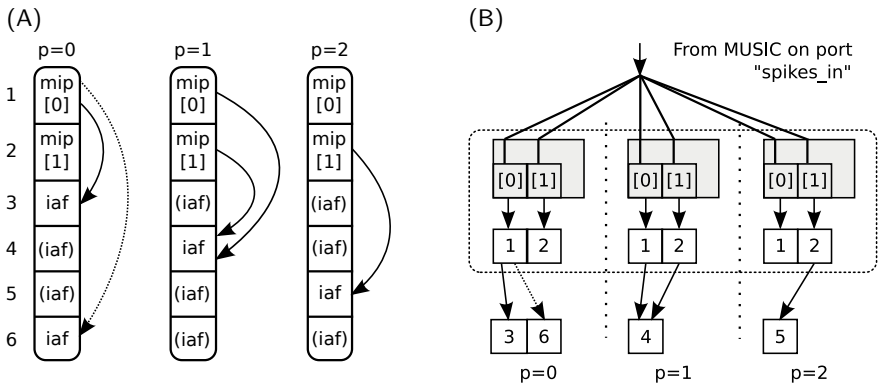


Figure 5.7: *Network representation for the* `music_in_proxy` *(taken from Djurfeldt et al., 2010): (A) Nodes in NEST are distributed over the processes ($p = 0, 1, 2$).* `iaf` *denotes an integrate-and-fire neuron,* `(iaf)` *denotes a NEST proxy.* `mip` *denotes a* `music_in_proxy`*. The numbers on the left indicate the global id of the nodes. MUSIC channel ids are indicated in square brackets for each* `music_in_proxy`*. The STDP connection is indicated by a dotted arrow. (B) A sketch of the complete connectivity from MUSIC (channels in square brackets) to the MUSIC event handler (grew rectangles) to the proxies (squares labeled 1 and 2) to the actual target nodes (lower squares). The STDP connection is indicated by a dotted arrow. The dashed box encloses all event handlers and proxies that represent a MUSIC input port.*

Figure 5.7 shows the network in NEST after the above commands were executed using three NEST processes. The proxies are created in all three processes, while connections are only established on the process, where the postsynaptic neuron exists. In contrast to the `music_out_proxy`, the connections that originate at `music_in_proxy`s can use all of NEST's built-in synapse types.

Buffering of incoming events

The MUSIC standard allows to send events to a target application already long before they are due. This means that the events have to be buffered in the `MusicEventHandler` that corresponds to the port, on which the event enters the application.

During simulation, the events are just queued by `operator()`. Once per update cycle, the function `update()` is called on each `MusicEventHandler` in order to deliver the spikes that are due. The algorithm iterates over the event queues of all channels of the port and checks if the current value in the queue falls into the current time slice of the simulation. If no, it returns. If yes, it creates a new `SpikeEvent` with the corresponding time stamp, and passes it directly to the `handle()` function of the proxy associated with the channel. This bypasses the synapse system in NEST and only informs the proxy about a new spike in a remote application. Upon arrival of new events, the `handle()` function of the `music_in_proxy` immediately calls `Network::send()` to deliver the event to all local targets via the synapse system.

Figure 5.8 shows the internal order of events for a complete example of an `iaf_neuron` that receives its events from a `music_in_proxy` from sources in remote MUSIC applications.

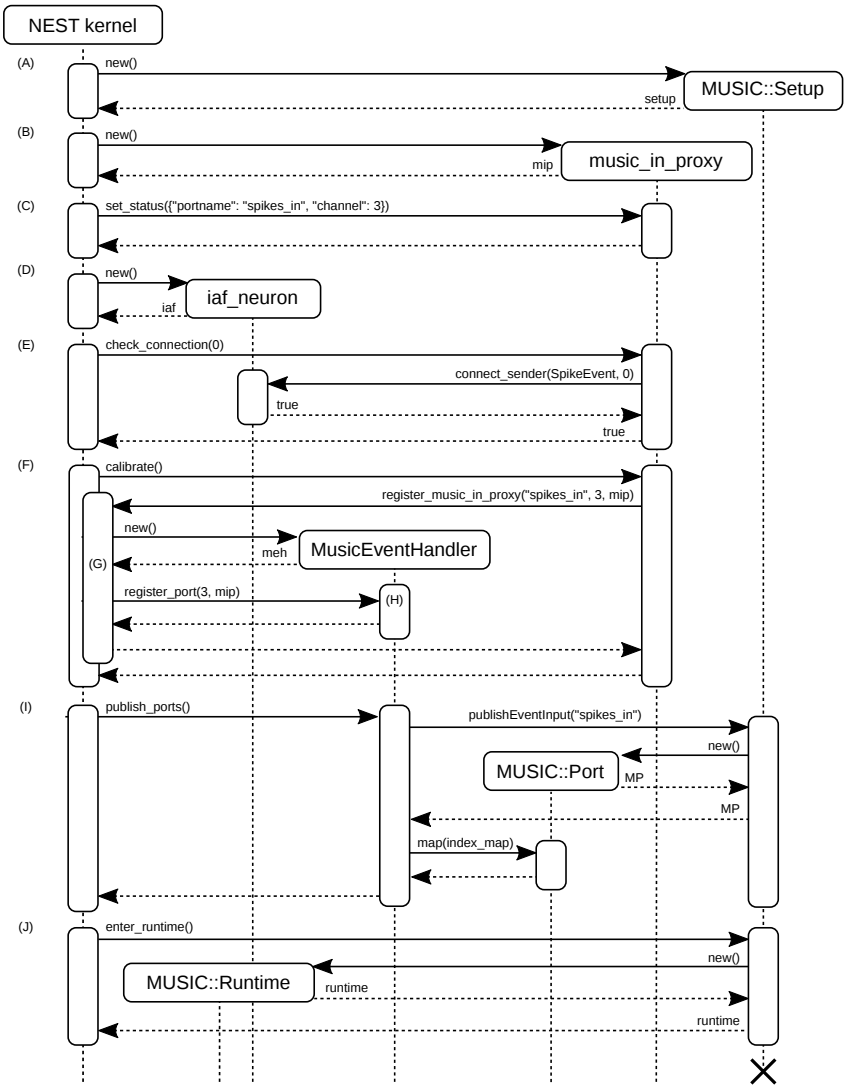


Figure 5.8: *Sequence diagram for the NEST-MUSIC interaction using a `music_in_proxy`: Sequence diagram showing the events from setup phase to run-time phase of a network containing a `music_out_proxy`. (A) On start-up, NEST creates a MUSIC `Setup` object. (B) The proxy is created by the command `mip = Create("music_in_proxy")`. (C) The port name and and the channel of the proxy are set using `SetStatus(mip, {"portname": "spikes_in", "music_channel": 3})`. (D) A neuron is created by calling `iaf = Create("iaf_neuron")`. (E) The proxy is connected to the neuron using `Connect(mip, n)`. (F) Calling `Simulate()` first calls `calibrate()` on the proxy, where it registers itself with NEST kernel using the function `register_music_in_proxy()` with the port name, the channel, and itself as arguments. (G) the NEST kernel stores the new `MusicEventHandler` in its `music_in_portmap` under the name `spikes_in`. (H) The `MusicEventHandler` registers `mip` as the proxy for channel 3. (I) The function `publish_ports()` creates and maps a MUSIC output port object for the ports that are known to NEST. (J) NEST then enters the MUSIC run-time phase by calling `enter_runtime()` on the `Setup` object, which returns a `Runtime` object and destroys itself thereafter.*

5.6 Summary

The interfaces in NEST and MOOSE were tested with a number of toy scenarios to exchange spikes with the tools that come with MUSIC. To show the applicability to real-world simulations, a live demonstration of these interfaces was given at the INCF booth at the *Society for Neuroscience Conference* in Washington 2008, where a detailed striatal model in MOOSE was coupled with a cortical model in NEST. The performance of the system was measured later on in systematic benchmark simulations. A large network model was simulated once in pure NEST, and once by splitting it into two parts that communicate via MUSIC. The two parts were again simulated in NEST. This setup showed that we obtain a scaling behavior with MUSIC that is comparable to the scaling without using MUSIC (Djurfeldt et al., 2010).

Meanwhile a number of neuroscience laboratories are using the MUSIC interface for NEST to couple their own research tools to network models developed in NEST at run time. One example for the integration of models that is carried out using this interface is the model by Casagnes et al. (2010). In this model, visual stimuli are generated by an application module written in C++ and Python and fed into a model of the superior colliculus implemented in NEST. The activity of the superior colliculus model is transferred to a model of the brainstem in NEST, which is again coupled to a model controlling controls eye movements via motor commands (written in C++). All four stages of the model use MUSIC to communicate with the respective next stage. The complete model thus constitutes a closed-loop brain-scale functional circuit.

Currently, both NEST and MOOSE only support the communication via spikes. This was sufficient to demonstrate the feasibility of our design and can already be used for many studies that were not possible before. However, MUSIC also supports the communication of continuous data values and arbitrary text messages. The extension to these types of communication fits naturally into the framework that has been designed during this thesis and will be implemented if future studies require it.

Chapter 6

Discussion

The basis for this thesis is provided by four projects concerned with the design and the improvement of communication architectures for NEST, a simulator for biological neural networks. In particular, the projects were concerned with the following communication systems:

- Algorithms and data structures to exchange spikes between nodes across threads and processes.
- Algorithms to provide convenient access to the data structures of the new simulation engine of the simulator.
- Algorithms and data structures to exchange commands and data between the simulator and programmable user interfaces for Python.
- Algorithms and data structures to exchange spikes with other simulators and analysis tools via the MUSIC library.

Although the new features and changes were carried out in the NEST simulator, they solve general problems related to the simulation of spiking neural networks. The algorithms can easily be adapted to be used in other simulators as well. Especially our approach to creating Python bindings for NEST is sufficiently general to be used by applications also outside neuroscience.

After first tests within the NEST Initiative, the new algorithms and data structures were made available to the users in the official pre-releases, available at the homepage of the NEST Initiative at `http://www.nest-initiative.org/`. The reactions to the new user interface and other improvements at conferences, summer schools, and on our support mailing list were very positive, and, according to our users, the new features helped to spread NEST in the computational neuroscience community.

The algorithms that resulted from the work described in this thesis were presented at several conferences for computational neuroscience and have meanwhile been published in peer-reviewed computer science and neuroinformatics journals. For the convenience of the reader, the original articles are contained in Appendix A.

This chapter provides an embedding of the work presented in the previous chapters into a broader context and discusses some open questions. It also provides an outlook on our ideas for future directions in the development of NEST.

6.1 Communication inside the simulator

The availability of distributed computing (using multiple processes and message passing) for NEST enables researchers to simulate networks that exceed the memory available on a single computer, and allows to exploit the computing power of multi-processor computers, computer clusters, and facilities for high-performance computing. This technique also allows to reduce network construction time, because the network can be built in parallel on multiple machines. This is especially interesting for very large networks. The improved performance and scaling already enabled studies that would otherwise not have been possible because of the network size or simulation run-times (Schrader et al., 2009; Morrison et al., 2007; Helias et al., 2009). Likewise, the availability of a new framework for the representation of connections allowed the implementation of structural plasticity (Helias et al., 2008), which was impossible with earlier versions of NEST.

One of the main problems in earlier versions of NEST was related to the access to nodes and connections in a multi-threaded setup. Our new algorithms allow to access the connection information, and the data collected by the different threads, without the need for knowledge of the internal data structures. This means that the user only specifies the number of threads before the simulation and does not need to make any other changes to the code compared to a serial simulation.

Chapter 2 shows that the performance and scaling of the simulation kernel is better with message passing than with threads. The main reason for this are the cache problems that occur when different threads access memory concurrently (cf. Section 2.2). As of now, it is unclear if and how these can be resolved without putting too much effort into the optimization of data structures and replicating a lot of the work of the developers of operating system kernels and message passing libraries.

On very large machines, the major part of the memory that is used for storing nodes is used up by proxies, not by neurons and devices. This results in a node list that can no longer be kept in the cache memory of the processor, which slows down the simulation. This problem can be solved by using a more abstract representation of the proxy nodes and a hash table for the storage of the local target lists. Using this method can cut down the memory required for nodes, however at the cost of adding an indirection for node access. In a prototype implementation, we currently test the feasibility of this approach. This refinement process expresses the need to iteratively search for new solutions for known problems, mediating between the problem and the hardware domain, while allowing future extensions of the solutions (Gamma et al., 1994).

In their recent work, Kunkel et al. (2009) analyzed the memory consumption of NEST in a distributed setup. The authors show that the data structure for the storage of connections entails a large memory overhead, which can be reduced by using optimized data structures instead of the current array based approach. Such a data structure can be obtained by using hash tables instead of arrays for all dimensions of the connection storage system that are only sparsely populated.

The question of how the algorithms and data structures of NEST have to be adapted to efficiently support the upcoming *petascale* high-performance clusters is still unclear and a topic of our active research. At the same time, we still need to be able to run the software on small and medium-sized multi-processor machines and computer clusters, because these kinds of machines are already available in many laboratories around the world.

6.2 Communication between user and simulator

PyNEST allows a clean formulation of simulation scripts in a modern high-level programming language. The main reason why Python was chosen for the new user interface of NEST is that many laboratories in the neuroscience community are also moving towards Python and it is becoming the de-facto standard language of computational neuroscience (Kötter et al., 2009; Muller et al., 2009). As such, Python helps to reduce the complexity barrier for code exchange between different researchers, for coupling different tools with each other, and for porting model descriptions from one simulator to another.

During the manuscript submission of Eppler et al. (2009), one of the reviewers noted that the high-level API of PyNEST stays very close to the API of NEST, although Python would have allowed to create an object oriented interface for NEST. The reason for the current syntax of the PyNEST API is twofold: first, we wanted to make the transition for users easier that already use SLI. By using the same function names and the same basic way of operation, we allow an easy translation of simulation descriptions written in SLI to PyNEST. Second, by keeping the same name and basic semantics of the functions, we can use the user-level documentation for both SLI and PyNEST.

There are two basic possibilities to create a more convenient API for PyNEST: the first is to create a second high-level API, which only uses the low-level API, but uses an object oriented approach for the interaction between the user and NEST. The second approach is to use the existing high-level API and provide an object-oriented API on top of this. However, as PyNN reached a state that is very usable and it is questionable if the users would accept such a new interface at all.

We are currently investigating the features of the new version 3 of Python, which will break backwards compatibility with Python 2.x. For the transition period, we will provide two separate versions of the high-level API to give our users the greatest possible flexibility with regard to the version of Python they use.

6.3 A common interface for different simulators

PyNN provides a uniform API to NEST and many other simulators. This is possible through simulator-specific backends built on top of the Python interfaces of the simulators (i.e. PyNEST, in case of NEST). PyNN enables researchers to switch between different simulators very efficiently. This enables the validation of models across multiple simulators, but also to validate the simulators themselves.

With PyNN, the user has a complete chain of tools from NeuroML (Crook & Howell, 2007; Crook et al., 2007) to PyNEST, which makes it easier to re-use models developed by others. In general, this opens interesting possibilities for the implementation of future standards: the support for such standards can be implemented relatively fast in Python, but may not be as efficient as in one of the low-level programming languages in which the simulators are written. However, this allows to test the standards with many simulators without the need for support in each of them. Once the standardization process reached a usable state, the implementations can be transferred to more efficient implementations in the simulation engines of the simulators.

In addition to the support for different simulators, PyNN also has a backend for the analog neural network hardware that was developed in the EU project FACETS (Schemmel et al.,

2008). Here, PyNN was an important tool to verify and calibrate the hardware against the established and well-tested implementations of algorithms in the classical simulators.

The different simulators provide efficient support for modeling and simulating neural systems. PyNN additionally supports concepts like physical units or common data formats for the resulting data files. These features may well be easier to implement in Python than in the languages the simulators are written in. Moreover, the use of physical units in the simulation engine could considerably slow down the simulation.

Last, but not least, the use of a common language for writing model description fosters the re-use of models and components of models without the need to port them from one simulator to another. This leads to faster progress in the field, as researchers are not forced to implement existing models from scratch.

We plan to further support the development of PyNN by providing code and knowledge to its main developers. On the other side, we already implemented many features in NEST in order to support a simplification of the NEST backend for PyNN. This has proven to be a good way of supporting the development of a high-level interface for NEST without wasting our resources in the development of more sophisticated interfaces for NEST itself.

6.4 Communication between different simulators

NEST was one of the first simulators to be extended by an interface for the MUSIC library. During the design and creation of the interface, we worked together closely with the original authors of MUSIC in order to speed up the implementation process. This gave us the opportunity to influence the standardization process of the MUSIC specification, to clarify ambiguities, and to shape the library implementation into a form that can easily be used by simulator and tool developers.

Our interface to the MUSIC library did not require major changes to NEST, as the idea of sending and receiving channels and ports fell nicely into the already existing infrastructure of NEST's concepts. The MUSIC library is only used by NEST if requested by the user at build time. All additional functionality provided by the interface is protected by preprocessor macros in order to keep NEST independent. The interface consists of very little code, so that the additional effort for maintaining the interface is only small.

We currently do not support all of MUSIC's features. For example, the interface does not support any events other than spikes to be sent or received through the MUSIC interface. For future releases, we plan to also support currents and message events to add the possibility for more sophisticated communication with other simulators and tools.

The availability of a MUSIC interface for NEST allows new modeling paradigms to be explored by coupling NEST to other simulators and other software. This allows to build models that span multiple levels of detail and already has proven to be an important selling point for NEST over other simulators. Several laboratories are currently exploring the possibilities of this interface in connection with their own tools for stimulus generation and data analysis.

At the moment of writing, only NEST and MOOSE (`http://moose.ncbs.res.in/`) support the MUSIC standard. However, interfaces for NEURON (Hines & Carnevale, 1997) and PCSIM (Pecevski et al., 2009) are currently under development. The fact that the MUSIC project was initiated and is funded by the INCF gives us hope that it will be widely used.

6.5 Productization of NEST

The term *productization* in the field of software engineering characterizes the sum of all work that is required to develop a software project written for a single and specific purpose into a software product that is useful for others.

During the time of this thesis, NEST gained a lot of new users due to its use in the EU project FACETS and due to its use in several international summer schools for computational neuroscience. While this development shows that NEST reached a state that can be used for many different applications in computational neuroscience by many users, it also brings new challenges with respect to NEST as a software product.

Some important steps in the productization of NEST were accomplished during the time of this thesis. In particular, the following components were improved:

- Configuration, compilation and installation.
- Availability of good user level documentation.
- Availability of good developer documentation.
- Intuitive and convenient user interface.

Since its earliest versions, NEST uses the GNU Build System (`http://www.gnu.org/software/hello/manual/automake/GNU-Build-System.html`) for its configuration and build process. The first step in the productization was fully achieved by the introduction of a proper installation target to this process in 2006. It follows the recommendations of the *Filesystem Hierarchy Standard* (`http://www.pathname.com/fhs/`) and allows to install NEST on a variety of Unix-like platforms conveniently. This made it easier for our users to install and update NEST. In addition, this was the basis for creating binary packages for a variety of Linux distributions, and finally to create a "live CD", which can be run on almost all modern computers without the need to permanently install Linux or NEST at all. Especially the last step made the use of NEST at summer schools easier, as it allows the tutors to concentrate on the usage of NEST, instead of its installation during the course.

The section "Documentation" on the homepage of the NEST Initiative at `http://www.nest-initiative.org/` offers documentation for the usage of the different subsystems of NEST, and provides tutorials for the setup of neural simulations. Many of the examples are available once for the simulation language interpreter SLI and for the new user interface PyNEST. The homepage also contains tutorials for the creation of extension modules, new synapse and neuron models, and for the configuration of NEST. This documentation is updated upon changes in NEST.

PyNEST, the new Python-based user interface for NEST proved to be a good alternative to SLI in that it attracts many new users that are already familiar with the Python programming language. The feedback from these users clearly attests that the new interface indeed lowers the initial problems that kept them from using earlier versions of NEST.

Finally, we present NEST regularly at conferences for computational neuroscience and neuroinformatics in order to inform our users about new features and changes. Meanwhile NEST is used as one of the standard simulators at different summer schools and courses in the field to teach the concepts of modeling.

6.6 Character of the work

From its very beginning, NEST has been developed in a collaborative effort by several laboratories for computational neuroscience. The idea was that each research group contributes the technology that it currently needs to conduct its research, and that the common parts of the simulator are developed together. This need for collaboration has not changed since then. Quite to the contrary: the use of NEST in large-scale projects such as FACETS even reinforces the need for collaboration. NEST has reached a level of complexity that cannot be handled by small groups alone, and we depend on the cooperation of our users more than ever. Most of the improvements described in this thesis have been designed and implemented in close cooperation with the users that need to work with them. This is also reflected in the number of authors of the articles that resulted from this work.

The work for this thesis was carried out in source code, which was simultaneously used and modified by many other developers. Thus, the data structures and algorithms had to be compatible with the existing structure of the code, and with changes by other people in the spirit of the iterative and incremental development strategy of the NEST Initiative (Diesmann & Gewaltig, 2002; Booch, 1996; Brooks, 1995). During the time of this thesis (i.e. May 2006 through June 2010), 15 people worked on the code base of NEST and committed approximately 3100 changesets.

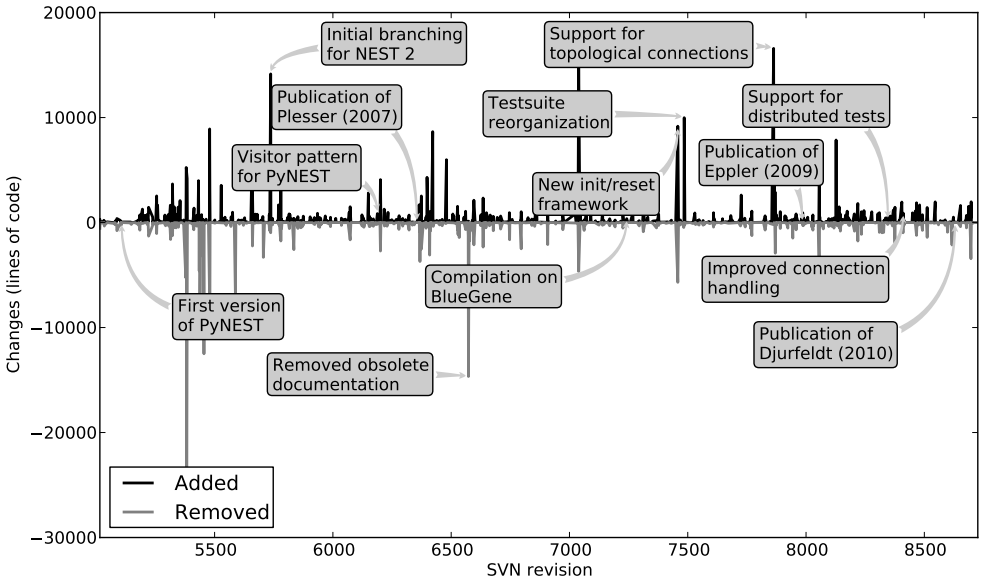


Figure 6.1: *Number of code lines changed per revision: The largest changes are annotated to show the temporal sequence of new features. The symmetry between added and removed lines shows that a large fraction of the changes consists of optimizations rather than the addition or removal of code.*

The high activity on the NEST source code is illustrated in Figure 6.1, which shows the commit history of NEST's code repository during the time of this thesis. Specifically, it shows when code lines were added (black) or removed (gray).

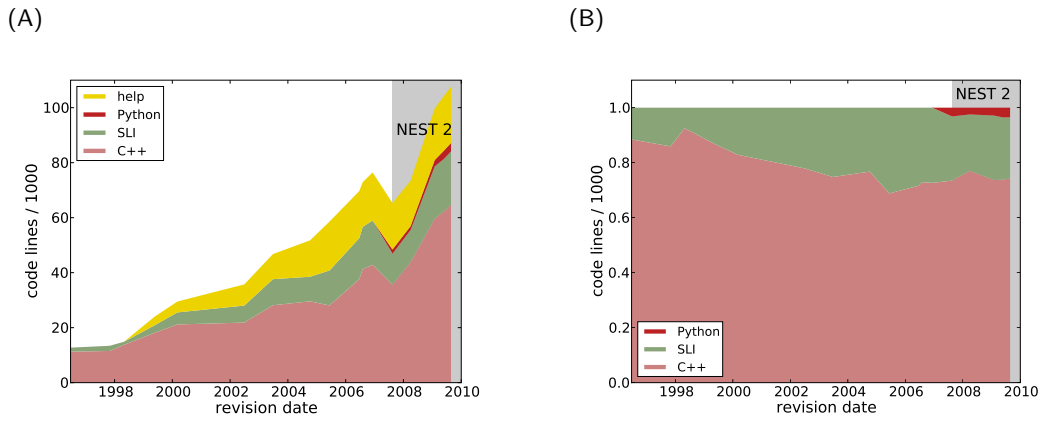


Figure 6.2: *Lines of code in NEST: (A) Absolute number of lines of code including lines of user-level documentation (yellow). (B) Relative number of lines of code.*

Figure 6.2 shows the development of the lines of code in NEST over the past fourteen years (obtained with a modified version of SLOCCount, `http://www.dwheeler.com/sloccount/`). Panel (A) of the figure shows the absolute number of code lines. In the beginning of the development for NEST 2, we lost a lot of code, as not all models were ported to the new API yet. Later, we gained Python code, as examples and test scripts for PyNEST were written. Panel (B) of the figure shows that the relative fractions of the different code parts stayed essentially constant throughout the development of NEST with a slight trend from compiled C++ code towards high-level code written in SLI or Python.

6.7 Outlook

Real biological neurons communicate not only by using spikes. They exhibit a large number of other mechanisms to transmit information between each other. Examples are electrical couplings between neurons (so-called *gap junctions*, see e.g. Connors & Long, 2004), which allow the direct flow of ions from one neuron to the other, and neuromodulators, which can influence the activity of whole groups of neurons simultaneously. Knock-out experiments that deactivate these mechanisms have shown that they play an important role in the information processing. However, the exact role of these mechanisms is not entirely understood and implementations to support them in simulations in NEST remain to be investigated. It is clear, however, that this would require substantial changes to NEST's network representation and to the algorithms and data structures of the communication infrastructure.

The PyNEST interface currently only allows the communication from Python to NEST. For implementing on-line visualization tools and to support neuron and synapse models written in Python (e.g. for rapid prototyping), it is desirable to also let NEST talk to Python. In principle, this is possible by connecting to the Python interpreter at run-time and executing the respective code there. (for a general discussion of interpreter-interpreter interaction, see Diesmann & Gewaltig, 2002). However, Python currently only allows access to its interpreter by a single thread at once. This is incompatible with the multi-threaded update scheme of NEST. A solution for this problem remains to be investigated.

With the growing complexity of the code, it becomes more and more difficult to ensure the correctness and consistency of the simulator. To guarantee the correctness of the implementation and the results obtained with NEST, we have designed a framework for running systematic unit tests in NEST (Eppler et al., 2009). The introduction of the technology, including support for test-driven programming, seems necessary to keep up the pace of software development with the rapid development and the changing research directions in the field of computational neuroscience.

Currently, networks have to be built from scratch for each experiment, and the state of the network is completely lost when NEST is shut down. This is a problem for studies that involve plasticity, or that have long simulation run-times. In addition, we often face time limits on the run-time of a job on large clusters. A solution for these problems is to store the full state of the network once a certain state is reached, and later resume the simulation from this point. Especially for networks that involve plasticity and learning, this would allow to conduct several experiments from the same starting conditions without the need to wait until these have developed. We are currently exploring methods for storing the network to the hard disks of the computer. This would have the additional advantage that the network can be analyzed with external programs to obtain statistical measures of the connectivity.

The computational neuroscience community currently develops standards for model descriptions based on XML (Crook & Howell, 2007; Crook et al., 2007). Several standards for different levels of detail are developed under the name NeuroML. We actively participate in these efforts and plan to support them in NEST, once the standardization process reached a usable state.

In the present work we have focused on advanced communication interfaces and aspects of high-performance computing. This research needs to continue to enable the simulation of networks on the brain-scale in order to close the functional circuits, and to make predictions for new neuroscientific studies. However, several recent publications (Brette et al., 2007; Djurfeldt & Lansner, 2007; De Schutter, 2008) point out that the development of technologies to guarantee the reproducibility and verify the correctness of neuronal simulations are of the same importance.

Appendix A

Publications

A.1 Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers

Hans Ekkehard Plesser[1], Jochen Martin Eppler[2,3], Abigail Morrison[4], Markus Diesmann[3,4], and Marc-Oliver Gewaltig[2,3] (2007) Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.), Euro-Par 2007: Parallel Processing, Volume 4641 of Lecture Notes in Computer Science, Berlin, pp. 672–681. Springer-Verlag. doi:10.1007/978-3-540-74466-5_71.

[1] Dept. of Mathematical Sciences and Technology
Norwegian University of Life Sciences
PO Box 5003, 1432 Ås, Norway

[2] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[3] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

[4] Computational Neuroscience Group
RIKEN Brain Science Institute
Wako-shi, Saitama, Japan

Contributions

- I designed the data structures for network representation (nodes and connections) in the hybrid (message passing + threads) simulation kernel.
- I designed the data structures for the buffering of local events.
- I designed the local event delivery algorithms.
- I designed a system to allow the use of heterogeneous synapse types in simulations.
- I designed the message passing facilities to abstract the interaction of network elements like neurons from the underlying inter-process communication via MPI.
- I conceived a handshake algorithm to check compatibility between sending and receiving nodes during network creation.
- I wrote the major part of the text and created the figures.
- I presented the novel data structures and algorithms for the hybrid simulation engine at the EuroPVM/MPI 2007 meeting (short presentation + poster).

Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers

Hans E. Plesser[1], Jochen M. Eppler[2,3], Abigail Morrison[4], Markus Diesmann[3,4], and Marc-Oliver Gewaltig[2,3]

[1] Dept. of Mathematical Sciences and Technology, Norwegian University of Life Sciences, PO Box 5003, 1432 Ås, Norway, hans.ekkehard.plesser@umb.no
[2] Honda Research Institute, Offenbach/Main, Germany
[3] Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany
[4] RIKEN Brain Science Institute, Wako-shi, Saitama, Japan

Abstract. To understand the principles of information processing in the brain, we depend on models with more than 10^5 neurons and 10^9 connections. These networks can be described as graphs of threshold elements that exchange point events over their connections.
From the computer science perspective, the key challenges are to represent the connections succinctly; to transmit events and update neuron states efficiently; and to provide a comfortable user interface. We present here the neural simulation tool NEST, a neuronal network simulator which addresses all these requirements. To simulate very large networks with acceptable time and memory requirements, NEST uses a hybrid strategy, combining distributed simulation across cluster nodes (MPI) with thread-based simulation on each computer. Benchmark simulations of a computationally hard biological neuronal network model demonstrate that hybrid parallelization yields significant performance benefits on clusters of multi-core computers, compared to purely MPI-based distributed simulation.

1 Introduction

The neuronal networks in our brains can be described as weighted, directed graphs, with neurons as nodes and synaptic connections as edges. Neurons communicate by sending and receiving point events (spikes) through their connections (synapses). In the mammalian cortex, each neuron sends connections to about 10^4 other neurons and receives connections from as many. Just 1 mm^3 cortex contains some 10^5 neurons with 10^9 connections [1]. This represents a threshold size for simulations, as a realistic number of synapses per neuron can be combined with realistic sparseness (connection probability ~ 0.1). Brain function emerges from the spatio-temporal patterns of neuronal spike activity, but

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 672–681, 2007.

the principles are poorly understood. Progress in understanding brain function therefore depends on simulation studies of large cortical networks.

In large neuronal networks, we can neglect the geometric and biophysical complexity of individual nerve cells and describe neurons as point-like objects with a dynamic state governed by a set of ODEs. The most common state variable is the membrane potential V, which is affected by spikes that arrive at the neuron's synapses. Whenever V crosses a threshold value V_{th}, the neuron produces a spike, which is transmitted to all adjacent neurons with a delay of a few milliseconds. Each connection can have a different delay and weight. Weights may evolve as a result of neuronal activity, a phenomenon known as synaptic plasticity, the biological substrate of learning. The spikes of an individual neuron are rare and occur at rates of 1–50 Hz, whereas the rate of incoming spikes is of the order of 100 kHz due to some 10^4 incoming connections.

Simulating large-scale neuronal networks poses several challenges: (i) 10^9–10^{12} connections must be stored; this requires a distributed representation. (ii) A large number of spikes must be buffered until they are transmitted across the network. (iii) Simulation results must be reproducible down to the level of membrane potentials and spike times. (iv) The object-oriented implementation must be appropriate for the problem domain and allow network and machine level optimizations such as efficient caching.

In this contribution, we describe how the Neural Simulation Tool NEST [2] addresses these issues to efficiently simulate neuronal networks of more than 10^5 neurons and 10^9 synapses. In section 2 we discuss how NEST represents nodes and connections, before describing the update and communication algorithms in section 3. Section 4 demonstrates the performance of our hybrid approach. NEST is available from `www.nest-initiative.org`.

2 Network Representation

A network model consists of nodes, connections, and events, each represented by an abstract base class. Models for neurons and devices inherit from `class Node` and implement the state vector, the internal dynamics, and the responses to different types of events.

NEST distributes a network model over N_{VP} *virtual processes*. A virtual process is a POSIX thread that lives in one of N_{MPI} MPI processes [3, 4]. The total number of virtual processes N_{VP} is the number of MPI processes times the number of threads per process: $N_{\mathrm{VP}} = N_{\mathrm{MPI}} \times N_{\mathrm{Thrd}}$. NEST ensures that for a given number of virtual processes N_{VP}, all simulations of a model yield identical results, independent of the combination of N_{MPI} and N_{Thrd}.

Neuron models are often stochastic, consuming many random numbers. To distribute the load of random number generation while obtaining identical simulation results for different combinations of N_{MPI} and N_{Thrd} on $N_{\mathrm{VP}} = \mathrm{const}$ virtual processes, we give each virtual process its own random number generator. By default we use a lagged Fibonacci generator, because it can be seeded to produce non-overlapping sequences of random numbers [5].

2.1 Nodes and Proxies

In a network with n nodes, each node is given a unique integer `gid` in order of creation, and is assigned to a virtual process such that `vid` = `gid` mod N_{VP}. Each virtual process manages the memory for its nodes. In addition, each virtual process has a vector of size n, called *node list*, with pointers to its own nodes and pointers to proxies for nodes that belong to other virtual process. Within an MPI process, we can collapse the node lists of all its virtual processes into a single list to conserve memory, as illustrated in figure 1.

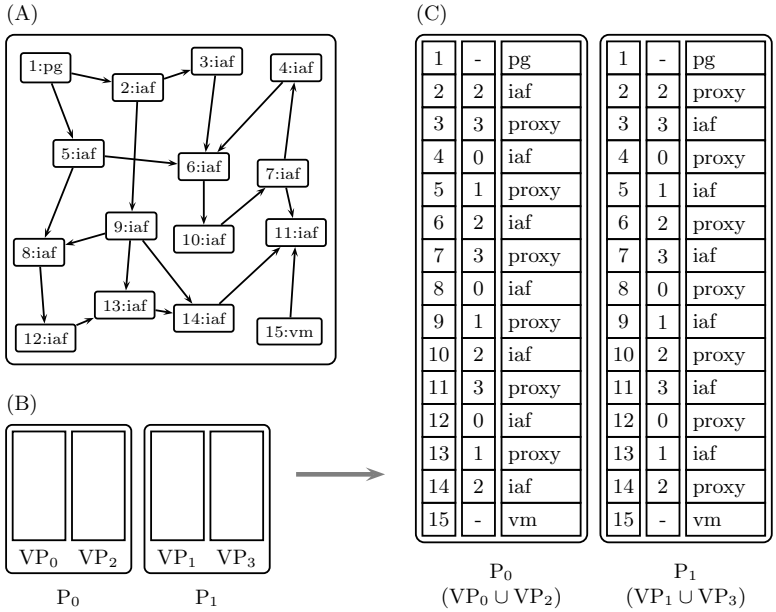


Fig. 1. Distributed and multi-threaded network representation. (A) The network as a directed graph. (B) Sketch of the distribution of four virtual processes onto two MPI processes, P_0 and P_1. (C) Collapsed node lists of the two MPI processes containing the nodes of two VPs each. The first column shows the node's `gid`, the second contains the VP a node is assigned to; '`-`' indicates that a node is created for each VP. The third column contains the node type: `pg`, Poisson spike generator device; `vm`, voltmeter device; `iaf`, integrate-and-fire model neurons.

This representation has the following advantages: (i) we can access each node directly with its `gid` as index to the node list on any virtual process; (ii) each virtual process knows the type of each node in the network; (iii) there is no memory overhead for pure multi-threaded simulations.

For a network of n nodes, each requiring S_N bytes of memory, and proxy nodes requiring S_P bytes, the required memory is given by:

$$M_{\text{nodes}} = \frac{n}{N_{\text{MPI}}} S_N + \left(n - \frac{n}{N_{\text{MPI}}} \right) S_P \ . \tag{1}$$

Typical values are $S_N = 480$ bytes and $S_P = 56$ bytes. Thus, for 10 and more MPI processes, more than half of all memory occupied by objects is occupied by proxy nodes, and more than 90% for more than 80 MPI processes. In absolute numbers, though, M_{nodes} is only around 70 MB for a network of 10^6 nodes distributed across 100 processes, which is negligible compared to the memory required for the connections in the network, cf. sec. 2.2. From a performance aspect, a node list filled with mostly proxies could become suboptimal if the node list became so large that it could no longer be cached efficiently. In this case, a fast hashing lookup may become more efficient.

Devices So far we assumed that nodes correspond to neurons. We will now discuss nodes representing devices, such as spike injectors and recorders. There are two types of devices: recording devices and stimulation devices. Recording devices measure the state of one or more neurons and write the data to disk. If nodes on different virtual processes are assigned to the same device, each of the virtual processes gets its own instance of the recording device. This has two advantages: (i) the measured data need not be transmitted between virtual processes; (ii) each device instance can write to its own local disk. Stimulation devices supply signals to one or more neurons, thereby manipulating their state. Again, each virtual process has its own instance of a given stimulation device to reduce the amount of data that must be exchanged between virtual processes. For some stimulation devices which produce random signals we must ensure that all instances in different virtual processes produce identical signals. Thus, these devices cannot use the random number generator of the virtual process, but have their own random number generators, initialized with identical seeds.

2.2 Connections

A connection between two nodes is defined by at least four numbers: the `gid` of the sending node, the `gid` of the receiving node, a weight, and a delay. More complicated connections have weights that change, depending on the activity of the connected nodes.

Different types of connections can be implemented by classes that derive from the abstract base class `Connector`. These can implement arbitrarily complex dynamics, provided they only depend on the previous state, the time since the last event, and information available from the target node. The most important applications are synaptic depression [6] and spike-timing dependent plasticity (STDP) [7]. An algorithm for STDP suitable for distributed computing can be found in [8].

Because the connections dominate the memory requirements of large networks, NEST splits them up such that each virtual process only stores the incoming connections to its own nodes. A vector inside each `Connector` contains pointers to all local target nodes, along with the delay (integer), and weight (double). The memory required for connections per MPI process is:

$$M_{\text{conn}} = \frac{n \times c \times S_C}{N_{\text{MPI}}} \,, \tag{2}$$

where c is the number of outgoing connections per node and S_C the memory per connection. For connections with constant weight and delay, $S_C = 32$ bytes; plastic synapses require more memory. A network of 10^6 nodes with 10^4 connections each thus requires 32 GB connection memory per process if distributed across 10 MPI processes, but only 3.2 GB per process if distributed across 100 processes.

A `ConnectionManager` stores the connections to all virtual processes of an MPI process in a three-dimensional data-structure. The first dimension is the thread number (virtual process) of the target node. The second dimension is the `gid` of the source node, and the third dimension is the index of the connection type. This memory layout has two advantages: (i) we can construct networks with heterogeneous synaptic dynamics; (ii) it is optimal for multi-threaded event delivery (cf. sec. 3.2) and the efficient implementation of synaptic dynamics [8].

3 Network Update and Event Exchange

Conceptually, NEST evaluates the network model on an evenly spaced time-grid $t_i := i \cdot \Delta$ and at each point, the network is in a well-defined state S_i. Starting at an initial state S_0, a global state transfer function $\mathrm{U}(S)$ propagates the system from one state to the next, such that $S_{t+\Delta} \leftarrow \mathrm{U}(S_t, \Delta)$. As a side effect of $\mathrm{U}(S_t)$, nodes create events that must be delivered to the target nodes after a delay that depends on the connection.

NEST evaluates a network model using the following algorithm:

1: $T \leftarrow 0$
2: **while** $T < T_{\text{stop}}$ **do**
3: **parallel on all** $vp \in N_{\text{VP}}$ **do**
4: deliver all events due
5: call $\mathrm{U}(S_T)$ for all nodes
6: **end parallel**
7: exchange events between VPs
8: increment network time: $T \leftarrow T + \Delta$
9: **end while**

The N_{VP} virtual processes evaluate steps 4 and 5 in parallel, and in step 6 they synchronize to exchange their events in step 7.

Although NEST uses a discrete event framework, it does not use a global event-driven update [9, 10]. Event-driven simulation assumes that the communication between nodes is rare and the update of a node is expensive. This does not

hold for biological networks, however: If a typical cortical neuron receives spikes from $\sim 10^4$ other neurons at a rate of ~ 10 Hz, the average interval between spike arrivals is ~ 0.01 ms. However, for most neuron models integration steps of $h \sim 0.1$ ms are sufficient. Thus event-driven update would need one order of magnitude *more* updates than time-driven update; see [11] for details.

In the following we describe in more detail (i) how to maximize the time increment Δ and (ii) how to collect, exchange, and deliver events between virtual processes.

3.1 Exploiting Delays for Cache-Efficient Update

Nodes affect each other by exchanging events that arrive at their destination with a delay $d_{ij} > 0$. The time period Δ is the largest permissible temporal desynchronization between any two nodes in the network. Δ may be increased as long as this does not change the order of events. This is equivalent to a system of distributed clocks that synchronize each other with events. Lamport showed that the smallest transmission delay $d_{\min}$ defines the interval at which clocks must be synchronized to maintain the order of events [12]. Accordingly, NEST sets Δ to $d_{\min}$. During this period, all nodes are effectively decoupled.

Most neural simulators use the integration step h of the neuron dynamics as the time increment. Maximizing Δ, typically to ~ 1 ms, i.e. about 10 times larger than h, has two advantages: (i) the virtual processes can run independently for a longer time, thereby reducing the number of synchronizations and thus the communication overhead; (ii) the state-update of each node can run a few tens of integration steps *en bloc*, keeping all required data in the CPU's L1 cache.

3.2 Global Event Exchange

NEST does not transmit individual events between virtual processes, as there are far too many. Instead, for each node that produced an event, the following information is transmitted: the `gid` of the sending node and the time at which the event occurred (address event representation [13]). All other connection parameters, such as the list of weights, delays and targets, are available at each virtual process. With this information, the virtual processes reconstruct the actual events and deliver them to their destinations.

We describe below the buffering and transmission of spike events constrained to a discrete time grid $t_n = nh$. This scheme is easily extended to spikes at arbitrary times [11].

Sender-Side Buffering Each MPI process has a three-dimensional buffer (*spike register*) to record the nodes that produced a spike-event during the update interval Δ. The first dimension represents the VP, so that they can write without collisions. The second dimension represents the time of the event with one entry per integration step h. The third dimension is a list of `gids`, one for each spike on a given thread at a given time. The total number of spikes per

virtual process per update interval is small: even assuming 10^6 neurons firing at 10 Hz and distributed across 20 VPs, only some 500 spikes occur per VP and update interval. With 4 threads per MPI process, the spike register occupies less than 20 kB.

Spike Exchange and Delivery Before spikes are exchanged between MPI processes, they are copied from the spike register to a communication buffer as follows: their `gid`s are written to the buffer, ordered by the integration time step at which the spikes were generated. Sentinels separate spikes generated during different steps. Since the number of integration steps per update cycle is fixed, the receiver can reconstruct the spike time from the sentinels. Each process also maintains buffers to receive the `gid`s from other processes. Once all buffers are set up, the spike buffers are exchanged between MPI processes by simultaneous pairwise exchange using CPEX [14–16].

Each virtual process delivers the spikes to its nodes in the parallel step 4 of the update algorithm (sec. 3). For each entry of the communication buffer, which now contains both local and remote spikes, it executes the following algorithm.

1: $n_{\text{sentinels}} \leftarrow 0$
2: **if** entry is sentinel **then**
3: $\quad n_{\text{sentinels}} \leftarrow n_{\text{sentinels}} + 1$
4: **else**
5: $\quad$ calculate t_{spike} from network time and $n_{\text{sentinels}}$
6: $\quad$ **for all** `tgt` $\in$ local targets **do**
7: $\quad\quad$ send spike time, weight, delay to `tgt`
8: $\quad\quad$ `tgt` stores spike in its ring buffer according to delay [14].
9: $\quad$ **end for**
10: **end if**

4 Performance

The scaling of large-scale simulations of neural networks depends significantly on the computational load of the individual neuron. The more complex the neuron, the better the scaling, as the ratio of local computation to communication costs increases. We therefore consider the following benchmark to be a hard problem in the field of distributed neural network simulations: the computation load is low, because the neuron and synapse models are simple, but the communication load is high, as the network has a biologically realistic connection density.

Benchmark Simulation The network consists of 12500 leaky integrate-and-fire neurons (80% excitatory, 20% inhibitory), each receiving input from 10% of all neurons, mediated by alpha-shaped current-injecting synapses with a synaptic delay of 1 ms (total number of synapses: 15.6×10^6). The neurons are initialized with random membrane potentials and receive a constant DC input adjusted to sustain asynchronous irregular firing at 12.7 Hz [17]. For a complete network specification and numerics, see [11].

Simulation times were measured on a cluster of Sun X4100 compute nodes equipped with two dual-core 2.4 GHz AMD Opteron 280 processors, 8GB RAM, and Mellanox MTS2400 Infiniband interconnect under SuSE Linux Enterprise Server 9 using the Scali MPI Connect 4.4 library. Threads were bound to CPU cores using the `taskset` command.

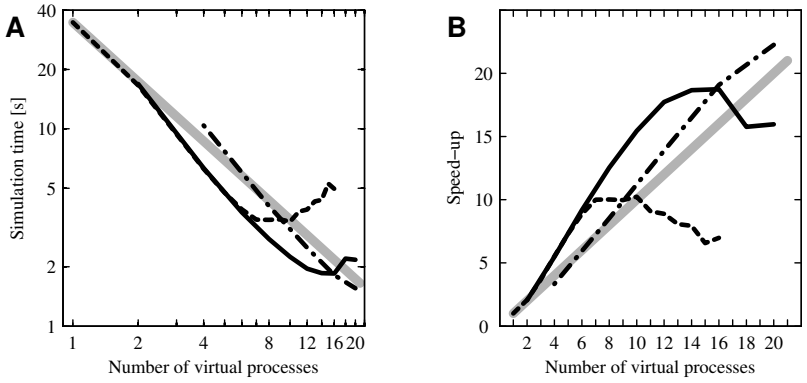


Fig. 2. Performance of different parallelization strategies as a function of the number of virtual processes. Single-thread MPI processes, dashed line; MPI processes with 2 threads, solid line; MPI processes with 4 threads, dash-dotted line. (A) Simulation time for one biological second in double-logarithmic representation. (B) Speed-up. The gray diagonal indicates the slope for a linear speed-up in both cases. Data obtained for simulations of 10 s biological time with a time step of 0.1 ms, averaged over 5 trials.

Results Figure 2 clearly demonstrates that the parallelization strategy significantly affects the scaling and absolute run-time of the simulation. A purely MPI-parallelized simulation shows supra-linear speed-up up to 8 virtual processes, rapidly saturates, and then undergoes a significant decrease in performance. The supra-linear speed-up is due to increasingly efficient caching [14], and the saturation in performance is due to the communication overhead.

By using a hybrid strategy with two threads per MPI process, such that both threads are bound to the same CPU, the number of MPI processes is halved. This reduces the number of send/receive operations per communication step by a factor of four and results in a performance which is better than the single-threaded case for numbers of virtual processes greater than eight. The performance of this hybrid strategy remains supra-linear up to 16 virtual processes, thus substantially reducing the absolute simulation time.

Reducing the number of MPI processes further by increasing the number of threads per MPI process to four leads to worse performance for small numbers of

virtual processors. This is due to the fact that memory allocation is performed by a single thread on each MPI process; as a result of the NUMA architecture, memory access is sub-optimal for the two threads on the non-allocating processor. The role of memory access is corroborated by simulating with two threads per MPI process as above, but binding the threads to different CPUs. This results in a performance which lies between that of the two-thread same-CPU variant discussed above and that of the four-thread variant (data not shown). This analysis is further supported by benchmarks performed on a Sun V40z server with four dual-core 2.2GHz AMD Opteron 875 processors, in which the threads used during simulation were placed at arbitrary cores relative to the thread constructing the network. Simulation times increased with increasing memory-access distance between the core used for construction and those used for simulation [18]. The costs of the sub-optimal memory access outweigh the benefits of decreasing the number of packets until 16 virtual processes, after which the four-thread variant becomes the most efficient simulation strategy.

5 Conclusions

Supra-linear scaling for a distributed biological neural network simulation was demonstrated for the first time in [14]. This result has since been confirmed by several other laboratories. In the present work we show that a hybrid approach to neural network simulation, combining multi-threading and distributed computing techniques, achieves an even better performance than a purely distributed solution. This suggests that the infrastructure of NEST is appropriate for future generations of multiprocessor, multi-core clusters.

The problem studied here was chosen to be particularly hard with respect to communication. In studies with larger neural networks or with more complex dynamics, NEST performance saturates at much larger numbers of processors: Simulation time for a network of 10^5 neurons with 10^9 synapses, driven by Poisson background input, shows supra-linear scaling up to 80 virtual processes on the same hardware. Other laboratories have shown good scaling of large-scale simulations on systems with thousands of processors, albeit on less hard problems [19, 20]. The scaling of NEST on such systems remains to be investigated.

The benchmarking results demonstrate the importance of sophisticated memory allocation on modern NUMA machines. Future work on NEST will be concerned with improving memory access times in a hybrid message-passing and multi-threading environment and further optimizing communication with respect to number of packets and latency hiding.

Acknowledgements HEP acknowledges Anita Woll for the execution of benchmark tests. Benchmarks were performed on supercomputing equipment at the Norwegian University of Life Sciences and the University of Freiburg. Partially funded by DAAD/NFR 313-PPP-N4-lk, DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, and EU Grant 15879 (FACETS).

References

1. Braitenberg, V., Schüz, A.: Cortex: Statistics and Geometry of Neuronal Connectivity. Second edn. Springer, Berlin (1998)
2. Gewaltig, M.O., Diesmann, M.: NEST. Scholarpedia (2007)
3. Message Passing Interface Forum: MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee (1994)
4. Lewis, B., Berg, D.J.: Multithreaded programming with pthreads. Sun Microsystems, Mountain View (1998)
5. Knuth, D.E.: The Art of Computer Programming. Third edn. Volume 2. Addison-Wesley, Reading, MA (1998)
6. Thomson, A.M., Deuchars, J.: Temporal and spatial properties of local circuits in neocortex. Trends Neurosci **17** (1994) 119–126
7. Bi, G.Q., Poo, M.M.: Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. J Neurosci **18** (1998) 10464–10472
8. Morrison, A., Aertsen, A., Diesmann, M.: Spike-time dependent plasticity in balance recurrent networks. Neural Comput **19** (2007) 1437–1467
9. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation. Second edn. Academic Press, Amsterdam (2000)
10. Brette, R., et al.: Simulation of networks of spiking neurons: A review of tools and strategies. J Comput Neurosci (in press) (2007)
11. Morrison, A., Straube, S., Plesser, H.E., Diesmann, M.: Exact subthreshold integration with continuous spike times in discrete time neural network simulations. Neural Comput **19** (2007) 47–79
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
13. Lazzaro, J.P., Wawrzynek, J., Mahowald., M., Sivilotti, M., Gillespie, D.: Silicon auditory processors as computer peripherals. IEEE Transactions on Neural Networks **4** (1993) 523–528
14. Morrison, A., Mehring, C., Geisel, T., Aertsen, A., Diesmann, M.: Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Comput **17** (2005) 1776–1801
15. Tam, A., Wang, C.: Efficient scheduling of complete exchange on clusters. In: 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000), Las Vegas (2000)
16. Gross, J., Yellen, J.: Graph Theory and its Applications. CRC Press (1999)
17. Brunel, N.: Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. J Comp Neurosci **8** (2000) 183–208
18. Woll, A.: Performance analysis of an MPI- and thread-parallel neural network simulator. Master's thesis, Norwegian University of Life Sciences (2007)
19. Djurfeldt, M., Johansson, C., Ekeberg, Ö., Rehn, M., Lundqvist, M., Lansner, A.: Massively parallel simulation of brain-scale neuronal network models. Technical Report Technical Report TRITA-NA-P0513, KTH, School of Computer Science and Communication Stockholm, Stockholm (2005)
20. Migliore, M., Cannia, C., Lytton, W.W., Markram, H., Hines, M.: Parallel network simulations with NEURON. J Comp Neurosci **21** (2006) 119–223

A.2 Multithreaded and distributed simulation of large biological neuronal networks

Jochen Martin Eppler[1,2], Hans Ekkehard Plesser[3], Abigail Morrison[4], Markus Diesmann[2,4], and Marc-Oliver Gewaltig[1,2], (2007) Multithreaded and Distributed Simulation of Large Biological Neuronal Networks. In F. Cappello, T. Herault, and J. J. Dongarra (Eds.), EuroPVM/MPI 2007: Recent advances in parallel virtual machine and message passing interface, Volume 4757 of Lecture Notes in Computer Science, Berlin, pp. 391-392. Springer-Verlag. doi:10.1007/978-3-540-75416-9_55.

[1] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[2] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

[3] Dept. of Mathematical Sciences and Technology
Norwegian University of Life Sciences
PO Box 5003, 1432 Ås, Norway

[4] Computational Neuroscience Group
RIKEN Brain Science Institute
Wako-shi, Saitama, Japan

Contributions

- I designed the data structures for network representation (nodes and connections) in the hybrid (message passing + threads) simulation kernel.
- I designed the data structures for the buffering of local events.
- I designed the local event delivery algorithms.
- I designed a system to allow the use of heterogeneous synapse types in simulations.
- I designed the message passing facilities to abstract the interaction of network elements like neurons from the underlying inter-process communication via MPI.
- I conceived a handshake algorithm to check compatibility between sending and receiving nodes during network creation.
- I wrote the text of the article.
- I presented the novel data structures and algorithms for the hybrid simulation engine at the EuroPVM/MPI 2007 meeting (short presentation + poster).

Multithreaded and Distributed Simulation of Large Biological Neuronal Networks

Jochen M. Eppler[1,4], Hans E. Plesser[2], Abigail Morrison[3], Markus Diesmann[3,4], and Marc-Oliver Gewaltig[1,4]

[1] Honda Research Institute, Offenbach/Main, Germany,
eppler@biologie.uni-freiburg.de
[2] Dept. of Mathematical Sciences and Technology, Norwegian University of Life Sciences, PO Box 5003, 1432 Ås, Norway,
[3] Computational Neuroscience Group, RIKEN Brain Science Institute, Wako-shi, Saitama, Japan
[4] Bernstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany

1 Introduction

To understand the principles of information processing in the brain, we depend on models with more than 10^5 neurons and 10^9 connections [1]. These networks can be described as graphs of threshold elements that exchange point events.

From the computer science perspective, the key challenges are to represent the connections succinctly and to transmit events and update neuron states efficiently. We present the Neural Simulation Tool NEST (www.nest-initiative.org, [2]), a neuronal network simulator which addresses all these requirements. To simulate very large networks in acceptable time and with acceptable memory requirements, NEST uses a hybrid strategy, combining distributed simulation across cluster nodes (MPI) with thread-based simulation on each computer.

2 Network Representation and Update

Conceptually, NEST represents the network as a list of nodes. Nodes are either neuron models, devices for recording and stimulation, or sub-networks and are assigned to one of N_{VP} virtual processes, using a simple modulo algorithm [3]. A virtual process (VP) is a POSIX thread that lives in one of N_{MPI} MPI processes. Each of the processes contains the same number of threads, N_{Thrd}. Device nodes are created for each virtual process to allow parallel data i/o. This is particularly important for device nodes that have to deliver large amounts of data to their targets. To balance the load of all virtual processes, neurons are only created on the virtual process they are assigned to. On all other virtual processes, they have light-weight proxies. Each node or proxy only stores the subset of connections that reach nodes (but not proxies) on the same virtual process. Thus, the network connections are also distributed, while cache problems are reduced to a minimum.

NEST evaluates the network model on an evenly spaced time-grid $t_i := i \cdot \Delta$, where Δ is determined by the shortest transmission delay in the system. At each

point, the network is in a well-defined state S_i. Starting at an initial state S_0, a global state transfer function $\mathrm{U}(S)$ propagates the system from one state to the next, such that $S_{t+\Delta} \leftarrow \mathrm{U}(S_t)$. As a side effect of $\mathrm{U}(S_t)$, nodes create events that must be delivered to the target nodes after a delay that depends on the connection. The network model in NEST is evaluated by executing the following algorithm:

1: $t \leftarrow 0$
2: **while** $t < T_{\mathrm{stop}}$ **do**
3: **parallel on all** VP **do**
4: deliver all events due
5: call $\mathrm{U}(S_t)$ for all nodes
6: **end parallel**
7: exchange events between VPs
8: increment network time: $t \leftarrow t + \Delta$
9: **end while**

The optimized data structures used for communication are described in [3].

3 Results

We demonstrate the performance of NEST, using a benchmark simulation of a large biological neural network model. We show that NEST scales supra-linearly for different combinations of threads and MPI processes.

On a cluster with 96 processor cores in 24 compute nodes and a central Infiniband switch we achieve real time with a network of 10^5 neurons with 10^9 synapses. On this architecture, the `MPI_Allgather` function performs better than the CPEX algorithm [4]. We are now investigating how different implementations of Allgather influence the performance of our multi-threaded/distributed communication scheme.

Acknowledgments This work was partially funded by DAAD/NFR 313-PPP-N4-lk, DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, and EU Grant 15879 (FACETS).

References

1. Abeles, M.: Corticonics: Neural Circuits of the Cerebral Cortex. 1st edn. Cambridge University Press, Cambridge (1991)
2. Gewaltig, M.O., Diesmann, M.: http://www.scholarpedia.org/article/NEST (Neural Simulation Tool). Scholarpedia (2007)
3. Morrison, A., Mehring, C., Geisel, T., Aertsen, A., Diesmann, M.: Advancing the boundaries of high connectivity network simulation with distributed computing. Neural Computation **17**(8) (2005) 1776–1801
4. Tam, A., Wang, C.: Efficient scheduling of complete exchange on clusters. In: 13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000), Las Vegas (2000)

A.3 PyNEST: A convenient interface to the NEST simulator

Jochen Martin Eppler[1,2], Moritz Helias[2], Eilif Muller[3], Markus Diesmann[2,4,5], and Marc-Oliver Gewaltig[1,2] (2009) PyNEST: A convenient interface to the NEST simulator. Frontiers in Neuroinformatics 2. doi:10.3389/neuro.11.012.2008.

[1] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[2] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

[3] Laboratory of Computational Neuroscience
Ecole Polytechnique Federale de Lausanne
Lusanne, Switzerland

[4] Theoretical Neuroscience Group
RIKEN Brain Science Institute
Wako City, Saitama, Japan

[5] Brain and Neural Systems Team
Computational Science Research Program
RIKEN Brain Science Institute
Wako City, Saitama, Japan

Contributions

- I designed the mapping of data types between Python and SLI.
- Together with Moritz Helias, I adapted the visitor pattern for the type conversion from SLI to Python.
- I designed a framework for exception handling for the interface enabling the propagation of C++ and SLI exceptions to the Python level.
- I designed and implemented the API for the high-level interface.
- I developed a testsuite for the interface which is integrated with the SLI-level testsuite.
- I developed a framework to integrate a Python-style build process (distutils) with the build process of NEST (autotools).
- I coordinated the preparation of the manuscript, wrote the major part of the text and created the figures.
- I coordinated the manuscript revisions towards acceptance as the corresponding author.
- I presented PyNEST at the FACETS student course "modeling for beginners" in 2007 (tutor), at the course "Python in computational neuroscience" 2008 (tutor), at the INCF 2008 congress (poster), at the CNS 2009 (invited talk), and at the INCF 2009 congress (demo session).

frontiers in
NEUROINFORMATICS

ORIGINAL RESEARCH ARTICLE
published: 29 January 2009
doi: 10.3389/neuro.11.012.2008

PyNEST: A convenient interface to the NEST simulator

Jochen Martin Eppler[1,2]*†, Moritz Helias[2†], Eilif Muller[3], Markus Diesmann[2,4,5] and Marc-Oliver Gewaltig[1,2]

[1] *Honda Research Institute Europe GmbH, Offenbach, Germany*
[2] *Bernstein Center for Computational Neuroscience, Albert-Ludwig University, Freiburg, Germany*
[3] *Laboratory for Computational Neuroscience, Swiss Federal Institute of Technology, EPFL, Lausanne, Switzerland*
[4] *Theoretical Neuroscience Group, RIKEN Brain Science Institute, Wako City, Japan*
[5] *Brain and Neural Systems Team, Computational Science Research Program, RIKEN, Wako City, Japan*

Edited by:
Rolf Kötter, Radboud University Nijmegen, The Netherlands

Reviewed by:
Upinder S. Bhalla, National Center for Biological Sciences, India
Terrence C. Stewart, Carleton University, Canada

****Correspondence:***
Jochen Martin Eppler, Honda Research Institute Europe GmbH, Carl-Legien-Str. 30, 63073 Offenbach am Main, Germany.
e-mail: eppler@biologie.uni-freiburg.de

†Eppler and Helias contributed equally to this work.

The neural simulation tool NEST (`http://www.nest-initiative.org`) is a simulator for heterogeneous networks of point neurons or neurons with a small number of compartments. It aims at simulations of large neural systems with more than 10^4 neurons and 10^7 to 10^9 synapses. NEST is implemented in C++ and can be used on a large range of architectures from single-core laptops over multi-core desktop computers to super-computers with thousands of processor cores. Python (`http://www.python.org`) is a modern programming language that has recently received considerable attention in Computational Neuroscience. Python is easy to learn and has many extension modules for scientific computing (e.g. `http://www.scipy.org`). In this contribution we describe PyNEST, the new user interface to NEST. PyNEST combines NEST's efficient simulation kernel with the simplicity and flexibility of Python. Compared to NEST's native simulation language SLI, PyNEST makes it easier to set up simulations, generate stimuli, and analyze simulation results. We describe how PyNEST connects NEST and Python and how it is implemented. With a number of examples, we illustrate how it is used.

Keywords: Python, modeling, integrate-and-fire neuron, large-scale simulation, scientific computing, networks, programming

INTRODUCTION

The first user interface for NEST (Gewaltig and Diesmann, 2007; Plesser et al., 2007) was the simulation language SLI, a stack-based language derived from PostScript (Adobe Systems Inc., 1999). However, programming in SLI turned out to be difficult to learn and users asked for a more convenient programming language for NEST.

When we decided to use Python as the new simulation language, it was almost unknown in Computational Neuroscience. In fact, Matlab (MathWorks, 2002) was far more common, both for simulations and for analysis. Other simulators, like e.g. CSIM (Natschläger, 2003), already used Matlab as their interface language. Thus, Matlab would have been a natural choice for NEST as well.

Python has a number of advantages over commercial software like Matlab and other free scripting languages like Tcl/Tk (Ousterhout, 1994). First, Python is installed by default on all Linux and Mac-OS based computers. Second, Python is stable, portable, and supported by a large and active developer community, and has a long history in scientific fields outside the neurosciences (Dubois, 2007). Third, Python is a powerful interactive programming language with a surprisingly concise and readable syntax. It supports many programming paradigms such as object-oriented and functional programming. Through packages like NumPy (`http://www.numpy.org`) and SciPy (`http://www.scipy.org`), Python supports scientific computing and visualization à la Matlab. Finally, a number of neuroscience laboratories meanwhile use Python for simulation and analysis, which further supports our choice.

Python is powerful at steering other applications and provides a well documented interface (API) to link applications to Python (van Rossum, 2008). To do so, it is common to map the application's functions and data structures to Python classes and functions. This approach has the advantage that the coupling between the application and Python is as tight as possible. But there is also a drawback: Whenever a new feature is implemented in the application, the interface to Python must be changed as well.

On many high-performance computers Python is not available and we have to preserve NEST's native simulation language SLI. In order to avoid two different interfaces, one to Python and one to SLI, we decided to deviate from the standard way of coupling applications to Python. Rather than using NEST's classes, we use NEST's simulation language as the interface: Python sends data and SLI commands to NEST and NEST responds with Python data structures.

Exchanging data between Python and NEST is easy since all important data types in NEST have equivalents in Python. Executing NEST commands from Python is also straightforward: Python only needs to send a string with commands to NEST, and NEST will execute them. With this approach, we only need to maintain one binary interface to the simulation kernel instead of two: Each new feature of the simulation kernel only needs to be mapped to SLI and immediately becomes accessible in PyNEST without changing its binary interface. This generic interpreter interface allows us to program PyNEST's high-level API in Python. This is an advantage, because programming in Python is more productive than programming in C++ (Prechelt, 2000). Python is also more expressive: A given number of lines of Python code achieve much more than the same number of lines in C++ (McConnell, 2004).

NEST users benefit from the increased productivity. They can now take advantage of the large number of extension modules for Python. NumPy is the Python interface to the BLAS libraries, the same libraries which power Matlab. Matplotlib (`http://matplotlib.sourceforge.net`) provides many routines to plot scientific data in publication quality. Many other packages exist to analyze and visualize data. Thus, PyNEST allows users to combine simulation, data analysis, and visualization in a single programming language.

In the Section "Using PyNEST", we introduce the basic modeling concepts of NEST. With a number of PyNEST code examples, we illustrate how simulations are defined and how the results are analyzed and plotted. In the Section "The Interface Between Python and NEST", we describe in detail how we bind NEST to the Python interpreter. In the Section "Discussion", we discuss our implementation and analyze its performance. The complete API reference for PyNEST is contained in Appendix A. In Appendix B we illustrate advanced PyNEST features, using a large scale model.

USING PyNEST

A neural network in NEST consists of two basic element types: Nodes and connections. Nodes are either neurons, devices or subnetworks. Devices are used to stimulate neurons or to record from them. Nodes can be arranged in subnetworks to build hierarchical networks like layers, columns, and areas. After starting NEST, there is one empty subnetwork, the so-called *root node*. New nodes are created with the command `Create()`, which takes the model name and optionally the number of nodes as arguments and returns a list of handles to the new nodes. These handles are integer numbers, called *id*s. Most PyNEST functions expect or return a list of ids (see Appendix A). Thus it is easy to apply functions to large sets of nodes with a single function call.

Nodes are connected using `Connect()`. Connections have a configurable delay and weight. The weight can be static or dynamic, as for example in the case of spike timing dependent plasticity (STDP; Morrison et al., 2008). Different types of nodes and connections have different parameters and state variables. To avoid the problem of *fat interfaces* (Stroustrup, 1997), we use *dictionaries* with the functions `GetStatus()` and `SetStatus()` for the inspection and manipulation of an element's configuration. The properties of the simulation kernel are controlled through the commands `GetKernelStatus()` and `SetKernelStatus()`. PyNEST contains the submodules *raster_plot* and *voltage_trace* to visualize spike activity and membrane potential traces. They use Matplotlib internally and are good templates for new visualization functions. However, it is not our intention to develop PyNEST into a toolbox for the analysis of neuroscience data; we follow the modularity concept of Python and leave this task to others (e.g. NeuroTools, `http://www.neuralensemble.org/NeuroTools`).

EXAMPLE

We illustrate the key features of PyNEST with a simulation of a neuron receiving input from an excitatory and an inhibitory population of neurons (modified from Gewaltig and Diesmann, 2007). Each presynaptic population is modeled by a Poisson generator, which generates a unique Poisson spike train for each target. The simulation adjusts the firing rate of the inhibitory input population such that the neurons of the excitatory population and the target neuron fire at the same rate.

First, we import all necessary modules for simulation, analysis and plotting.

```
1 from nest import *
2 from scipy.optimize import bisect
3 import nest.voltage_trace as plot
```

Second, the parameters for the simulation are set.

```
4 t_sim = 100000.0  #[ms] simulation time
5 n_ex  =  16000    #size of exc. population
6 n_in  =   4000    #size of inh. population
7 r_ex  =      5.0  #[Hz] rate of exc. neurons
8 epsc  =     45.0  #[pA] amplitude of exc.
9                   #synaptic currents
10 ipsc  =    -45.0  #[pA] amplitude of inh.
11                  #synaptic currents
12 d     =      1.0  #[ms] synaptic delay
13 lower =      5.0  #[Hz] lower bound of the
14                  #search interval
15 upper =     25.0  #[Hz] upper bound of the
16                  #search interval
17 prec  =     0.05 #accuracy goal (in percent
18                  #of inhibitory rate)
```

Third, the nodes are created using `Create()`. Its arguments are the name of the neuron or device model and optionally the number of nodes to create. If the number is not specified, a single node is created. `Create()` returns a list of ids for the new nodes, which we store in variables for later reference.

```
19 neuron        = Create("iaf_neuron")
20 noise         = Create("poisson_generator", 2)
21 voltmeter     = Create("voltmeter")
22 spikedetector = Create("spike_detector")
```

Fourth, the excitatory Poisson generator (`noise[0]`) and the voltmeter are configured using `SetStatus()`, which expects a list of node handles and a list of parameter dictionaries. The rate of the inhibitory Poisson generator is set in line 32. For the neuron and the spike detector we use the default parameters.

```
23 SetStatus([noise[0]], [{"rate": n_ex*r_ex}])
24 SetStatus(voltmeter, [{"interval": 1000.0,
25                        "withgid": True}])
```

Fifth, the neuron is connected to the spike detector and the voltmeter, as are the two Poisson generators to the neuron:

```
26 Connect(neuron, spikedetector)
27 Connect(voltmeter, neuron)
28 ConvergentConnect(noise, neuron,
29                    [epsc, ipsc], [d, d])
```

The command `Connect()` has different variants. Plain `Connect()` (line 26 and 27) just takes the handles of pre- and postsynaptic nodes and uses the default values for weight and delay. `ConvergentConnect()` (line 28) takes four arguments: A list of presynaptic nodes, a list of postsynaptic nodes, and lists of weights and delays. It connects all presynaptic nodes to each postsynaptic node. All variants of the `Connect()` command reflect the direction of signal flow in the simulation kernel rather than the physical process of inserting an electrode into a neuron. For example, neurons send their spikes to a spike detector, thus the neuron is the

first argument to `Connect()` in line 26. By contrast, a voltmeter polls the membrane potential of a neuron in regular intervals, thus the voltmeter is the first argument of `Connect()` in line 27. The documentation of each model explains the types of events it can send and receive.

To determine the optimal rate of the neurons in the inhibitory population, the network is simulated several times for different values of the inhibitory rate while measuring the rate of the target neuron. This is done until the rate of the inhibitory neurons is determined up to a given relative precision (`prec`), such that the target neuron fires at the same rate as the neurons in the excitatory population. The algorithm is implemented in two steps:

First, the function `output_rate()` is defined to measure the firing rate of the target neuron for a given rate of the inhibitory neurons.

```
30 def output_rate(guess):
31     rate = float(abs(n_in*guess))
32     SetStatus([noise[1]], [{"rate": rate}])
33     SetStatus(spikedetector, [{"n_events": 0}])
34     Simulate(t_sim)
35     n_events = GetStatus(spikedetector,
36                          "n_events")[0]
37     r_target = n_events*1000.0/t_sim
38     print "r_in=%.4f Hz," % guess,
39     print "r_target=%.3f Hz" % r_target
40     return r_target
```

The function takes the firing rate of the inhibitory neurons as an argument. It scales the rate with the size of the inhibitory population (line 31) and configures the inhibitory Poisson generator (`noise[1]`) accordingly (line 32). In line 33, the spike-counter of the spike detector is reset to zero. Line 34 simulates the network using `Simulate()`, which takes the desired simulation time in milliseconds and advances the network state by this amount of time. During the simulation, the spike detector counts the spikes of the target neuron and the total number is read out at the end of the simulation period (line 35). The return value of `output_rate()` is an estimate of the firing rate of the target neuron in Hz.

Second, we determine the optimal firing rate of the neurons of the inhibitory population using the bisection method.

```
41 print "Desired target rate: %.2f Hz" % r_ex
42 r = bisect(lambda x: output_rate(x)-r_ex,
43            lower, upper, rtol=prec)
44 print "Resulting inhibitory rate: %.4f" % r
```

The SciPy function `bisect()` takes four arguments: First a function whose zero crossing is to be determined. Here, the firing rate of the target neuron should equal the firing rate of the neurons of the excitatory population. Thus we define an anonymous function (using `lambda`) that returns the difference between the actual rate of the target neuron and the rate of the excitatory Poisson generator, given a rate for the inhibitory neurons. The next two arguments are the lower and upper bound of the interval in which to search for the zero crossing. The fourth argument of `bisect()` is the desired relative precision of the zero crossing.

Finally, we plot the target neuron's membrane potential as a function of time.

```
45 plot.from_device(voltmeter, timeunit="s")
```

A transcript of the simulation session and the resulting plot are shown in **Figure 1**.

PyNEST ON MULTI-CORE PROCESSORS AND CLUSTERS

NEST has built-in support for parallel and distributed computing (Morrison et al., 2005; Plesser et al., 2007): On multi-core processors, NEST uses POSIX threads (Lewis and Berg, 1997), on computer clusters, NEST uses the Message Passing Interface (MPI; Message Passing Interface Forum, 1994). Nodes and connections are assigned automatically to threads and processes, i.e. the same script can be executed single-threaded, multi-threaded, distributed over multiple processes, or using a combination of both methods. This naturally carries over to PyNEST: To use multiple threads for the simulation, the desired number has to be set prior to the creation of nodes and connections. Note that the network setup is carried out by a single thread, as only a single instance of the Python interpreter exists

A

```
jochen@winston: $ python balancedneuron.py
NEST 1.9.7865 (C) 2008 The NEST Initiative
Desired target rate:  5.00 Hz
  r_in=5.0000 Hz, r_target=434.580 Hz
  r_in=25.0000 Hz, r_target=0.020 Hz
  r_in=15.0000 Hz, r_target=347.410 Hz
  r_in=20.0000 Hz, r_target=34.350 Hz
  r_in=22.5000 Hz, r_target=0.000 Hz
  r_in=21.2500 Hz, r_target=0.680 Hz
  r_in=20.6250 Hz, r_target=7.160 Hz
                 ⋮
  r_in=20.7837 Hz, r_target=4.640 Hz
  r_in=20.7825 Hz, r_target=5.000 Hz
Resulting inhibitory rate:  20.7825 Hz
```

B

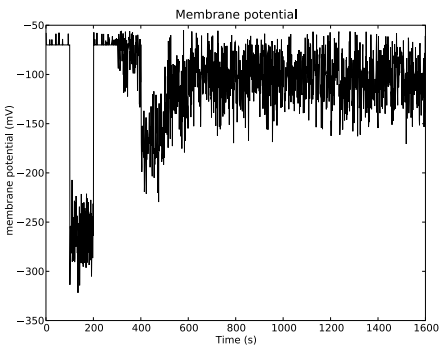


FIGURE 1 | Results of the example simulation. (A) The transcript of the simulation session shows the intermediate results of `r_target` as `bisect()` searches for the optimal rate. **(B)** The membrane potential of the target neuron as a function of time. Repeated adjustment of the spike rate of the inhibitory population by `bisect()` results in a convergence of the mean membrane potential to –112 mV, corresponding to an output spike rate of 5.0 Hz.

in each process. Only the simulation takes advantage of multiple threads. Distributed simulations can be run via the `mpirun` command of the respective MPI implementation. Where, for SLI, one would execute `mpirun -np n nest simulation.sli` to distribute a simulation onto `n` processes, one has to call `mpirun -np n python simulation.py` to get the same result with PyNEST. In the distributed case, `n` Python interpreters run in parallel and execute the same simulation script. This means that both network setup and simulation are parallelized. With third-party tools like IPython (`http://ipython.scipy.org`) or MPI for Python (`http://mpi4py.scipy.org`), it is possible to use PyNEST interactively even in distributed scenarios. For a more elaborate documentation of parallel and distributed simulations with NEST, see the NEST user manual (`http://www.nest-initiative.org`).

THE INTERFACE BETWEEN PYTHON AND NEST

NEST's built-in simulation language (SLI) is a stack-based language in which functions expect their arguments on an operand stack to which they also return their results. This means that in every expression, the arguments must be entered before the command that uses them (*reverse polish notation*). For many new users, SLI is difficult to learn and hard to read. This is especially true for math: The simple expression $\alpha = t \cdot e^{-t/\tau}$ has to be written as `/alpha t t neg tau div exp mul def` in SLI. But SLI is also a high-level language where functions can be assembled at run time, stored in variables and passed as arguments to other functions (functional programming; Finkel, 1996). Powerful indexing operators like `Part` and functional operators like `Map`, together with data types like heterogeneous arrays and dictionaries, allow a compact and expressive formulation of algorithms.

Stack-based languages are often used as intermediate languages in compilers and interpreters (Aho et al., 1988). This inspired us to couple NEST and Python using SLI as an intermediate language.

THE PyNEST LOW-LEVEL INTERFACE

The low-level API of PyNEST is implemented in C/C++ using the Python C-API (van Rossum, 2008). It exposes only three functions to Python, and has private routines for converting between SLI data types and their Python equivalents. The exposed functions are:

1. `sli_push(py_object)`, which converts the Python object `py_object` to the corresponding SLI data type and pushes it onto SLI's operand stack.
2. `sli_pop()`, which removes the top element from SLI's operand stack and returns it as a Python object.
3. `sli_run(slicommand)`, which uses NEST's simulation language interpreter to execute the string `slicommand`. If the command requires arguments, they have to be present on SLI's operand stack or must be part of `slicommand`. After the command is executed, its return values will be on the interpreter's operand stack.

Since these functions provide full access to the simulation language interpreter, we can now control NEST's simulation kernel without explicit Python bindings for all NEST functions. This interface also provides a natural way to execute legacy SLI code from within a PyNEST script by just using the command `sli_run("(legacy.sli) run")`. However, it does not provide any benefits over plain SLI from a syntactic point of view: All simulation specific code still has to be written in SLI. This problem is solved by a set of high-level functions.

THE PyNEST HIGH-LEVEL INTERFACE

To allow the researcher to define, run and evaluate NEST simulations using only Python, PyNEST offers convenient wrappers for the most important functions of NEST. These wrappers are implemented on top of the low-level API and execute appropriate SLI expressions. Thus, at the level of PyNEST, SLI is invisible to the user. Each high-level function consists essentially of three parts:

1. The arguments of the function are put on SLI's operand stack.
2. One or more SLI commands are executed to perform the desired action in NEST.
3. The results (if any) are fetched from the operand stack and returned as Python objects.

A concrete example of the procedure is given in the following listing, which shows the implementation of `Create()`:

```
1 def Create(model, n=1):
2     sli_run("/%s" % model)
3     sli_push(n)
4     sli_run("CreateMany")
5     lastid = sli_pop()
6     return range(lastid - n + 1, lastid + 1)
```

In line 2, we first transfer the model name to NEST. Model names in NEST have to be of type *literal*, a special symbol type that is not available in Python. Because of this, we cannot use `sli_push()` for the data transfer, but have to use `sli_run()`, which executes a given command string instead of just pushing it onto SLI's stack. The command string consists of a slash followed by the model name, which is interpreted as a literal by SLI. Line 3 uses `sli_push()` to transmit the number of nodes (`n`) to SLI. The nodes are then created by `CreateMany` in line 4, which expects the model name and number of nodes on SLI's operand stack and puts the id of the last created node back onto the stack. The id is retrieved in line 5 via `sli_pop()`. To be consistent with the convention that all PyNEST functions work with lists of nodes, we build a list of all created nodes' ids, which is returned in line 6.

A sequence diagram of the interaction between the different software layers of PyNEST is shown in **Figure 2** for a call to the `Create()` function.

DATA CONVERSION

From Python to SLI

The data conversion between Python and SLI exploits the fact that most data types in SLI have an equivalent type in Python. The function `sli_push()` calls `PyObjectToDatum()` to convert a Python object `py_object` to the corresponding SLI data type (see ② in **Figure 2**). `PyObjectToDatum()` determines the type of `py_object` in a cascade of type checks (e.g. `PyInt_Check()`, `PyString_Check()`, `PyFloatCheck()`) as described by van Rossum (2008). If a type check succeeds, the Python object is used to create a new

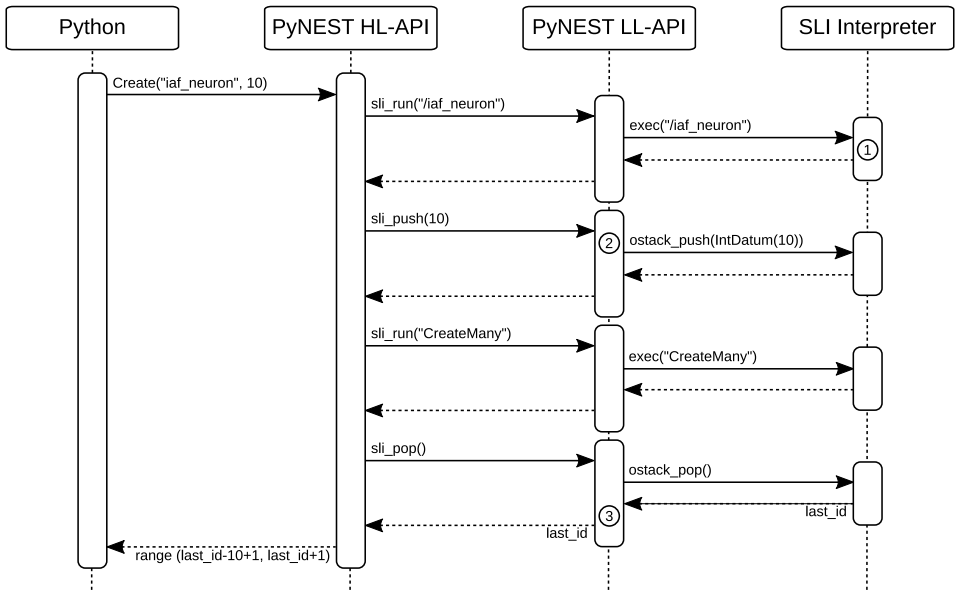


FIGURE 2 | Sequence diagram showing the interaction between Python and SLI. A call to the PyNEST high-level function `Create()` first transmits the model name to SLI using `sli_run()`. It is converted to the SLI type *literal* by the interpreter (①). Next, it pushes the number of nodes (10) to SLI using `sli_push()`. The PyNEST low-level API converts the argument to a SLI datum (②) and pushes it onto SLI's operand stack. Next, it executes appropriate SLI code to create the nodes of type `iaf_neuron` in the simulation kernel. Finally it retrieves the results of the NEST operations using `sli_pop()`, which converts the data back to a Python object (③). The result of the operation in SLI (the id of the last node created) is used to create a list with the ids of all new nodes, which is returned to Python.

SLI `Datum` of the respective type. `PyObjectToDatum()` is called recursively on the elements of lists and dictionaries. The listing below shows how this technique is used for the conversion of the Python type `float` and for NumPy arrays of `doubles`:

```
1 Datum* PyObjectToDatum(PyObject *py_object)
2 {
3    if (PyFloat_Check(py_object)) //float?
4    {
5     return new DoubleDatum(PyFloat_AsDouble(
6                                py_object));
7    }
8
9    if (PyArray_Check(py_object)) //NumPy array?
10   {
11    int size = PyArray_Size(py_object);
12    PyArrayObject *array;
13    array = (PyArrayObject*) py_object;
14    assert(array != 0);
15    switch (array->descr->type_num)
16    {
17     case PyArray_DOUBLE:
18     {
19       double *begin = (double*) array->data;
20       return new DoubleVectorDatum(
21           new std::vector<double>(
22                 begin, begin+size));
23      }
24     //cases for NumPy arrays of other types
25    }
26    }
27    //checks for other supported Python types
28 }
```

From SLI to Python

To convert a SLI data type to the corresponding Python type, we can avoid the cascade of type checks, since all SLI data types are derived from a common base class, called `Datum`. The C++ textbook solution would add a pure virtual conversion function `convert()` to the class `Datum`. Each derived class (e.g. `DoubleDatum`, `DoubleVectorDatum`) then overloads this function to implement its own conversion to the corresponding Python type. This approach is shown for the SLI type `DoubleDatum` in the following listing. The function `get()` is implemented in each `Datum` and returns its data member.

```
1 PyObject*
2 DoubleDatum::convert()
3 {
4   return PyFloat_FromDouble(get());
5 }
```

However, this solution would make SLI's type hierarchy (and thus NEST) depend on Python. To keep NEST independent of Python, we split the implementation in two parts: The first is Python-unspecific and resides in the NEST source code (**Figure 3**, left rectangle), the second is Python-specific and defined in the PyNEST source code (**Figure 3**, right rectangle).

We move the Python-specific conversion code from `convert()` to a new function `convert_me()`, which is then called by the

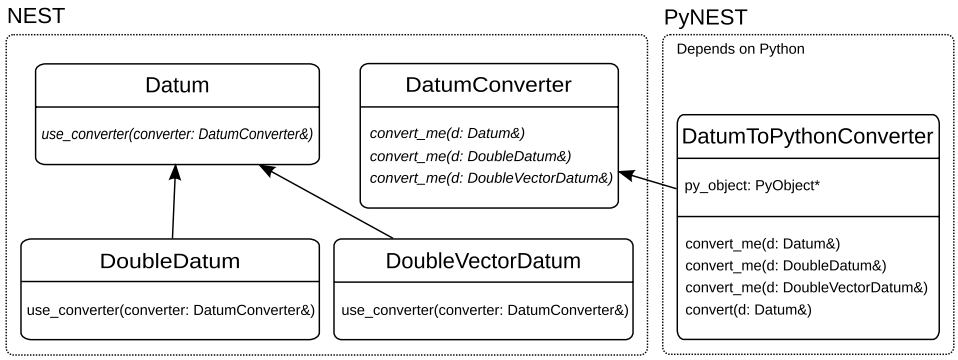


FIGURE 3 | Class diagram for the acyclic visitor pattern used to convert SLI types to Python types. The left rectangle contains classes belonging to NEST, the right rectangle contains classes that are part of PyNEST. All SLI data types are derived from the base class `Datum` and inherit its function `use_converter()`. The class `DatumConverter` is the base class of `DatumToPythonConverter`. The actual data conversion is carried out in one of `DatumToPythonConverter`'s `convert_me()` functions. Virtual functions are typeset in italics.

interface function `use_converter()`. This function is now independent of Python:

```
1 void
2 Datum::use_converter(DatumConverter& converter)
3 {
4   converter.convert_me(*this);
5 }
```

The function `use_converter()` is defined in the base class `Datum` and inherited by all derived classes. It calls the `convert_me()` function of `converter` that matches the type of the derived `Datum`. NEST's class `DatumConverter` is an abstract class that defines a pure virtual function `convert_me(T&)` for each SLI type T:

```
1 class DatumConverter
2 {
3  public:
4   virtual void convert_me(Datum&);
5   virtual void convert_me(DoubleDatum&)=0;
6   virtual void convert_me(DoubleVectorDatum&)=0;
7   //convert_me() function for other Datums
8 };
```

The Python-specific part of the conversion is encapsulated in the class `DatumToPythonConverter`, which derives from `DatumConverter` and implements the `convert_me()` functions to actually convert the SLI types to Python objects. `DatumToPythonConverter::convert_me()` takes a reference to the `Datum` as an argument and is overloaded for each SLI type. It stores the result of the conversion in the class variable `py_object`. An example for the conversion of `DoubleDatum` is given in the following listing:

```
1 void
2 DatumToPythonConverter::convert_me(
3     DoubleDatum& dd)
4 {
5   py_object = PyFloat_FromDouble(dd.get());
6 }
```

`DatumToPythonConverter` also provides the function `convert()`, which converts a given `Datum d` to a Python object by calling `d.use_converter()` with itself as an argument. It is used in the implementation of `sli_pop()` (see ③ in **Figure 2**). After the call to `use_converter()`, the result of the conversion is available in the member variable `py_object`, and is returned to the caller:

```
1 PyObject*
2 DatumToPythonConverter::convert(Datum& d)
3 {
4   d.use_converter(*this);
5   return py_object;
6 }
```

In the Computer Science literature, this method of decoupling different parts of program code is called the *acyclic visitor pattern* (Martin et al., 1998). Our implementation is based on Alexandrescu (2001).

As an example, the diagram in **Figure 4** illustrates the sequence of events in `sli_pop()`: First, `sli_pop()` retrieves a SLI `Datum d` from the operand stack (not shown). Second, it creates an instance of `DatumToPythonConverter` and calls its `convert()` function, which then passes itself as visitor to the `use_converter()` function of `d`. `Datum::use_converter()` calls the `DatumToPythonConverter`'s `convert_me()` function that matches the type of `d`. The function `convert_me()` then creates a new Python object from the data in `d` and stores it in the `DatumToPythonConverter`'s member variable `py_object`, which is returned to `sli_pop()`.

NumPy support

To make PyNEST depend on NumPy only if it is available, we use conditional compilation based on the preprocessor macro `HAVE_NUMPY`, which is determined during the configuration of PyNEST prior to compilation. For example, the following listing shows the implementation of the `DatumToPythonConverter::convert_me()` function to convert homogeneous arrays of doubles from SLI to Python. If NumPy is available during compilation, its

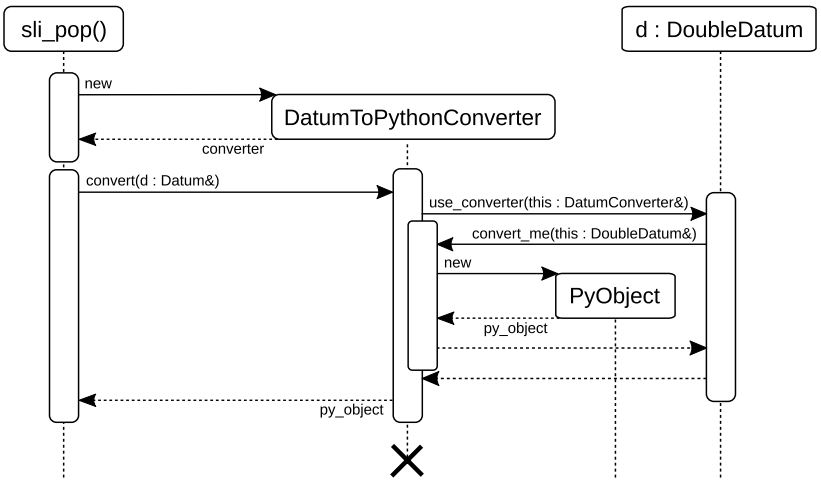


FIGURE 4 | Sequence diagram of the acyclic visitor pattern for data conversion from SLI to Python. For the conversion of a SLI datum `d`, `sli_pop()` creates an instance of `DatumToPythonConverter`. It then calls the `DatumToPythonConverter`'s `convert()` function, which passes itself as a visitor to the `use_converter()` function of `d`. `Datum::use_converter()` calls the `DatumToPythonConverter`'s `convert_me()` function that matches `d`'s type. `convert_me()` creates a new Python object from the data contained in `d`. The new Python object is returned to `sli_pop()`.

homogeneous array type is used to store the data. Without NumPy, a Python list is used instead.

```
1 void
2 DatumToPythonConverter::convert_me(
3     DoubleVectorDatum& d)
4 {
5   int dims = d->size();
6 #ifdef HAVE_NUMPY
7   PyArrayObject* array;
8   array = (PyArrayObject*)
9       PyArray_FromDims(1, &dims, PyArray_DOUBLE);
10  std::copy(d->begin(), d->end(),
11            (double*) array->data);
12  py_object = (PyObject*) array;
13 #else
14  py_object = PyList_New(dims);
15  for(int i=0; i<dims; i++)
16    PyList_SetItem(py_object, i,
17                   PyFloat_FromDouble((*d)[i]));
18 #endif
19 }
```

ERROR HANDLING

Error handling in NEST is implemented using C++ exceptions that are propagated up the calling hierarchy until a suitable error handler catches them. In this section, we describe how we extend this strategy to PyNEST.

PyNEST executes SLI code using `sli_run()` as described in the Section "The PyNEST High-Level Interface". However, the high-level API does not call `sli_run()` directly, but rather through the wrapper function `catching_sr()`:

```
1 def catching_sr(cmd):
2     sli_run("{" + cmd + "} runprotected")
3     if not sli_pop():   #cmd caused an error
4         errorname = sli_pop()
5         commandname = sli_pop()
6         raise NESTError("NEST error: " +
7                         errorname + " in " +
8                         commandname)
```

In line 2, `catching_sr()` converts the command string `cmd` to a SLI procedure by adding braces. It then calls the SLI command `runprotected` (see listing below), which executes the procedure in a `stopped` context (PostScript; Adobe Systems Inc., 1999). If an error occurs, `stopped` leaves the name of the failed command on the stack and returns `true`. In this case, `runprotected` extracts the name of the error from SLI's error dictionary, converts it to a string, and puts it back on the operand stack, followed by `false` to indicate the error condition to the caller. Otherwise, `true` is put on the stack. In case of an error, `catching_sr()` uses both the name of the command and the error to raise a Python exception (`NESTError`), which can be handled by the user's simulation code. The following listing shows the implementation of `runprotected`:

```
1 /runprotected
2 {
3   stopped dup
4   {
5     errordict /commandname get cvs
6     % tell NEST that the error was handled
7     errordict /newerror false put
8   } if
9   not
10 } def
```

Forwarding the original NEST errors to Python has the advantage that PyNEST functions do not have to check their arguments, because the underlying NEST functions already do. This makes the code of the high-level API more readable, while at the same time, errors are raised as Python exceptions without requiring additional

code. Moreover, this results in consistent error messages in NEST and PyNEST.

DISCUSSION

The previous sections describe the usage and implementation of PyNEST. Here we discuss consequences and limitations of the PyNEST implementation.

PERFORMANCE

The use of PyNEST entails a certain computational overhead over pure SLI-operated NEST. This overhead can be split into two main components:

1. Call overhead because of using SLI over direct access to the NEST kernel.
2. Data exchange between Python and NEST.

For most real-world simulations, the first is negligible, since the number of additional function calls is small. In practice, most overhead is caused by the second component, which we can reduce by minimizing the number of data conversions. For an illustration of the technique, see the following two listings that both add up a sequence of numbers in SLI. The first creates the sequence of numbers in Python, pushes them to SLI one after the other and lets SLI add them. Executing it takes approx. 15 s on a laptop with an Intel Core Duo processor at 1.83 GHz.

```
1 sli_push(0)
2 for i in range(1, 100001):
3     sli_push(i)
4     sli_run("add")
```

The second version computes the same result, but instead of creating the sequence in Python, it is created in SLI:

```
1 sli_run("0 1 1 100000 { add } for")
```

Although Python loops are about twice as fast as SLI loops, this version takes only 0.6 s, because of the reduced number of data conversions and, to a minor extent, the repeated parsing of the command string and the larger number of function calls in the first version.

The above technique is used in the implementation of the PyNEST high-level API wherever possible. The same technique is also applied for other loop-like commands (e.g. `Map`) that exist in both interpreters. However, it is important to note that the total run time of the simulation is often dominated by the actual creation and update of nodes and synapses, and by event delivery. These tasks take place inside of the optimized C++ code of NEST's simulation kernel, hence the choice between SLI or Python has no impact on performance.

INDEPENDENCE

One of the design decisions for PyNEST was to keep NEST independent of third-party software. This is important because NEST is used on architectures, where Python is not available or only available as a minimal installation. Moreover, since NEST is a long term project that has already seen several scripting languages and graphics libraries coming and going, we do not want to introduce a hard dependency on one or the other. The stand-alone version of NEST can be compiled without any third-party libraries. Likewise, the implementation of PyNEST does not depend on anything except Python itself. The use of NumPy is recommended, but optional. The binary part of the interface is written by hand and does not depend on interface generators like SWIG (`http://www.swig.org`) or third-party libraries like Boost.Python (`http://www.boost.org`). In our opinion, this strategy is important for the long-term sustainability of our scientific software.

EXTENSIBILITY

NEST can never provide all models and functions needed by every researcher. Extensibility is hence important.

Due to the asymmetry of the PyNEST interface (see "Assymmetry of the Interface"), neuron models, devices and synapse models have to be implemented in C++, the language of the simulation kernel. However, new analysis functions and connection routines can be implemented in either Python, SLI or C++, depending on the performance required and the skills of the user. The implementation in Python is easy, but performance may be limited. However, this approach is safe, as the real functionality is performed by SLI code, which is often well tested. To improve the performance, the implementation can be translated to SLI. This requires knowledge of SLI in addition to Python. Migrating the function down to the C++ level yields the highest performance gain, but requires knowledge of C++ and the internals of the simulation kernel.

Since the user can choose between three languages, it is easy to extend PyNEST, while at the same time, it is possible to achieve high performance if necessary. The hierarchy of languages also provides abstraction layers, which make it possible to migrate the implementation of a function between the different languages, without affecting user code. The intermediate layer of SLI allows the decoupling of the development of the simulation kernel from the development of the PyNEST API. This is also helpful for developers of abstraction libraries like PyNN (Davison et al., 2008), who only need limited knowledge of the simulation kernel.

ASSYMMETRY OF THE INTERFACE

Our implementation of PyNEST is asymmetric in that SLI code can be executed from Python, but NEST cannot respond, except for error handling and data exchange. Although this is sufficient to run NEST simulations from within a Python session, it could be beneficial to allow NEST to execute Python code: The user of PyNEST already knows the Python programming language, hence it might be easier to extend NEST in Python rather than to modify the C++ code of the simulation kernel. SciPy, NumPy and other packages provide well tested implementations of mathematical functions and numerical algorithms. Together with callback functions, these libraries would allow rapid prototyping of neuron and synapse models or to initialize parameters of neuron models or synapses according to complicated probability distributions: Python could be the middleware between NEST's simulation kernel and the numerical package. Using online feedback from the simulation, callback functions could also control simulations. Moreover, with a symmetric interface and appropriate Python modules it would be easier to add graphical user interfaces to NEST, along with online display of observables, and experiment management.

Different implementations of the symmetric interface are possible: One option is to pass callback functions from Python to NEST. Another option is to further exploit the idea that the "language is the protocol". In the same way as PyNEST generates SLI code, NEST would emit code for Python. Already Harrison and McLennan (1998) mention this technique, and in experimental implementations it was used successfully to symmetrically couple NEST with Tcl/Tk (Diesmann and Gewaltig, 2002), Mathematica, Matlab and IDL. The fact that none of these interfaces is still maintained confirms the conclusions of the Section "Independence".

LANGUAGE CONSIDERATIONS

At present, PyNEST maps NEST's capabilities to Python. Further advances in the expressiveness of the language may be easier to achieve at the level of Python or above (e.g. PyNN; Davison et al., 2008) without a counterpart in SLI. An example for this is the support of units for physical quantities as available in SBML (Hucka et al., 2002) or Brian (Goodman and Brette, 2008).

More generally, the development of simulation tools has not kept up with the increasing complexity of network models. As a consequence the reliable documentation of simulation studies is challenging and laboratories notoriously have difficulties in reproducing published results (Djurfeldt and Lansner, 2007). One component of a solution is the ability to concisely formulate simulations in terms of the neuroscientific problem domain like connection topologies and probability distributions. At present little research has been carried out on the particular design of such a language (Davison et al., 2008; Nordlie et al., 2008), but a general purpose high-level language interface to the simulation engine is a first step towards this goal.

APPENDIX

A. PyNEST API REFERENCE

Models

`Models(mtype="all", sel=None)`: Return a list of all available models (nodes and synapses). Use `mtype="nodes"` to only see node models, `mtype="synapses"` to only see synapse models. `sel` can be a string, used to filter the result list and only return models containing it.

`GetDefaults(model)`: Return a dictionary with the default parameters of the given `model`, specified by a string.

`SetDefaults(model, params)`: Set the default parameters of the given `model` to the values specified in the `params` dictionary.

`GetStatus(model, keys=None)`: Return a dictionary with status information for the given `model`. If `keys` is given, a value is returned instead. `keys` may also be a list, in which case a list of values is returned.

`CopyModel(existing, new, params=None)`: Create a `new` model by copying an `existing` one. Default parameters can be given as `params`, or else are taken from `existing`.

Nodes

`Create(model, n=1, params=None)`: Create `n` instances of type `model` in the current subnetwork. Parameters for the new nodes can be given as `params` (a single dictionary, or a list of dictionaries with size `n`). If omitted, the `model`'s defaults are used.

`GetStatus(nodes, keys=None)`: Return a list of parameter dictionaries for the given list of `nodes`. If `keys` is given, a list of values is returned instead. `keys` may also be a list, in which case the returned list contains lists of values.

`SetStatus(nodes, params, val=None)`: Set the parameters of the given `nodes` to `params`, which may be a single dictionary, or a list of dictionaries of the same size as `nodes`. If `val` is given, `params` has to be the name of a property, which is set to `val` on the `nodes`. `val` can be a single value, or a list of the same size as `nodes`.

Connections

`Connect(pre, post, params=None, delay=None, model="static_synapse")`: Make one-to-one connections of type `model` between the nodes in `pre` and the nodes in `post`. `pre` and `post` have to be lists of the same length. If `params` is given (as a dictionary or as a list of dictionaries with the same size as `pre` and `post`), they are used as parameters for the connections. If `params` is given as a single float, or as a list of floats of the same size as `pre` and `post`, it is interpreted as weight. In this case, `delay` also has to be given (as a float, or as a list of floats with the same size as `pre` and `post`).

`ConvergentConnect(pre, post, weight=None, delay=None, model="static_synapse")`: Connect all nodes in `pre` to each node in `post` with connections of type `model`. If `weight` is given, `delay` also has to be given. Both can be specified as a float, or as a list of floats with the same size as `pre`.

`RandomConvergentConnect(pre, post, n, weight=None, delay=None, model="static_synapse")`: Connect `n` randomly selected nodes from `pre` to each node in `post` with connections of type `model`. Presynaptic nodes are drawn independently for each postsynaptic node. If `weight` is given, `delay` also has to be given. Both can be specified as a float, or as a list of floats of size `n`.

`DivergentConnect(pre, post, weight=None, delay=None, model="static_synapse")`: Connect each node in `pre` to all nodes in `post` with connections of type `model`. If `weight` is given, `delay` also has to be given. Both can be specified as a float, or as a list of floats with the same size as `post`.

`RandomDivergentConnect(pre, post, n, weight=None, delay=None, model="static_synapse")`: Connect each node in `pre` to `n` randomly selected nodes from `post` with connections of type `model`. If `weight` is given, `delay` also has to be given. Both can be specified as a float, or as a list of floats of size `n`.

Structured networks

`CurrentSubnet()`: Return the id of the current subnetwork.

`ChangeSubnet(subnet)`: Make `subnet` the current subnetwork.

`GetLeaves(subnet)`: Return the ids of all nodes under `subnet` that are not subnetworks.

`GetNodes(subnet)`: Return the complete list of `subnet`'s children (including subnetworks).

`GetNetwork(subnet, depth)`: Return a nested list of `subnet`'s children up to `depth` (including subnetworks).

`LayoutNetwork(model, shape, label=None, customdict=None)`: Create a subnetwork of shape `shape` that contains nodes of type `model`. `label` is an optional name for the subnetwork. If present, `customdict` is set as custom dictionary of

the subnetwork, which can be used by the user to store custom information.

`BeginSubnet(label=None, customdict=None)`: Create a new subnetwork and change into it. `label` is an optional name for the subnetwork. If present, `customdict` is set as custom dictionary of the subnetwork, which can be used by the user to store custom information.

`EndSubnet()`: Change to the parent subnetwork and return the id of the subnetwork just left.

Simulation control

`Simulate(t)`: Simulate the network for `t` milliseconds.

`ResetKernel()`: Reset the simulation kernel. This will destroy the network as well as all custom models created with `CopyModel()`. The parameters of built-in models are reset to their defaults. Calling this function is equivalent to restarting NEST.

`ResetNetwork()`: Reset all nodes and connections to the defaults of their respective model.

`SetKernelStatus(params)`: Set the parameters of the simulation kernel to the ones given in `params`.

`GetKernelStatus()`: Return a dictionary with the parameters of the simulation kernel.

`PrintNetwork(depth=1, subnet=None)`: Print the network tree up to `depth`, starting at `subnet`. If `subnet` is omitted, the current subnetwork is used instead.

B. ADVANCED EXAMPLE

In the Section "Using PyNEST", we introduced the main features of PyNEST with a short example. This section contains a simulation of a balanced random network of 10,000 excitatory and 2,500 inhibitory integrate-and-fire neurons as described in Brunel (2000). We start with importing the required modules.

```
1 from nest import *
2 import nest.raster_plot as plot
3 import time
```

We store the current time at the start of the simulation.

```
4 startbuild = time.time()
```

Next, we use `SetKernelStatus()` to set the temporal resolution for the simulation to 0.1 ms.

```
5 SetKernelStatus({"resolution": 0.1})
```

We define variables for the simulation duration, the network size and the number of neurons to be recorded.

```
6 simtime =   500.0 #[ms] Simulation time
7 NE      = 10000   #number of exc. neurons
8 NI      =  2500   #number of inh. neurons
9 N_rec   =    50   #record from 50 neurons
```

The following are the parameters of the integrate-and-fire neuron that deviate from the defaults.

```
10 tauMem = 20.0 #[ms] membrane time constant
11 theta  = 20.0 #[mV] threshold for firing
12 t_ref  =  2.0 #[ms] refractory period
13 E_L    =  0.0 #[mV] resting potential
```

The synaptic delay and weights and the number of afferent synapses per neuron are assigned to variables. By choosing the relative strength of inhibitory connections to be $|J_{in}| / |J_{ex}| = g = 5.0$, the network is in the inhibition-dominated regime.

```
14 delay   = 1.5             #[ms] synaptic delay
15 J_ex    = 0.1             #[mV] exc. synaptic strength
16 g       = 5.0             #ratio between inh. and exc.
17 J_in    = -g*J_ex         #[mV] inh. synaptic strength
18 epsilon = 0.1             #connection probability
19 CE      = int(epsilon*NE) #exc. synapses/neuron
20 CI      = int(epsilon*NI) #inh. synapses/neuron
```

To reproduce Figure 8C from Brunel (2000), we choose parameters for asynchronous, irregular firing: ν_θ denotes the external Poisson rate which results in a mean free membrane potential equal to the threshold. We set the rate of the external Poisson input to $\nu_{ext} = \eta\nu_\theta = 2\nu_\theta$.

```
21 eta    = 2.0                #fraction of ext. input
22 nu_th  = theta/(J_ex*tauMem) #[kHz] ext. rate
23 nu_ext = eta*nu_th          #[kHz] exc. ext. rate
24 p_rate = 1000.0*nu_ext      #[Hz] ext. Poisson rate
```

In the next step we set up the populations of excitatory (`nodes_ex`) and inhibitory (`nodes_in`) neurons. The neurons of both pools have identical parameters, which are configured for the model with `SetDefaults()`, before creating instances with `Create()`.

```
25 print "Creating network nodes ..."
26 SetDefaults("iaf_psc_delta", {"C_m"  : tauMem,
27                                "tau_m": tauMem,
28                                "t_ref": t_ref,
29                                "E_L"  : E_L,
30                                "V_th" : theta})
31 nodes_ex = Create("iaf_psc_delta", NE)
32 nodes_in = Create("iaf_psc_delta", NI)
33 nodes = nodes_ex+nodes_in
```

Next, a Poisson spike generator (`noise`) is created and its rate is set. We use it to provide external excitatory input to the network.

```
34 noise = Create("poisson_generator",
35                 params={"rate": p_rate})
```

The next paragraph creates the devices for recording spikes from the excitatory and inhibitory population. The spike detectors are configured to record the spike times and the id of the sending neuron to a file.

```
36 SetDefaults("spike_detector", {"withtime": True,
37                                 "withgid" : True,
38                                 "to_file" : True})
39 espikes = Create("spike_detector")
40 ispikes = Create("spike_detector")
```

Next, we use `CopyModel()` to create copies of the synapse model `"static_synapse"`, which are used for the excitatory and inhibitory connections.

```
41 SetDefaults("static_synapse", {"delay": delay})
42 CopyModel("static_synapse", "excitatory",
43           {"weight": J_ex})
44 CopyModel("static_synapse", "inhibitory",
45           {"weight": J_in})
```

A

```
jochen@winston: $ python brunel.py
NEST 1.9.7753 (C) 2008 The NEST Initiative
Creating network nodes ...
Connecting network ...
Simulating 500.0 ms ...
Brunel network simulation using PyNEST:
Number of neurons :  12500
Number of synapses:  15637600
       Excitatory :  12512600
       Inhibitory :  3125000
Excitatory rate   :  31.52 Hz
Inhibitory rate   :  31.96 Hz
Building time     :  34.06 s
Simulation time   :  78.88 s
```

B

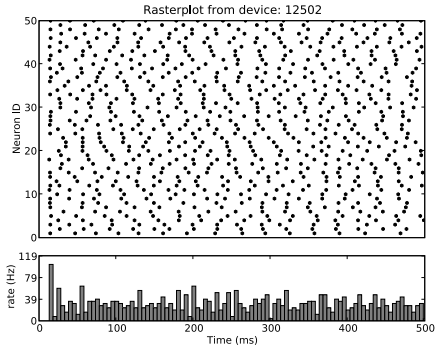


FIGURE 5 | Results of the balanced random network simulation. (A) The transcript of the simulation session shows the output during network setup and the summary printed at the end of the simulation. **(B)** Spike raster (top) and spike time histogram (bottom) of the `N_rec` recorded excitatory neurons.

The following code connects neurons and devices. `DivergentConnect()` connects one source node with each of the given target nodes and is used to connect the Poisson generator (`noise`) to the excitatory and the inhibitory neurons (`nodes`). `ConvergentConnect()` is used to connect the first `N_rec` excitatory and inhibitory neurons to the corresponding spike detectors.

```
46 print "Connecting network ..."
47 DivergentConnect(noise, nodes,
48                   model="excitatory")
49 ConvergentConnect(nodes_ex[:N_rec], espikes,
50                    model="excitatory")
51 ConvergentConnect(nodes_in[:N_rec], ispikes,
52                    model="excitatory")
```

The following lines connect the neurons with each other. The function `RandomConvergentConnect()` draws `CE` presynaptic neurons randomly from the given list (first argument) and connects them to each postsynaptic neuron (second argument). The presynaptic neurons are drawn repeatedly and independent for each postsynaptic neuron.

```
53 RandomConvergentConnect(nodes_ex, nodes, CE,
54                         model="excitatory")
55 RandomConvergentConnect(nodes_in, nodes, CI,
56                         model="inhibitory")
```

To calculate the duration of the network setup later, we again store the current time.

```
57 endbuild = time.time()
```

We use `Simulate()` to run the simulation.

```
58 print "Simulating", simtime, "ms ..."
59 Simulate(simtime)
```

Again, we store the time to calculate the runtime of the simulation later.

```
60 endsimulate = time.time()
```

The following code calculates the mean firing rate of the excitatory and the inhibitory neurons, determines the total number of synapses, and the time needed to set up the network and to simulate it. The firing rates are calculated from the total number of events received by the spike detectors. The total number of synapses is available from the status dictionary of the respective synapse models.

```
61 events_ex   = GetStatus(espikes, "n_events")[0]
62 rate_ex     = event_ex/simtime*1000.0/N_rec
63 events_in   = GetStatus(ispikes, "n_events")[0]
64 rate_in     = events_in/simtime*1000.0/N_rec
65 synapses_ex = GetStatus("excitatory",
66                           "num_connections")
67 synapses_in = GetStatus("inhibitory",
68                           "num_connections")
69 synapses    = synapses_ex+synapses_in
70 build_time  = endbuild-startbuild
71 sim_time    = endsimulate-endbuild
```

The next lines print a summary with network and runtime statistics.

```
72 print "Brunel network simulation using PyNEST:"
73 print "Number of neurons :", len(nodes)
74 print "Number of synapses:", synapses
75 print "       Exitatory  :", synapses_ex
76 print "       Inhibitory :", synapses_in
77 print "Excitatory rate   : %.2f Hz" % rate_ex
78 print "Inhibitory rate   : %.2f Hz" % rate_in
79 print "Building time     : %.2f s" % build_time
80 print "Simulation time   : %.2f s" % sim_time
```

Finally, `nest.raster_plot` is used to visualize the spikes of the `N_rec` selected excitatory neurons, similar to Figure 8C of Brunel (2000).

```
81 plot.from_device(espikes, hist=True)
```

The resulting plot is shown in **Figure 5** together with a transcript of the simulation session. The simulation was run on a laptop with an Intel Core Duo processor at 1.83 GHz and 1.5 GB of RAM.

ACKNOWLEDGMENTS

We are grateful to our colleagues in the NEST Initiative and the FACETS project for stimulating discussions, in particular to Hans

Ekkehard Plesser for drawing our attention to the visitor pattern. Partially funded by DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grant 15879 (FACETS), and "The Next-Generation Integrated Simulation of Living Matter" project, part of the Development and Use of the Next-Generation Supercomputer Project of the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

REFERENCES

Adobe Systems Inc. (1999). Postscript Language Reference Manual, third edn. Reading, MA, Addison-Wesley.

Aho, A. V., Sethi, R., and Ullman, J. D. (1988). Compilers, Principles, Techniques, and Tools. Reading, MA, Addison-Wesley.

Alexandrescu, A. (2001). Modern C++ Design. Boston, Addison-Wesley.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 183–208.

Davison, A., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinformatics* 2. doi: 10.3389/neuro.11.011.2008.

Diesmann, M., and Gewaltig, M.-O. (2002). NEST: an environment for neural systems simulations. In Forschung und wisschenschaftliches Rechnen, Beitrage zum Heinz-Billing-Preis 2001, Vol. 58 of GWDG-Bericht, T. Plesser and V. Macho, eds (Gottingen, Ges. für Wiss. Datenverarbeitung), pp. 43–70.

Djurfeldt, M., and Lansner, A. (2007). Workshop report: 1st INCF workshop on large-scale modeling of the nervous system. *Nature Precedings*, doi: 10.1038/npre.2007.262.1.

Dubois, P. F. (2007). Guest editor's introduction: Python: batteries included. *Comput. Sci. Eng.* 9, 7–9.

Finkel, R. A. (1996). Advanced Programming Languages. Menlo Park, CA, Addison-Wesley.

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (Neural Simulation Tool). *Scholarpedia* 2, 1430.

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinformatics* 2. doi: 10.3389/neuro.11.005.2008.

Harrison, M., and McLennan, M. (1998). Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk. Reading, MA, Addison-Wesley.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A. et al. (2002). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* 19, 524–531.

Lewis, B., and Berg, D. J. (1997). Multithreaded Programming With PThreads. Upper Saddle River: Sun Microsystems Press.

Martin, R. C., Riehle, D., and Buschmann, F. (eds) (1998). Pattern Languages of Program Design 3. Reading, MA, Addison-Wesley.

MathWorks (2002). MATLAB The Language of Technical Computing: Using MATLAB. Natick, MA, 3 Apple Hill Drive.

McConnell, S. (2004). Code Complete: A practical Handbook of Software Construction. 2nd edn. Redmond, WA, Microsoft Press.

Message Passing Interface Forum (1994). MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230.

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike-timing. *Biol. Cybern.* 98, 459–478.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.

Natschläger, T. (2003). CSIM: A Neural Circuit SIMulator. Technical report.

Nordlie, E., Plesser, H. E., and Gewaltig, M.-O. (2008). Towards reproducible descriptions of neuronal network models. Volume Conference Abstract: Neuroinformatics 2008. doi: 10.3389/conf.neuro.11.2008.01.086.

Ousterhout, J. K. (1994). Tcl and the Tk Toolkit. Professional Computing. Reading Massachusetts: Addison-Wesley.

Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., and Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In Euro-Par 2007: Parallel Processing, Volume 4641 of Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bouge, and T. Priol, eds (Berlin, Springer-Verlag), pp. 672–681.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *COMPUTER* 33, 23–29.

Stroustrup, B. (1997). The C++ Programming Language, 3rd edn. New York, Addison-Wesely.

van Rossum, G. (2008). Python/C API Reference Manual. Available at: http://docs.python.org/api/api.html.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 14 September 2008; paper pending published: 29 September 2008; accepted: 30 December 2008; published online: 29 January 2009.

Citation: Eppler JM, Helias M, Muller E, Diesmann M and Gewaltig M-O (2009) PyNEST: a convenient interface to the NEST simulator. Front. Neuroinform. (2009) 2:12. doi: 10.3389/neuro.11.012.2008

A.4 A Python interface to NEST

Jochen Martin Eppler[1,2] (2009) PyNEST: A convenient interface to NEST. The Neuromorphic Engineer. doi:10.2417/1200912.1703.

[1] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[2] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

Contributions

- I designed the mapping of data types between Python and SLI.
- Together with Moritz Helias, I adapted the visitor pattern for the type conversion from SLI to Python.
- I designed a framework for exception handling for the interface enabling the propagation of C++ and SLI exceptions to the Python level.
- I designed and implemented the API for the high-level interface.
- I developed a testsuite for the interface which is integrated with the SLI-level testsuite.
- I developed a framework to integrate a Python-style build process (distutils) with the build process of NEST (autotools).
- I coordinated the preparation of the manuscript, wrote the text and created the figure.
- I coordinated the manuscript revisions towards acceptance as the corresponding author.
- I presented PyNEST at the FACETS student course "modeling for beginners" in 2007 (tutor), at the course "Python in computational neuroscience" 2008 (tutor), at the INCF 2008 congress (poster), at the CNS 2009 (invited talk), and at the INCF 2009 congress (demo session).

ine Institute of Neuromorphic Engineering

THE NEUROMORPHIC ENGINEER
A PUBLICATION OF INE-WEB.ORG

10.2417/1200912.1703

A Python interface to NEST

Jochen Martin Eppler

With PyNEST, stimulus generation, simulation and data analysis can be performed in a single programming language.

NEST[1] is a simulator for large networks of spiking neurons, and has a long history of use in computational neuroscience for the simulation of large spiking networks. It runs on many different architectures (ranging from normal desktop computers to computer clusters with thousands of processor cores[2]), is written in C++, and has a built-in simulation language interpreter (SLI) to help the user set up the network. NEST's simulation language is stack-based and inspired by PostScript,[3] which means that each function expects its arguments to be on the stack and returns results back to it. SLI is a high-level language with functional operators like `Map` and data-structures like associative arrays. However, learning SLI has turned out to be difficult for many users: a more convenient simulation language was required.

When we were thinking about a new scripting interface for NEST, Python was almost unknown in computational neuroscience. However, we noticed a strong trend towards it in the scientific community in general.[4] Python has a number of advantages over commercial programming environments like Matlab[5] or Mathematica:[6] it is installed on almost all Linux and MacOS-based computers, is free, and is being developed by an active community. Thanks to the multitude of packages for scientific computing (`http://www.scipy.org`), Python can be used for stimulus generation, data analysis, and plotting in the same way its commercial alternatives can. As a result, a number of other neuroscience laboratories are also using Python.[7]

The usual approach to creating Python interfaces for existing software is to create a wrapper library that exposes all data structures and functions of the application to Python. We decided to deviate from this approach and keep the existing SLI interface as an intermediate layer between the new user interface and the simulation engine. The reason for this is threefold: first, a lot of SLI code has already been written, and we did not want to render this code useless with newer versions of NEST. Second, Python is not yet available on some exotic hardware platforms, which we still need to use. Third, NEST needs to remain independent of third party software to guarantee long-term sustainability.

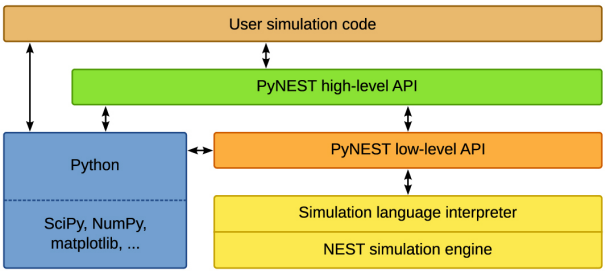


Figure 1. *Shown is the NEST architecture. The lowest level is the simulation engine, which is used by the simulation language interpreter and by the PyNEST low-level API. The PyNEST high-level API uses the low-level API to communicate with the simulation engine. The user's simulation code can use functions from PyNEST, from Python, and from its extension modules.*

As a result, the PyNEST interface consists of two separate layers: the low-level API, written using the Python C API,[8] is responsible for the data conversion from SLI to Python and back. It provides access to SLI with only three functions: `sli_push()` for pushing data onto the operand stack, `sli_pop()` for getting data from the SLI stack back to Python, and `sli_run()` for executing SLI commands. The high-level API uses these functions of the low-level API to provide Python versions of all important SLI commands. The functions of the high-level API are used by the user's simulation code. The complete architecture of PyNEST is shown in Figure 1.

SLI already provides all the necessary commands to build and simulate a neural network. Thus, we can create a complete Python interface to NEST by creating wrapper functions that just call the respective functions in SLI. This technique is illustrated in the following listing, which defines a function that returns the list of available models:

```
def Models():
    sli_run("modeldict")
    return sli_pop().keys()
```

In general, each function has three parts: First, we push the arguments onto the SLI stack with `sli_push()`. Second, we execute one or more SLI commands to perform the desired action

Continued on next page

Institute of Neuromorphic Engineering

THE NEUROMORPHIC ENGINEER
A PUBLICATION OF INE-WEB.ORG

10.2417/1200912.1703 Page 2/2

inside of NEST using `sli_run()`. Third, we retrieve the results from the stack via `sli_pop()`. Note that the example above does not contain a call to `sli_push()`, as no arguments are required. The low-level API catches all errors of NEST and raises an appropriate Python exception.

The function `sli_push()` has to convert a given Python object to the corresponding SLI data type. We first determine the type of the Python object and then instantiate a new SLI datum of the right type. `sli_pop()` converts a SLI datum to a Python object. This conversion has a more elegant implementation, because it can exploit the fact that each SLI datum knows its own type. A SLI datum thus can convert itself to a Python object of the right type. To avoid that SLI datums directly depend on Python, we use the acyclic visitor pattern,[9] which moves Python dependent code to a separate class. The details of this technique are explained in Eppler et al. 2008.[10]

We have shown an alternative approach for creating Python bindings for an application by using a generic interpreter-interpreter interaction instead of direct wrapping of the underlying functions and data structures. The implementation of PyNEST is described in detail together with examples and the complete API reference in Eppler et al. 2008.[10] NEST's source code is available under an open-source license for non-commercial use on the homepage of the NEST Initiative at `http://nest-initiative.org`. Currently, we are investigating methods for writing neuron and synapse models in Python instead of C++ in order to ease the development for users not familiar with the internal workings of NEST. In another project, we are improving the scalability of NEST for very large clusters with tens of thousands of processors, e.g. the IBM BlueGene architecture.

The author is grateful for the support of the NEST Initiative, in particular to Markus Diesmann, Marc-Oliver Gewaltig, Moritz Helias, Eilif Muller, and Hans Ekkehard Plesser. Partially funded by DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grant 15879 (FACETS), the Next-Generation Supercomputer Project of MEXT (Japan), and the Helmholtz Alliance on Systems Biology (Germany).

Author Information

Jochen Martin Eppler
Honda Research Institute Europe GmbH
Offenbach am Main, Germany
Bernstein Center for Computational Neuroscience
Freiburg im Breisgau, Germany

J. M. Eppler received his Diploma in Computer Science from the University of Freiburg in 2006. Currently, he is pursuing his PhD in a joint project by the Honda Research Institute Europe and the Bernstein Center for Computational Neuroscience in Freiburg.

References

1. M.-O. Gewaltig and M. Diesmann, *NEST (Neural Simulation Tool)*, **Scholarpedia 2** (4), p. 1430, 2007.
2. H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, *Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers*, in A.-M. Kermarrec, L. Bougé, and T. Priol (eds.), **Euro-Par 2007: Parallel Processing**, pp. 672–681, Springer-Verlag, Berlin, 2007. doi:10.1007/978-3-540-74466-5_71
3. Adobe Systems Inc., **The PostScript Language Reference Manual**, 2 ed., Addison-Wesley, 1991.
4. P. F. Dubois, *Guest Editor's Introduction: Python: Batteries Included*, **Computing in Science and Engineering 9** (3), pp. 7–9, 2007. doi:10.1109/MCSE.2007.51
5. MathWorks, **MATLAB The Language of Technical Computing: Using MATLAB**. Natick, MA, 2002. 3 Apple Hill Drive, Natick, Mass. 01760-2098
6. S. Wolfram, **The Mathematica Book**, 5 ed., Wolfram Media Incorporated, 2003.
7. R. Kötter, J. A. Bednar, A. Davison, M. Diesmann, M.-O. Gewaltig, M. Hines, and E. Muller (eds.), **Frontiers in Neuroinformatics: Special topic on Python in Neuroscience**, Frontiers Research Foundation, 2009. http://frontiersin.org/neuroinformatics/specialtopics/8
8. G. van Rossum, *Python/C API Reference Manual* 2008. http://docs.python.org/api/api.html
9. A. Alexandrescu, **Modern C++ Design**, Addison-Wesley, Boston, 2001.
10. J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M. Gewaltig, *PyNEST: a convenient interface to the NEST simulator*, **Front. Neuroinform. 2**, p. 12, 2008. doi:10.3389/neuro.11.012.2008

A.5 PyNN: A common interface for neuronal network simulators

Andrew Davison[1], Daniel Brüderle[2], Jochen Martin Eppler[3,4], Jens Kremkow[5,6], Eilif Muller[7], Dejan Pecevski[8], Laurent Perrinet[6], and Pierre Yger[1] (2008) PyNN: A common interface for neuronal network simulators. Frontiers in Neuroinformatics 2. doi:10.3389/neuro.11.011.2008.

[1] Unité de Neurosciences Intégratives et Computationnelles
CNRS
1 Avenue de la Terrasse, 91198 Gif-sur-Yvette, France

[2] Kirchhoff-Institut für Physik
Ruprecht-Krals- Universität Heidelberg
Im Neuenheimer Feld 227, 69120 Heidelberg, Germany

[3] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[4] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

[5] Neurobiology and Biophysics
Institute of Biology III Albert-Ludwigs-Universität Freiburg
Schänzlestraße 1, 79104 Freiburg, Germany

[6] Institut de Neurosciences Cognitives de la Méditerranée
CNRS - Aix-Marseille Université II
31 chemin Joseph Aiguier, 13402 Marseille, France

[7] Laboratory of Computational Neuroscience
Ecole Polytechnique Federale de Lausanne
Lusanne, Switzerland

[8] Institute for Theoretical Computer Science
Technische Universität Graz
Inffeldgasse 16b/1, 8010 Graz, Austria

Contributions

- I contributed to the overall design of the PyNN API.
- I assisted Andrew Davison with the planning and implementation of the NEST 2 backend for PyNN.
- I contributed to the writing of the manuscript.
- I assisted in the execution of the example simulations for the manuscript.
- I presented PyNN at the FACETS student course "modeling for beginners" in 2007 (tutor) and at the INCF 2009 congress (demo session).

frontiers in
NEUROINFORMATICS

ORIGINAL RESEARCH ARTICLE
published: 27 January 2009
doi: 10.3389/neuro.11.011.2008

PyNN: a common interface for neuronal network simulators

Andrew P. Davison[1]*, Daniel Brüderle[2], Jochen Eppler[3,4], Jens Kremkow[5,6], Eilif Muller[7], Dejan Pecevski[8], Laurent Perrinet[6] and Pierre Yger[1]

[1] *Unité de Neurosciences Intégratives et Computationelles, CNRS, Gif sur Yvette, France*
[2] *Kirchhoff Institute for Physics, University of Heidelberg, Heidelberg, Germany*
[3] *Honda Research Institute Europe GmbH, Offenbach, Germany*
[4] *Berstein Center for Computational Neuroscience, Albert-Ludwigs-University, Freiburg, Germany*
[5] *Neurobiology and Biophysics, Institute of Biology III, Albert-Ludwigs-University, Freiburg, Germany*
[6] *Institut de Neurosciences Cognitives de la Méditerranée, CNRS, Marseille, France*
[7] *Laboratory of Computational Neuroscience, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*
[8] *Institute for Theoretical Computer Science, Graz University of Technology, Graz, Austria*

Edited by:
Rolf Kötter, Radboud University Nijmegen, The Netherlands

Reviewed by:
Graham Cummins, Montana State University, USA
Fred Howell, Textensor Limited, UK

****Correspondence:***
Andrew Davison, UNIC, Bât. 32/33, CNRS, 1 Avenue de la Terrasse, 91198 Gif sur Yvette, France.
e-mail: andrew.davison@unic.cnrs-gif.fr

Computational neuroscience has produced a diversity of software for simulations of networks of spiking neurons, with both negative and positive consequences. On the one hand, each simulator uses its own programming or configuration language, leading to considerable difficulty in porting models from one simulator to another. This impedes communication between investigators and makes it harder to reproduce and build on the work of others. On the other hand, simulation results can be cross-checked between different simulators, giving greater confidence in their correctness, and each simulator has different optimizations, so the most appropriate simulator can be chosen for a given modelling task. A common programming interface to multiple simulators would reduce or eliminate the problems of simulator diversity while retaining the benefits. PyNN is such an interface, making it possible to write a simulation script once, using the Python programming language, and run it without modification on any supported simulator (currently NEURON, NEST, PCSIM, Brian and the Heidelberg VLSI neuromorphic hardware). PyNN increases the productivity of neuronal network modelling by providing high-level abstraction, by promoting code sharing and reuse, and by providing a foundation for simulator-agnostic analysis, visualization and data-management tools. PyNN increases the reliability of modelling studies by making it much easier to check results on multiple simulators. PyNN is open-source software and is available from http://neuralensemble.org/PyNN.

Keywords: Python, interoperability, large-scale models, simulation, parallel computing, reproducibility, computational neuroscience, translation

INTRODUCTION

Science rests upon the three pillars of open communication, reproducibility of results and building upon what has gone before. In these respects, computational neuroscience ought to be in a good position, since computers by design excel at repeating the same task without variation, as many times as desired: reproducibility of computational results ought, then, to be a trivial task. Similarly, the Internet enables almost instantaneous transmission of research materials, i.e. source code, between labs.

However, in practice this theoretical ease of reproducibility and communication is seldom achieved outside of a single lab and a time frame of a few months or years. While a given scientist may easily be able to reproduce a result obtained a few months ago, precisely reproducing a result obtained several years ago is likely to be rather more difficult, and the general experience seems to be that reproducing the results of others is both difficult and time consuming: very many published papers lack sufficient detail to rebuild a model from scratch, and typographic errors are common.

Having available the source code of the model greatly improves the situation, but here still there are numerous barriers to reproducibility and to building upon previously published models. One is that source code can rapidly go out of date as computer architectures, compiler standards and simulators develop. Another is that model source code is often not written with reuse and extension in mind, and so considerable rewriting to modularize the code is necessary. Probably the most important barrier is that code written for one simulator is not compatible with any other simulator.

Although many computational models in neuroscience are written from the ground up in a general purpose programming language such as C++ or Fortran, probably the majority use a special purpose simulator that allows models to be expressed in terms of neuroscience-specific concepts such as neurons, ion channels, synapses; the simulator takes care of translating these concepts into a system of equations and of numerically solving the equations. A large number of such simulators are available (reviewed in Brette et al., 2007), mostly as open-source software, and each has its own programming language, configuration syntax and/or graphical interface, which creates considerable difficulty in translating models from one simulator to another, or even in understanding someone else's code, with obvious negative consequences for communication between investigators, reproducibility of others' models and building on existing models.

However, the diversity of simulators also has a number of positive consequences: (i) it allows cross-checking – the probability of two

different simulators having the same bugs or hidden assumptions is very small; (ii) each simulator has a different balance between efficiency (how fast the simulations run), flexibility (how easy it is to add new functionality; the range of models that can be simulated), scalability (for parallel, distributed computation on clusters or supercomputers), and ease of use, so the most appropriate can be chosen for a given task.

Addressing the problems associated with an ecosystem of multiple simulators while retaining the benefits would greatly increase the ease of reproducibility of computational models in neuroscience and hence make it easier to verify the validity of published models and to build upon previous work.

There are at least two possible (and complementary) approaches to this. One is to enable direct, efficient communication between different simulators at run-time, allowing different components of a model to be simulated on different simulators (Ekeberg and Djurfeldt, 2008). This approach addresses the problem of building a model from diverse components, but still leaves the problem of having to use different programming languages, and does not enable straightforward cross-checking. The other approach is to develop a system for model specification that is simulator-independent. Translation then only has to be done once for each simulator and not once for each model.

Here we can take advantage of the recent, rapid emergence of the Python programming language as an alternative interface to several of the more widely-used simulators. Thus, for example, both NEURON and NEST may be controlled either via their original, native interpreter (Hoc and SLI, respectively) or via Python. More recent simulators (e.g. PCSIM, Brian) have Python as the only available scripting language. This widespread adoption of Python is probably due to a number of factors, including the powerful data structures, clean and expressive syntax, extensive library, maturity of tools for numerical analysis and visualization (allowing use of a single language for the entire modelling workflow from simulation to analysis to graphing), and the ease-of-use of Python as a glue language which allows computation-intensive code written in a low-level language such as C to be transparently accessed within high-level Python code.

Python alone does not address the translation problem (although it does make the translation process easier, since at least simple data structures such as lists and arrays are the same for each simulator), since neuroscience-specific concepts are still expressed differently. However, it is now possible to define a simulator-independent Python interface for neuronal network simulators and to implement automatic translation to any Python-enabled simulator. We have designed and implemented such an interface, PyNN (pronounced "pine"). In this paper we describe its design, concepts, implementation and use. We do not attempt here to provide a complete user guide – this may be found online at http://neuralensemble.org/PyNN.

DESIGN GOALS

When designing and implementing a common simulator interface, the following goals should be taken into account. These are the goals we have kept in mind when designing and implementing the PyNN interface, but they are equally applicable to any other such interface.

Write the code for a model once, run it on any supported simulator or hardware device *without modification*. This is the primary design goal for PyNN.

Support a high-level of abstraction. For example, it is often preferable to deal with a single object representing a population of neurons than to deal with all the individual neurons directly. Each single neuron can be accessed when necessary, but in many cases the population is the more useful abstraction. The advantages of this approach are that (i) it is easier to maintain a conceptual idea of the model, without being distracted by implementation details, and (ii) the internal implementation of an object can be optimized for speed, parallelization or memory requirements without changing the interface presented to the user.

Support any feature provided by at least two supported simulators. The aim is to strike a balance between supporting all features of all simulators (unfeasible) and supporting only the subset of features common to all simulators (overly restrictive).

Allow mixing of PyNN and native simulator code. PyNN should not limit the range of models that can be implemented. Following the two-simulator rule, above, there will be things that are possible in one simulator and not in any other. Although a model implementation consisting of 100% PyNN is the best scenario for running on multiple simulators, an implementation with 50% PyNN code will be easier to convert between simulators than one with no PyNN code.

Facilitate porting of models between simulators. PyNN changes the process of porting a model between simulators from all-or-nothing, in which the validity of the translated model cannot be tested until the entire translation is complete, to an incremental approach, in which the native code is gradually replaced by simulator-independent code. At each stage, the hybrid code remains runnable, and so it is straightforward to verify that the model behaviour has not been changed.

Minimize dependencies, to make installation as simple as possible and maximize flexibility. There are no visualization and few data analysis tools built-in to PyNN, which means the user can use any such tools they wish.

Present a consistent interface on output as well as on input. The formats used for simulation outputs are consistent across simulator back-ends, making it a stable base upon which to build more complex systems of simulation control, data-analysis and visualization.

Prioritize compatibility over optimizations, but allow compatibility-breaking optimizations to be selected by a deliberate choice of the user (e.g. the `compatible_output` flag of the various `print()` methods is `True` by default, but can be set to `False` to get potentially-faster writing of data to file).

API Versioning. The PyNN API will inevitably evolve over time, as more simulators are supported and to take account of the preferences of the community of users. To ensure backwards compatibility, the API should be versioned so that the user can indicate which version was used for a particular implementation. Note that the examples given in this paper use version 0.4 of the API.

Transparent parallelization. Code that runs on a single processor should run on multiple processors (using MPI) without changes.

Some of these goals are somewhat contradictory: for example, having a high level of abstraction and making porting easy.

Reconciling this particular pair of goals has led to the presence in PyNN of both a high-level, object-oriented interface and a low-level, procedural interface that is more similar to the interface of many existing simulators. These will be discussed further below.

USAGE EXAMPLES

Before describing in detail the concepts underlying the PyNN interface, we will work through some examples of how it is used in practice: first a simple example using the low-level, procedural interface and then a more complex example using the high-level, object-oriented interface.

For the simple example, we will build a network consisting of a single integrate-and-fire (IF) cell receiving spiking input from a Poisson process.

First, we choose which simulator to use by importing the relevant module from PyNN:

```
>>> from pyNN.neuron import *
```

If we wanted to use PCSIM, we would just import `pyNN.pcsim`, etc. Whichever simulator back-end we use, none of the code below would change.

Next we set global parameters of the simulator:

```
>>> setup(timestep=0.1, min_delay=2.0)
```

Now we create two cells: an IF neuron with synapses that respond to a spike with a step increase in synaptic conductance, which then decays exponentially, and a "spike source", a simple cell that emits spikes at predetermined times but cannot receive input spikes.

```
>>> ifcell = create(IF_cond_exp,
...                 {'i_offset': 0.11,
...                  'tau_refrac': 3.0,
...                  'v_thresh' : -51.0})
>>> times = map(float, range(5,105,10))
>>> source = create(SpikeSourceArray,
...                 {'spike_times': times})
```

Behind the scenes, the `create()` function translates the standard PyNN model name, `IF_cond_exp` in this case, into the model name used by the simulator, `Standard_IF` for NEURON, `iaf_cond_exp` for NEST, for example and also translates parameter names and units into simulator-specific names and units. To take one example, the `i_offset` parameter represents the amplitude of a constant current injected into the cell, and is given in nanoamps. The equivalent parameter of the NEST `iaf_cond_exp` model has the name `I_e` and units of picoamps, so PyNN both converts the name and multiplies the numerical value by 1000 when running with NEST. Standard cell models and automatic translation are discussed in more detail in the next section.

The `create()` function returns an `ID` object, which provides access to the parameters of the cell models, e.g.:

```
>>> ifcell.tau_refrac
3.0
>>> ifcell.tau_m = 12.5
>>> ifcell.get_parameters()
{'tau_refrac': 3.0, 'tau_m': 12.5,
 'e_rev_E': 0.0, 'i_offset': 0.11,
 'cm': 1.0, 'e_rev_I': -70.0,
 'v_init': -65.0, 'v_thresh': -51.0,
 'tau_syn_E': 5.0, 'v_rest': -65.0,
 'tau_syn_I': 5.0, 'v_reset': -65.0}
```

Having created the cells, we connect them with the `connect()` function:

```
>>> connect(source, ifcell, weight=0.006,
...         synapse_type='excitatory', delay=2.0)
```

Now we tell the system what variable or variables to record, run the simulation and finish.

```
>>> record_v(ifcell, 'ifcell.dat')
>>> run(200.0)
>>> end()
```

The result of running the above model is shown in **Figure 1**, which also shows the degree of reproducibility obtainable between different simulators for such a simple network.

The low-level, procedural interface, using the `create()`, `connect()` and `record()` functions, is useful for simple models or when porting an existing model written in a different language that uses the create/connect idiom. For larger, more complex networks we have found that an object-oriented approach, with a higher-level of abstraction, is more effective, since it both clarifies the conceptual structure of the model, by hiding implementation details, and allows behind-the-scenes optimizations.

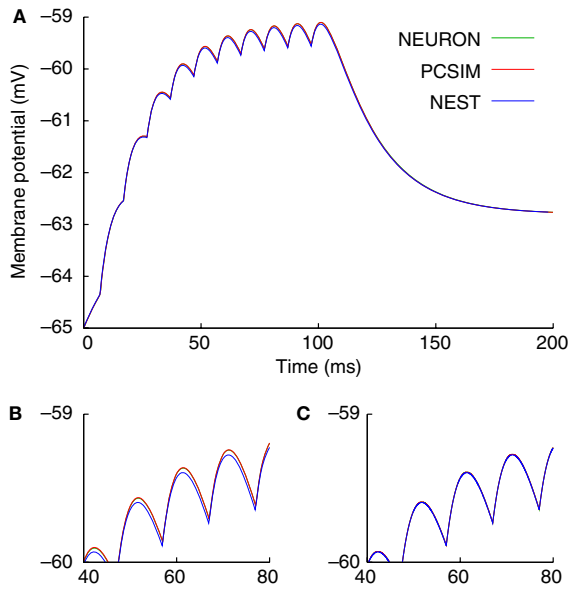


FIGURE 1 | Results of running first example given in the text, with NEURON, NEST and PCSIM as back-end simulators. (A) Entire membrane potential trace with integration time-step 0.1 ms. **(B)** Zoom into a smaller region of the trace, showing small numerical differences between the results of the different simulators. **(C)** Results of a simulation with integration time-step 0.01 ms, showing greatly reduced numerical differences.

To illustrate the high-level, object-oriented interface we turn now from the simple example of a few neurons to a more complex example: a network of several thousand excitatory and inhibitory neurons that displays self-sustained activity (based on the "CUBA" model of Vogels and Abbott (2005), and reproducing the benchmark model used in Brette et al. (2007)). This still is not a particularly complicated network, since it has only two cell types, no spatial structure and no heterogeneity of neuronal or connection properties, but in demonstrating how building such a network becomes trivial using PyNN we hope to convince the reader that building genuinely complex, structured and heterogeneous networks becomes manageable.

Again, we begin by choosing which simulator to use. We also import some classes from PyNN's `random` module.

```
>>> from pyNN.nest2 import *
>>> from pyNN.random import (NumpyRNG,
...                          RandomDistribution)
```

We next specify the parameters of the neuron model (the same model and same parameters are used for both excitatory and inhibitory neurons).

```
>>> cell_params = {
...     'tau_m':     20.0,  'tau_syn_E':   5.0,
...     'cm':         0.2,  'tau_syn_I':  10.0,
...     'v_rest':   -49.0,  'v_reset':   -60.0,
...     'v_thresh': -50.0,  'tau_refrac':  5.0
...     }
```

Parameters with dimensions of voltage are in millivolts, time in milliseconds and capacitance in nanofarads. The units convention is discussed further in the next section.

We now initialize the simulation, this time accepting the default values for the global parameters.

```
>>> setup()
```

Now, rather than creating each cell separately, we just create a `Population` object for each different type of cell:

```
>>> pE = Population(4000, IF_cond_exp,
...                 cell_params,
...                 label="Excitatory")
>>> pI = Population(1000, IF_cond_exp,
...                 cell_params,
...                 label="Inhibitory")
```

By default, all cells of a given Population are created with identical parameters, but these can be changed afterwards. Here we wish to randomize the value of the membrane potential at the start of the simulation to values between −50 and −70 mV.

```
>>> unif_distr = RandomDistribution('uniform',
...                                 [-50,-70])
>>> pE.randomInit(unif_distr)
>>> pI.randomInit(unif_distr)
```

`randomInit()` is a convenience method for randomizing the initial membrane potential. For the more general case of randomizing any cell parameter use `rset()`.

Just as individual neurons are encapsulated within `Populations`, connections between neurons are encapsulated within `Projections`. To create a `Projection` object, we need to specify how the neurons will be connected, either via an algorithm or via an explicit list. Different algorithms are encapsulated in different `Connector` classes, e.g. `FixedProbabilityConnector`, `AllToAllConnector`. An explicit list of connections can be provided via a `FromListConnector` or a `FromFileConnector`.

```
>>> FPC = FixedProbabilityConnector
>>> exc_conn = FPC(0.02, weights=0.004,
...                delays=0.1)
>>> inh_conn = FPC(0.02, weights=0.051,
...                delays=0.1)
```

Note that weights are in microsiemens and delays in milliseconds. Where the delay is not specified, the global minimum delay specified in the `setup()` function is used. Here we set all weights and delays of a `Projection` to the same value, but it is equally possible to pass the constructor a `RandomDistribution` object, as we did above for the initial membrane potential, or an explicit list of values.

To create a `Projection`, we need to specify the pre- and post-synaptic `Populations`, a `Connector` object, and a synapse type. The standard IF cells each have two synapse types, "`excitatory`" and "`inhibitory`". User-defined models can use arbitrary names, e.g. "`AMPA`", "`NMDA`".

```
>>> e2e = Projection(pE, pE, exc_conn,
...                  target='excitatory')
>>> e2i = Projection(pE, pI, exc_conn,
...                  target='excitatory')
>>> i2e = Projection(pI, pE, inh_conn,
...                  target='inhibitory')
>>> i2i = Projection(pI, pI, inh_conn,
...                  target='inhibitory')
```

Having constructed the network, we now need to instrument it, using the `record()` (for recording spikes) and `record_v()` (membrane potential) methods of the `Population` objects. Here we choose to record spikes from 1000 of the excitatory neurons (chosen at random) and all of the inhibitory neurons, and to record the membrane potential of two specific excitatory neurons. We then run the simulation for 1000 ms.

```
>>> pE.record(1000)
>>> pI.record()
>>> pE.record_v([pE[0], pE[1]])
>>> run(1000.0)
```

After running the simulation, we can access the results or write them to file.

```
>>> pI.getSpikes()[:5]
array([[ 715. ,     1.5],
       [ 609. ,     1.6],
       [ 708. ,     1.7],
       [ 796. ,     1.7],
       [  34. ,     1.8]])
```

```
>>> pE.get_v()[:5]
array([[  0.   ,     0.1  ,  -55.073],
       [  1.   ,     0.1  ,  -50.163],
       [  0.   ,     0.2  ,  -55.098],
       [  1.   ,     0.2  ,  -50.212],
       [  0.   ,     0.3  ,  -55.122]])
>>> end()
```

The results of running simulations of the above network with two different simulator back-ends are shown in **Figure 2**.

PRINCIPAL CONCEPTS

To achieve the goal of "*write the code for a model once, run it on any supported simulator without modification*" requires (i) a common interface, (ii) neuron and synapse models that are standardized across simulators, (iii) consistent handling of physical units, (iv) consistent handling of (pseudo-)random numbers. To achieve the twin goals of supporting a high-level of abstraction and facilitating porting of models between simulators requires both an object-oriented and a procedural interface. The implementation of all these requirements is described in more depth in the following. We also illustrate the mixing of PyNN and native simulator code, and how PyNN can support features that are found in only a single simulator back-end, by describing support for multi-compartmental models.

STANDARD CELL MODELS

A fundamental concept in PyNN is the cell type – a given model of a neuron, representable by a set of equations, and comprising sub-threshold behaviour, spiking mechanism and post-synaptic response. The public interface of a cell type is mainly defined by its parameters. Different neurons of the same cell type may have very different behaviour if they have different values of the parameters. For example, the Izhikevich model (Izhikevich, 2003), can reproduce a wide range of spiking patterns, from fast-spiking through regular spiking to multiple types of bursting, depending on the

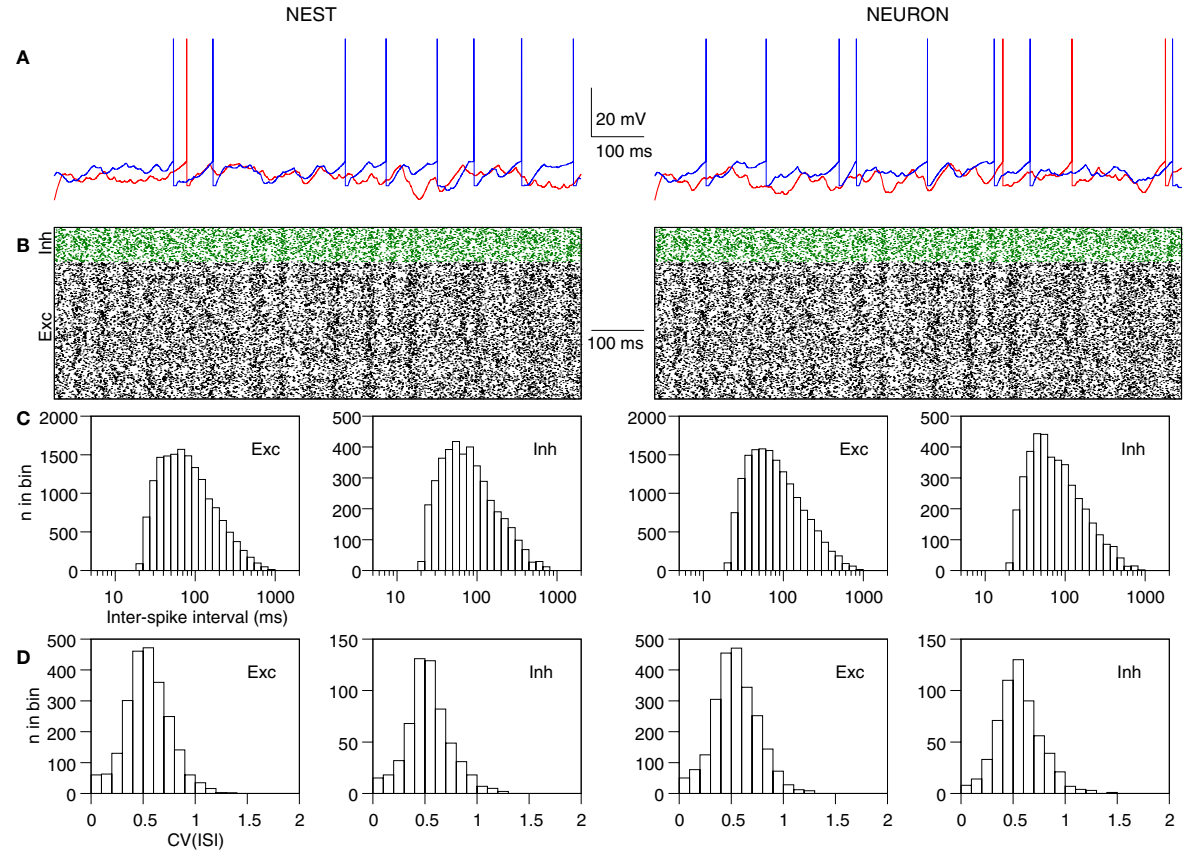


FIGURE 2 | Results of running the second example given in the text, with NEURON and NEST as back-end simulators. Note that the network connectivity and initial conditions were identical in the two cases. **(A)** Membrane potential traces for two excitatory neurons. Note that the NEST and NEURON traces are very similar for the first 50 ms, but after that diverge rapidly due to the effects of network activity, which amplifies the small numerical integration differences. **(B)** Spiking activity of excitatory (black) and inhibitory (green) neurons. Each dot represents a spike and each row of dots a different neuron. All 5000 neurons are shown. **(C)** Distribution of pooled inter-spike intervals (ISIs) for excitatory and inhibitory neurons. **(D)** Distribution over neurons of the coefficient of variation of the ISI [CV(ISI)].

parameter values chosen. A cell type is therefore a model type rather than a biologically defined cell type (such as "Layer V pyramidal neuron", for example).

When using a given simulator back-end, PyNN can work with any cell type that is supported by that simulator. In this case, the cell type is generally represented by a string, holding a model name that is meaningful for that simulator, e.g. "`iaf_neuron`" in NEST.

Of course, such a cell type will only work with one simulator. To create a model that will run on different simulators requires you to use one of PyNN's built-in, standard cell models, each represented by a sub-class of the `StandardCell` class. The models provided by PyNN include various simple IF models, the Izhikevich-like adaptive exponential IF model (Brette and Gerstner, 2005), a single-compartment neuron with Hodgkin–Huxley sodium and potassium channels, and various models that emit spikes (e.g. according to a Poisson process) but cannot receive them.

The `StandardCell` class contains machinery for translating model names, parameter names and parameter units between PyNN standardized values and simulator-specific values. This is particularly useful when the underlying simulators use different unit systems or different parameterizations of the same set of equations, e.g. when one simulator expects the membrane time constant and another the membrane leak conductance. An example of the translations performed by PyNN is given in **Table 1**.

Currently, all the standard cell types are single-compartment or point neuron models, since PyNN currently supports only one simulator for multi-compartmental models (NEURON). Further details on using multi-compartmental models with PyNN's NEURON back-end are given below. We plan in future to allow specifying multi-compartmental cell types using a NeuroML description (Crook et al., 2005).

UNITS

As is clear from the previous section, each simulator back-end has its own convention for which units to use for which physical quantities. The exception to this is Brian, which has a system for explicitly specifying units and for checking that equations are dimensionally consistent. In the future, we plan to adopt Brian's system for PyNN, but for now we have chosen to use a convention, which is similar to that of NEURON and NEST in that the units are those that tend to be used by experimental physiologists. An alternative would have been the convention used by PCSIM (and also by the GENESIS simulator) of using pure SI units with no prefixes. The advantage of the latter convention is that there is no need for checking equations for dimensional consistency. The disadvantage is that numerical values in such a system are often very large or very small, and hence the human intuition for reasonable and unreasonable parameter values is mostly lost.

Irrespective of the relative merits of different conventions, the most important thing is that PyNN now provides a single convention which is valid across simulators. In detail, the convention is as follows: voltage – mV, current – nA, conductance – μS, time – ms, capacitance – nF.

STANDARD SYNAPSE MODELS

In PyNN, the shape and time-course of the elementary post-synaptic current or conductance change in response to a pre-synaptic spike are considered to be a part of the post-synaptic neuron model, while all other properties of a synaptic connection, notably its weight (the peak current or conductance of the synaptic response), delay (for point models, this implicitly includes axonal propagation, chemical transmission and dendritic propagation; more morphologically and/or biophysically detailed models may model explicitly some or all of these sources of delay), and short- and long-term plasticity, are considered to depend on both pre- and post-synaptic neurons, and so are encapsulated in the concept of "synapse type" that mirrors the "cell type" discussed above.

The default type of synaptic connection in PyNN is static, with fixed synaptic weights. To model dynamic synapses, for which the synaptic weight (and possibly other properties, such as rise-time) varies depending on the recent history of post- and/or pre-synaptic activity, we use the same idea as for neurons, of standardized, named models that have the same interface and behaviour across simulators, even if the underlying implementation may be very different.

Where the approach for dynamic synapses differs from that for neurons is that we attempt a greater degree of compositionality, i.e. we decompose models into a number of components, for

Table 1 | Comparison of parameter names and units for different implementations of a leaky integrate-and-fire model with a fixed firing threshold and current-based, alpha-function synapses. This model is called `IF_curr_alpha` in PyNN, `iaf_psc_alpha` in NEST, `LIFCurrAlphaNeuron` in PCSIM and `StandardIF` in NEURON (this is a model template distributed with PyNN and is not in the standard NEURON distribution). Manual conversion of names and units is straightforward but error-prone and time-consuming. PyNN takes care of such conversions transparently.

Parameter	PyNN		NEST		NEURON		PCSIM	
Resting membrane potential	`v_rest`	mV	`E_L`	mV	`v_rest`	mV	`Vresting`	V
Reset membrane potential	`v_reset`	mV	`V_reset`	mV	`v_reset`	mV	`Vreset`	V
Membrane capacitance	`cm`	nF	`C_m`	pF	`CM`	nF	`Cm`	F
Membrane time constant	`tau_m`	ms	`tau_m`	ms	`tau_m`	ms	`taum`	s
Refractory period	`tau_refrac`	ms	`t_ref`	ms	`t_refrac`	ms	`Trefrac`	s
Excitatory synaptic time constant	`tau_syn_E`	ms	`tau_syn_ex`	ms	`tau_e`	ms	`TauSynExc`	s
Inhibitory synaptic time constant	`tau_syn_I`	ms	`tau_syn_in`	ms	`tau_i`	ms	`TauSynInh`	s
Spike threshold	`v_thresh`	mV	`V_th`	mV	`v_thresh`	mV	`Vthresh`	V
Injected current amplitude	`i_offset`	nA	`I_e`	pA	`i_offset`	nA	`Iinject`	A

example for short-term and long-term dynamics, or for the timing-dependence and the weight-dependence of STDP rules, that can then be composed in different ways.

The advantage of this is that if we have n different models for component A and m models for component B, then we require only $n + m$ models rather than $n \times m$, which had advantages in terms of code-simplicity and in shorter model names. The disadvantage is that not all combinations may exist, if the underlying simulator implements composite models rather than using components itself: in this situation, PyNN checks whether a given composite model AB exists for a given simulator and raises an Exception if it does not. The composite approach may be extended to neuron models in future versions of the PyNN interface depending on the experience with composite synapse models.

Currently only a single model exists in PyNN for the short-term plasticity component, the Tsodyks–Markram model (Markram et al., 1998). For long-term plasticity there is a spike-timing-dependent plasticity STDP component, which itself is composed of separate timing-dependence and weight-dependence components.

LOW-LEVEL, PROCEDURAL INTERFACE

We refer to the procedural interface as "low-level" because it deals with a lower level of abstraction – individual neurons and individual synapses – than the object-oriented interface. The procedural interface consists of the functions `create()`, `connect()`, `set()`, `record()` (for recording spikes) and `record_v()` (for recording membrane potential). Each of these functions operates on, or returns, either individual cell `ID` objects or lists of such objects. As was described in the Usage Examples section, as well as being passed around as arguments, the `ID` object may be used for accessing/modifying the parameters of individual neurons, and takes care of parameter translation using the `StandardCell` mechanisms described above.

It is possible to some extent to mix the low-level and high-level interfaces. For example, it is possible to access individual neurons within a `Population` as `ID` objects and then use the `connect()` function to connect them, instead of using a `Projection` object.

Why have both a low-level and high-level interface? Having both is a potential source of confusion for users and is definitely a maintenance burden for developers. The main reason is to support the use of PyNN as a porting tool. The majority of neuronal network models using existing simulators use a procedural approach, and so conversion to PyNN is easier if PyNN supports the same approach. In addition, when developing a PyNN interface for a simulator, or for neuromorphic hardware, that deals primarily with individual cells and synaptic connections, it is easier to implement only the low-level interface, since the high-level interface can be built upon it.

HIGH-LEVEL, OBJECT-ORIENTED INTERFACE

Object-oriented programming has been used for many years in computer science as a method for reducing program complexity. As the ambition and scope of large-scale, biologically detailed neuronal network modelling increases, reducing program complexity will become more and more critical, as the limiting factor in computational neuroscience becomes the productivity of the programmer and not the capacity of the computer (Wilson, 2006). It is for this reason that the preferred interface in PyNN for developing new models is an object-oriented one.

The object-oriented interface is built around three main classes:

Population – a group of cells all with the same cell type (model type). It is generally considered that the cells in a `Population` should all represent the same biological cell type, i.e. although parameter values may vary between cells in the group, all cells should have qualitatively the same firing response. This is not enforced, but is a good guideline to follow for producing understandable code. The `Population` class eliminates tedious iteration over lists of neurons and enables more efficient, array-based management of neuron properties.

Projection – the set of connections of a given synapse type between two `Populations`. Creating a Projection requires specifying the pre- and post-synaptic `Populations`, the synapse type, and the algorithm used to determine which neurons connect to which.

Connector – an encapsulation of the connection algorithm used in creating a Projection. Simple examples of such algorithms are "all-to-all", "one-to-one" and "connect-each-pre-and-post-synaptic-cell-with-a-fixed-probability". It is also possible to provide an explicit list of which cells are to be connected to which others. Each algorithm is defined within a subclass of the `Connector` class. PyNN contains a number of such classes, but it is fairly straightforward for a user to define their own algorithms.

In future development of PyNN, we plan to extend the interface to still higher-level abstractions, such as layers, cortical columns, brain areas and inter-areal projections. We also aim to use the high-level interface as a link between spiking network models and more abstract models that do not represent individual neurons, such as mean-field models.

RANDOM NUMBERS

The central nervous system contains many sources of noise, and activity patterns are often sufficiently complex, and possibly chaotic, to make a stochastic representation a reasonable model.

This can become a problem when comparing the behaviour of a given model run on different simulators, since random differences might obscure real inconsistencies between implementations of the model. Similarly, when performing distributed computations on parallel machines, the model behaviour should not depend on the number of processors used (Morrison et al., 2005), and random differences can conceal real differences between the parallel and serial implementations.

For these reasons, it is important to be able to use identical sequences of random numbers in different simulators, and to have the random number used at a particular point in the program execution be independent of which processor it is running on.

Another consideration is that simulations in most cases use only pseudo-random sequences, and low-quality random number generators (RNGs) may have correlations between different elements of the sequence that can significantly affect the qualitative behaviour of a network. Hence it is necessary to be able to test the simulation with different RNGs.

PyNN supports simulator-independent RNGs and use of different generators – currently any of the generators provided by

the numpy package or by the GNU Scientific Library (GSL) can be used.

This is done by wrapping the numpy and GSL RNGs in classes with a common interface. PyNN's random module contains the classes NumpyRNG and GSLRNG, which both have a single method, next(n, distribution, parameters), which returns n random numbers from a distribution of type distribution with parameters parameters, e.g.

```
>>> from pyNN.random import NumpyRNG, GSLRNG
>>> rngN = NumpyRNG(seed=76847376)
>>> rngG = GSLRNG(seed=87548753)
>>> rngN.next()
0.91457981651574294
>>> rngG.next(5)
array([ 0.02518011, 0.79118205, 0.16679516,
...       0.1902914, 0.66204769])
>>> rngN.next(3, 'gamma', [2.0, 0.5])
array([ 0.48903019, 0.63129009, 0.70428452])
>>> rngG.next(distribution='uniform')
0.93618978746235371
```

Since all PyNN code that uses random numbers accesses the RNG classes only through this next() method, a user can substitute their own RNG simply by defining a wrapper class with such a method.

Since very often one wishes to use the same random distribution repeatedly, rather than changing distribution each time, the random module also provides the RandomDistribution class, which is initialized with the distribution name and parameters, and thereafter the next() method is simplified to take a single argument, the number of values to draw from the distribution, e.g.

```
>>> from pyNN.random import (NumpyRNG,
...                          RandomDistribution)
>>> rng = NumpyRNG(seed=8745753)
>>> gamma_distr = RandomDistribution('gamma',
...                                  [2.0, 0.5],
...                                  rng=rng)
>>> gamma_distr.next(3)
array([ 0.72682412, 0.82490159, 1.03882654])
```

Note that NumpyRNG and GSLRNG distributions may not have the same names, e.g. "normal" for NumpyRNG and "gaussian" for GSLRNG, and the arguments may also differ. One of our future plans is to extend the random module in order to harmonize names across RNGs.

MULTI-COMPARTMENTAL MODELS

PyNN currently supports only a single simulator, NEURON, that is suitable for many-compartment models. Given the principle of supporting simulator-independence only for features that are shared by at least two of the supported simulators, and given PyNN's focus on network modelling, PyNN does not provide an API for specifying simulator-independent multi-compartmental models. This is a possible future development – preliminary work has been done on a PyNN interface to the MOOSE simulator (Ray and Bhalla, 2008) – but a more likely path would be to make use of the NeuroML standards for specifying multi-compartmental models. In this scenario, the filename of a NeuroML level 2 file, specifying a single cell type, would be passed as the cellclass argument to the PyNN create() function or Population constructor.

However, since native and PyNN code can be mixed, the pyNN.neuron module already supports simulations with multi-compartmental models. The pre-synaptic compartment whose voltage is watched to trigger synaptic transmission (e.g. axon terminal) can be specified using the source argument to the Projection constructor, and the post-synaptic mechanism specified with the target argument.

DEBUGGING

Should an error occur in a PyNN simulation, a good first step is to re-run it on another simulator back-end and so narrow down the source of the problem to one back-end in particular. Nevertheless, it has proven to be the case that the additional layers of abstraction provided by PyNN sometimes make it harder to track down sources of errors. To counterbalance this, PyNN traps errors coming from the simulator core and employs Python's introspection capabilities to provide additional information about the error context. For example, if an invalid parameter name is provided to a neuron model, the error message lists all the valid parameter names for that model. Furthermore, logging can be switched on via the init_logging() function in the pyNN.utility module, causing detailed information about what the system is doing to be written to file, a valuable resource for tracking down bugs.

IMPLEMENTATION

PyNN is both a definition of a common simulator interface and an implementation of this interface for each supported simulator. PyNN is implemented as a Python package containing a common module, which defines the API and contains functionality common to all simulator back-ends, a random module (described above), and a module for each simulator back-end, as shown in **Figure 3**. Each simulator module separately implements the API, although it can make use of much shared code in common. In most cases, the simulator modules have been implemented by, or in close collaboration with, the simulator developers.

PyNN currently fully supports the following simulators: NEURON (Carnevale and Hines, 2006; Hines and Carnevale, 1997; Hines et al., 2008), NEST (Eppler et al., 2008; Gewaltig and Diesmann, 2007), PCSIM (http://www.lsm.tugraz.at/pcsim/) and Brian (Goodman and Brette, 2008). Support for MOOSE (Ray and Bhalla, 2008) and for export in NeuroML format (Crook et al., 2005) is under development.

PyNN also supports the Heidelberg neuromorphic hardware system (Schemmel et al., 2007). This illustrates a major benefit of the existence of a common neuronal simulation interface: novel simulation or emulation systems do not need to develop their own programming interface, but can benefit from an existing one that guarantees interoperability with existing tools. Using PyNN as the interface to neuromorphic hardware systems provides the possibility of closing the gap between the two domains of numerical simulation and physical emulation, which have so far coexisted rather separately.

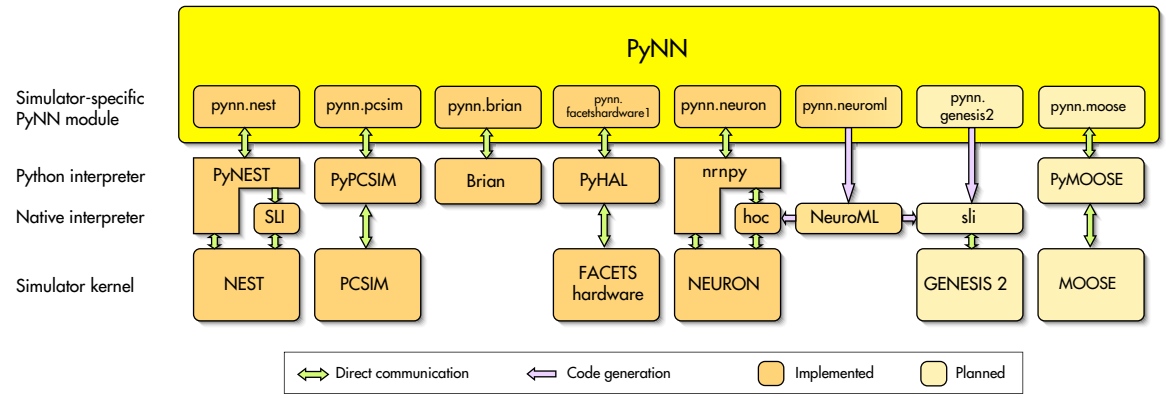


FIGURE 3 | The architecture of PyNN.

LIMITATIONS ON REPRODUCIBILITY

For a given model with a given parameter set run on a given version of a given simulator, it should be possible to exactly reproduce a simulation result, independent of computer architecture (except where this affects the precision of the floating-point representation) or operating system. For parallel systems, results should also be independent of how many threads or processes are used in the computation, although here exact quantitative reproduction is harder to achieve. Reproducibility across different versions of a given simulator is not essential provided the precise version used to generate a given result is specified, but it is of course highly desirable. When running a model on different simulators, exact reproduction is impossible to achieve, except in simple cases, due to round-off errors in floating point calculations. When validating a model implementation by running it on two or more simulators, therefore, what level of reproducibility is achievable, and how can we tell whether any differences are due to round-off error or to implementation errors?

To get a preliminary handle on this problem, we have compared the difference in model activity between two simulators to the difference due to two different initial conditions with the same simulator.

Our test case is the balanced random network, based on Vogels and Abbott (2005), whose implementation was shown above. The activity pattern of this network is very sensitive to initial conditions (chaotic or near-chaotic), and so we cannot use differences in the precise spike pattern to measure reproducibility: we are more interested in the statistical properties of the activity, and so we have chosen to take the distribution of inter-spike intervals (ISIs) of excitatory neurons (see **Figure 2C**) as a measure of network activity.

To measure the difference between the distributions from two different runs we use the Kolmogorov–Smirnov two-sample test. We ran the simulation ten times, each time with a different seed for the RNG used to generate the initial membrane potential distribution, with both NEURON and NEST back-ends. This gave values for the Kolmogorov–Smirnov D-statistic between 0.008 and 0.026 ($n \simeq 19000$) with a mean of 0.015, with associated p-values (probability that the two distributions are the same) between 6.3×10^{-5} and 0.68 with mean 0.15.

We then ran the simulation twenty times just on NEURON, each time with a different RNG seed, to give 10 pairs of distributions. In this case the D-values were in the range 0.007–0.026, mean 0.015, and the p-values in the range 2.8×10^{-5} to 0.77, mean 0.20.

In summary, the differences due to different simulators are in almost exactly the same range as those due to different initial conditions, suggesting that the differences between the simulators are indeed due to round-off errors and that there are not, therefore, any implementation errors in this case.

It is also interesting to note that in most cases the null hypothesis is supported, i.e. the distributions are the same, but that for some initial conditions there are highly significant differences between the ISI distributions. The ISI distribution may not therefore be the best measure for reproducibility in this case.

DISCUSSION

In this article we have presented PyNN, a Python-based common simulator interface, which allows simulator-independent model specification. PyNN is already in use in a number of research groups, and has been a key technology enabling improved communication between labs in a pan-European collaborative project with a major component of modelling and of neuromorphic hardware development (the FACETS project: http://www.facets-project.org).

By providing a standard simulation platform, PyNN also has the potential to act as the foundation for other, simulator agnostic but neuroscience-specific, tools such as analysis, visualization and data-management software.

PyNN is not the only project to address simulator-independent model specification and simulator interoperability (review in Cannon et al., 2007). neuroConstruct (Gleeson et al., 2007) is a tool to develop networks of morphologically-detailed neurons using a graphical user interface (GUI), that can generate code for both the NEURON and GENESIS simulators. A limitation with respect to PyNN is that since it uses code generation rather than a direct interface, neuroConstruct cannot receive information back from the simulator except by reading the data files it

generates. A second limitation is that features that are not available through the GUI cannot be incorporated in a model. The NeuroML standards (Crook et al., 2005, http://www.neuroml.org) are intended to provide an infrastructure for exchanging model specifications between groups in a simulator-independent way. Their scope includes much more detailed levels of modelling, e.g. membrane ion channels and detailed dendritic morphology, than are supported by PyNN. They have the advantage over PyNN of being language-independent, since specifications are written in XML, for which tools exist in all major programming languages. The major disadvantage of purely declarative specifications is lack of flexibility: if a concept or entity is not defined in the standard, it is not possible to specify models that use it, whereas with a procedural/imperative or mixed declarative-procedural specification such as is achievable with PyNN, arbitrary specifications are possible.

Although we emphasize here the differences between the GUI, pure-declarative, and programming-interface approaches to simulator-independent model specification, in fact they are highly complementary. Graphical interfaces are particularly good for beginners, for teaching, for giving high-level overviews of a system, and for integrating analysis and visualization tools. It would be very useful for neuroConstruct to be able to generate PyNN code, for example, in addition to code for NEURON and GENESIS. Declarative specifications reach the highest levels of system-independence, for the range of concepts that are supported. They are also particularly suitable for transformation into human-readable formats and for automated GUI generation. As such, they seem to be best suited for domains in which the modelling approach is fairly stable, e.g. for describing neuron morphologies or non-stochastic ion channel models. In PyNN, we plan to support simulator-independent multi-compartmental models using NeuroML: in this scenario cell models would be specified in NeuroML while PyNN would be used for network specification and for simulation setup and control.

Our main priorities for future development of PyNN are to increase the number of supported simulators (simulator developers who are interested in PyNN support for their simulator are encouraged to contact us), improve the support for multi-compartmental modelling, and extend the interface towards higher-level abstractions, such as cortical columns and more abstract modelling approaches. PyNN is open source software (CeCILL licence, http://www.cecill.info) and has an open development model: anyone who wishes to contribute is welcome and invited to do so.

ACKNOWLEDGEMENTS

This work was supported by the European Union (FACETS project, FP6-2004-IST-FETPI-015879). Jens Kremkow is also supported by the German Federal Ministry of Education and Research (BMBF grant 01GQ0420 to BCCN, Freiburg).

REFERENCES

Brette, R., and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.* 94, 3637–3642.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J., Diesmann, M., Morrison, A., Goodman P. H., Harris, F. Jr, Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A., El Boustani, S., and Destexhe, A. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *J Comput. Neurosci.* 23, 349–398.

Cannon, R., Gewaltig, M., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., and De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics* 5, 127–138.

Carnevale, N. T., and Hines, M. L. (2006). The NEURON Book. Cambridge, University Press.

Crook, S., Beeman, D., Gleeson, P., and Howell, F. (2005). XML for model specification in neuroscience: an introduction and workshop summary. *Brains Minds Media* 1, bmm228 (urn: nbn:de:0009–3–2282).

Ekeberg, Ö., and Djurfeldt, M. (2008). MUSIC – multisimulation coordinator: request for comments. *Nat. Precedings* http://dx.doi.org/10.1038/npre.2008.1830.1.

Eppler, J., Helias, M., Diesmann, M., and Gewaltig, M.-O. (2008). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform.* 2. doi: 10.3389/neuron.11.012.2008.

Gewaltig, M.-O., and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430.

Gleeson, P., Steuber, V., and Silver, R. A. (2007). neuroConstruct: a tool for modeling networks of neurons in 3D space. *Neuron* 54, 219–235.

Goodman, D., and Brette, R. (2008). Brian: a simulator for spiking neural networks in Python. *Front. Neuroinform.* 2. doi: 10.3389/neuron.11.005.2008.

Hines, M. L., and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput.* 9, 1179–1209.

Hines, M., Davison, A., and Muller, E. (2008). NEURON and Python. *Front. Neuroinform.* 2. doi: 10.3389/neuron.11.001.2009.

Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Trans Neural Netw.* 14, 1569–1572.

Markram, H., Wang, Y., and Tsodyks, M. (1998). Differential signaling via the same axon of neocortical pyramidal neurons. *Proc. Natl. Acad. Sci. USA* 95, 5323–5328.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural Comput.* 17, 1776–1801.

Ray, S., and Bhalla, U. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Front. Neuroinform.* 2. doi: 10.3389/neuron.11.006.2008.

Schemmel, J., Brüderle, D., Meier, K., and Ostendorf, B. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07). New Orleans, IEEE Press, pp. 3367–3370. doi: 10.1109/ISCAS.2007.378289.

Vogels, T., and Abbott, L. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci.* 25, 10786–10795.

Wilson, G. (2006). Where's the real bottleneck in scientific computing? *Am. Sci.* 94, 5–6.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 21 September 2008; paper pending published: 21 October 2008; accepted: 22 December 2008; published online: 27 January 2009.

Citation: Davison AP, Brüderle D, Eppler J, Kremkow J, Muller E, Pecevski D, Perrinet L and Yger P (2009) PyNN: a common interface for neuronal network simulators. Front. Neuroinform. (2009) 2:11. doi: 10.3389/neuro.11.011.2008

A.6 Run-time interoperability between neuronal network simulators based on the MUSIC framework

Mikael Djurfeldt[1], Johannes Hjorth[1], Jochen Martin Eppler[2,3], Niraj Dudani[4], Moritz Helias[3], Tobias Potjans[5,6], Upinder S. Bhalla[4], Markus Diesmann[3,6,7], Jeanette Hellgren Kotaleski[1], and Örjan Ekeberg[1] (2009) Run-Time Interoperability between Neuronal Network Simulators based on the MUSIC Framework. Neuroinformatics (2010) 8:43–60. doi:10.1007/s12021-010-9064-z.

[1] School of Computer Science and Communication
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

[2] Honda Research Institute Europe GmbH
Carl-Legien-Straße 30, 63073 Offenbach, Germany

[3] Bernstein Center for Computational Neuroscience
Albert-Ludwigs-Universität Freiburg
Hansastraße 9A, 79104 Freiburg, Germany

[4] National Centre for Biological Sciences
Bangalore, India

[5] Institute of Neurosciences and Medicine
Research Center Jülich
52425 Jülich, Germany

[6] Computational Neuroscience Group
RIKEN Brain Science Institute
Wako-shi, Saitama, Japan

[7] Brain and Neural Systems Team
Computational Science Research Program
RIKEN Brain Science Institute
Wako City, Saitama, Japan

Contributions

- I designed the data structures and algorithms to support incoming and outgoing MUSIC connections and for the handling of MUSIC ports and channels.
- I conceived an efficient buffering strategy for incoming events.
- I designed and implemented exception and error handling for the interface.
- I coordinated the benchmarks that were executed by Tobias Potjans and Mikael Djurfeldt.
- I wrote the NEST chapter of the article and created all figures.
- I proposed a consistent structure for the simulator sections with respect to the description of algorithms and data structures. I also supervised the realization of my proposal.
- Together with Mikael Djurfeldt, I wrote the introduction of the article and created figure 1.

Neuroinform (2010) 8:43–60
DOI 10.1007/s12021-010-9064-z

Run-Time Interoperability Between Neuronal Network Simulators Based on the MUSIC Framework

Mikael Djurfeldt · Johannes Hjorth · Jochen M. Eppler · Niraj Dudani · Moritz Helias · Tobias C. Potjans · Upinder S. Bhalla · Markus Diesmann · Jeanette Hellgren Kotaleski · Örjan Ekeberg

Published online: 2 March 2010

Abstract MUSIC is a standard API allowing large scale neuron simulators to exchange data within a parallel computer during runtime. A pilot implementation of this API has been released as open source. We provide experiences from the implementation of MUSIC interfaces for two neuronal network simulators of different kinds, NEST and MOOSE. A multi-simulation of a cortico-striatal network model involving both simulators is performed, demonstrating how MUSIC can promote inter-operability between models written for different simulators and how these can be re-used to build a larger model system. Benchmarks show that the MUSIC pilot implementation provides efficient data transfer in a cluster computer with good scaling. We conclude that MUSIC fulfills the design goal that it should be simple to adapt existing simulators to use MUSIC. In addition, since the MUSIC API enforces independence of the applications, the multi-simulation could be built from pluggable component modules without adaptation of the components to each other in terms of simulation time-step or topology of connections between the modules.

Keywords MUSIC · Large-scale simulation · Computer simulation · Computational neuroscience · Neuronal network models · Inter-operability · MPI · Parallel processing

Introduction

Large scale neuronal network models and simulations have become important tools in the study of the brain and the mind (Albus et al. 2007; Djurfeldt et al. 2008a). Such models work as platforms for integrating knowledge from many sources of data. They help to elucidate how information processing occurs in the healthy

Electronic Supplementary Material The online version of this article (doi:10.1007/s12021-010-9064-z) contains supplementary material, which is available to authorized users.

Johannes Hjorth and Jochen M. Eppler have contributed equally to the contents of the article.

M. Djurfeldt (✉) · J. Hjorth · J. Hellgren Kotaleski · Ö. Ekeberg
School of Computer Science and Communication, Royal Institute of Technology, 100 44 Stockholm, Sweden
e-mail: mikael@djurfeldt.com

M. Djurfeldt · M. Diesmann
RIKEN Brain Science Institute, Wako-shi, 351-0198 Saitama, Japan

J. M. Eppler
Honda Research Institute Europe GmbH, Carl-Legien-Straße 30, 63073 Offenbach, Germany

J. M. Eppler · M. Helias · M. Diesmann
Bernstein Center for Computational Neuroscience, Albert-Ludwigs-Universität Freiburg, Hansastraße 9A, 79104 Freiburg, Germany

N. Dudani · U. S. Bhalla
National Centre for Biological Sciences, Bangalore, India

T. C. Potjans
Institute of Neurosciences and Medicine, Research Center Jülich, 52425 Jülich, Germany

T. C. Potjans · M. Diesmann
RIKEN Computational Science Research Program, Wako-shi, 351-0198 Saitama, Japan

brain, while perturbations to the models can provide insights into the mechanistic causes of diseases such as Parkinson's disease, drug addiction and epilepsy. A better understanding of neuronal processing may also contribute to computer science and engineering by suggesting novel algorithms and architectures for fault tolerant and energy efficient computing (see, e.g., Schemmel et al. 2008). Simulations of increasingly larger network models are rapidly developing. In principle, we have, already today, the computational capability to simulate significant fractions of the mammalian cortex (Djurfeldt et al. 2008b). A great deal of effort has been put into the development of simulation software suites (see, e.g., Brette et al. 2007). Different software packages, such as NEURON (Carnevale and Hines 2006), GENESIS (Bower and Beeman 1998) and NEST (Gewaltig and Diesmann 2007) have been developed for simulations of neuron and network models.

Depending on the scientific question asked, or on the tradition in respective computational neuroscience lab, models of various parts of the brain have been formulated using different simulators. The positive side of this diversity is that it provides a repertoire of tools where different simulators have different strengths (see e.g. Brette et al. 2007, for a review of software for spiking neuron simulations). Diversity is also good for the strong ongoing development of simulation technology. On the negative side, the reuse of models is hampered by the lack of interoperability due to the multitude of languages and simulators used. Also, reimplementation of one model in other software is in practice both time consuming and error prone (personal experience, see also Cannon et al. 2007).

Interoperability can be facilitated in several ways. One approach is to provide a model specification in some standardized format which can be understood by many simulation tools. Two developments in this direction are PyNN and NeuroML. PyNN (Davison et al. 2009) is a common programming interface enabling model scripting in the Python programming language. PyNN already supports several simulators. The computational neuroscience community has built a growing toolbox around this environment for simulation and analysis of data. NeuroML is an XML-based standard for the description of model components at various levels of the nervous system which also allows models to be described in a simulator-independent way (Crook and Howell 2007; Crook et al. 2007). Another approach, *run-time interoperability* (Cannon et al. 2007), is to allow different simulation tools to communicate data online. MUSIC (Ekeberg and Djurfeldt 2008, 2009) is a standard interface for on-line communication between simulation tools. The MUSIC project was initiated by the INCF (International Neuroinformatics Coordinating Facility, http://www.incf.org) as a result of the 1st INCF Workshop on Large Scale Modelling of the Nervous System (Djurfeldt and Lansner 2007). A demonstration of MUSIC's capability to couple models was presented at the INCF booth at the Society for Neuroscience Conference in Washington 2008.

Here we report on our experiences and insights from connecting two preexisting models of very different kinds; one cortical network model using integrate-and-fire units and one striatal network based on biologically detailed units. The cortical network model was implemented in NEST while the striatal network model was developed using GENESIS, but simulated here in MOOSE. By adding a MUSIC interface to each simulator and connecting them using MUSIC, we could simulate the two systems together as one multi-simulation. Connecting the two models was interesting in its own right, but this would also serve as a realistic test of how hard it is to actually achieve interoperability between two independently developed models using the new MUSIC framework.

MUSIC

MUSIC is a recently developed standard for run-time exchange of data between MPI-based parallel applications in a cluster environment (Ekeberg and Djurfeldt 2009) so that any MUSIC-compliant tool may work out-of-the-box with another. A pilot implementation was released during 2009. The standard is designed specifically for interconnecting large scale neuronal network simulators, either with each-other or with other tools. The data sent between applications can be either event based, such as neuronal spikes, or graded continuous values, for example membrane voltages.

The primary objective of MUSIC is to support *multi-simulations* where each participating application itself is a parallel simulator with the capacity to produce and/or consume massive amounts of data. Figure 1 shows a typical multi-simulation where three applications, *A*, *B*, and *C*, are exchanging data during runtime.

MUSIC promotes *interoperability* by allowing models written for different simulators to be simulated together in a larger system. It also enables *reusability* of models or tools by providing a standard interface. The fact that data is spread out over a number of processors makes it non-trivial to coordinate the transfer of data so that it reaches the right destination at the right time. When applications are connected in loops, timing of communication also becomes complex. The task for

Neuroinform (2010) 8:43–60

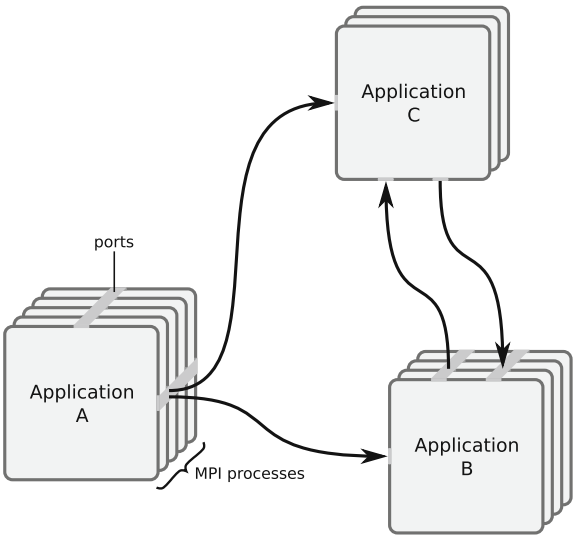


Fig. 1 Illustration of a typical multi-simulation using MUSIC. Three applications, *A*, *B*, and *C*, are exchanging data during runtime. Each application runs in a set of MPI processes. Data flows exit and enter *ports*, each spanning the set of processes of the application

MUSIC is to relieve the applications from handling this complexity.

The MUSIC pilot implementation consists of header files, a library and utilities. A MUSIC-compliant application is compiled against the MUSIC header files, linked with the MUSIC library and launched by the MUSIC launch utility (named `music`). Currently, an application needs to be compiled in a system where the full MUSIC implementation is installed. However, the resulting binary may be dynamically linked against other revisions of the MUSIC library. In a future release, standard header files can be separated so that these can be shipped together with the application.

The pilot implementation has been designed to run smoothly on state-of-the-art high-performance hardware. The software is written in C++, which is the de facto standard for current high-end hardware, and has been tested on simple multi-core machines up to massively parallel supercomputers such as the IBM Blue Gene/L. It can be automatically configured (GNU autotools) and compiled over the range of architectures tested, including 32- and 64-bit Intel-/AMD-based clusters and the Blue Gene/L.

At the beginning of this project, no simulators had been adapted to use MUSIC. In a collaborative effort, developers from the NEST and MOOSE communities worked together with the MUSIC developers in adapting these two simulators for use in a multi-simulation environment.

Phases of Execution

A multi-simulation, i.e. a set of interconnected parallel applications, is described by a MUSIC *configuration file* and executed in three distinct phases. From the simulator developers' point of view, these phases are clearly separated through the use of two main components of the MUSIC interface: the `Setup` and the `Runtime` objects.

Launch is the phase where all the applications are started on the processors. During this phase, MUSIC is responsible for interpreting the configuration file and launching the application binaries on the set of MPI processes allocated to the MUSIC job. Since MPI can be initialized only after the applications have been launched, most of this work needs to be performed outside of MPI. In particular, the tasks of accessing the command line arguments of the MUSIC launch utility and of determining the ranks of processes before MPI initialization therefore have to be handled separately for different MPI implementations.
Technically, the launch phase begins when `mpirun` launches the MUSIC launch utility and ends when the `Setup` object constructor returns. The Setup object is used for initialization and configuration and replaces the call to `MPI::Init`. (See further description below.)

Setup is the phase when all applications can publish what ports they are prepared to handle along with the time step they will use and where data will be present (where in memory and/or on what processor). During the setup phase, applications can read configuration parameters communicated via the common configuration file. At the end of the setup phase, MUSIC will establish all connections.
The setup phase begins when the `Setup` object has been created and ends when the `Runtime` object constructor returns.

Runtime is the phase when simulation data is actually transferred between applications. Via calls to `Runtime::tick()` the simulated time of the applications is kept in a consistent order.

The runtime phase begins when the `Runtime` object has been created and ends when its `finalize()` method is called.

When the application initializes MUSIC at the beginning of execution it receives the `Setup` object. This object gives access to the functionality relevant during the setup phase via its methods. When done with the setup, the application program makes the transition to the runtime phase by passing the `Setup` object as an argument to the `Runtime` object constructor which destroys the `Setup` object. The `Runtime` object provides methods relevant during the runtime phase of execution.

MUSIC requires that the application uses a communicator handed to it from the `Setup` object rather than using `MPI::COMM_WORLD` directly. This intra-communicator is used by the application to communicate within the group of processes allocated to it by MUSIC during launch, while the MUSIC library will internally use inter-communicators for communication of data between MUSIC ports. When a MUSIC-aware application is launched by `mpirun` instead of the MUSIC launch utility, the communicator returned from the `Setup` object will be identical to `MPI::COMM_WORLD`.

Communicating via MUSIC

In order to communicate via MUSIC, each participating application must be interfaced to the MUSIC API. One design goal of MUSIC has been to make it easy to adapt existing simulators. In most cases, it should be possible to add MUSIC library support without invasive restructuring of the existing code. The primary requirements on an application using MUSIC is that it declares what data should be exported and imported and that it repeatedly calls a function at regular intervals during the simulation to allow MUSIC to make the actual data transfer.

Ports and Indices

Communication between applications is handled by *ports*. Ports are named sources (output ports) or sinks (input ports) for the data flow. In the current MUSIC API, there are three kinds of ports: Continuous ports communicate multi-dimensional continuous time-series, for example membrane voltages. Event ports communicate time-stamped integer identifiers, for example neuronal spikes. Message ports communicate message strings, for example a command in a scripting language. The data to be communicated may be differently organized in process memory on the receiver side compared to the sender side. The applications may run on different numbers of processes, and, the data may be differently distributed among the sender processes and the receiver processes, as is shown in Fig. 2. How does MUSIC know which data to send where?

In MUSIC, there are two views of the data to be communicated over a connection. Data elements are enumerated differently according to these views. MUSIC uses *shared global indices* to enumerate the entire set of data to be sent over the connection while *local indices* enumerate the subset of data which is stored in the memory of a particular MPI process. Data does not need to be ordered in the same way according to the two views. For example, data stored in an array may be associated with an arbitrary subset of global indices in an arbitrary order.

The MUSIC library is informed about the relationship between global and local indices and how data is stored in memory during the setup phase. Two abstractions are used to carry this information:

The `IndexMap` maps local indices to global indices. That is, the `IndexMap` tells which parts of a distributed data array are handled by the local process and how the data elements are locally ordered.

The `DataMap` encapsulates how a port accesses its data. The `DataMap` contains an `IndexMap`. While an

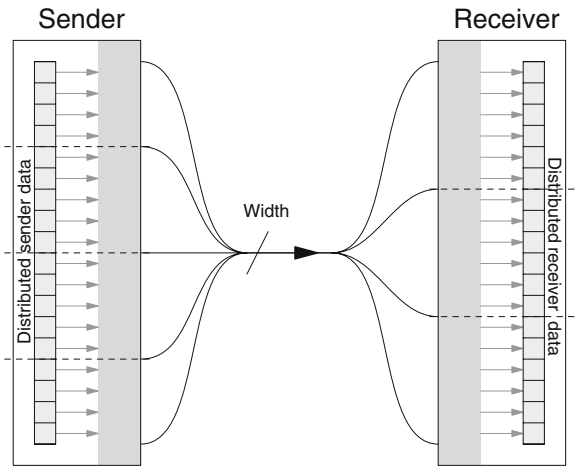


Fig. 2 Data transfer over a connection from an application running in four processes to an application running in three processes. The *light gray areas* in the sender and receiver represent the MUSIC port. *Dashed lines* divide the application into distinct processes. The *width* of the port is the total number of distinct data items being communicated from all sender processes to the receiver processes

index map is a mapping between two kinds of indices, the data map also contains information about where in memory data resides, how it is structured, and, the type of the data elements. The type is used for marshalling when running on a heterogeneous cluster.

During setup every process of the application individually provides the port with a `DataMap` (or an `IndexMap` in the case of event ports).

Events

Since event ports don't access data the same way as ports for continuous data, they do not require a full `DataMap`. Instead, an `IndexMap` is used to describe how indices in the application should be mapped to the shared global indices common to sender and receiver. The application is given the choice of using local indices or bypassing the index transformation by directly using the shared global indices when labelling events.

An event is a pair of an index identifier, either a shared global index or a local index, and a double-precision floating point time-stamp. The index usually refers to the source neuron generating a spike event. Events are given to MUSIC by the sending application through a call to the method `insertEvent`() on the port and delivered to the receiving application by MUSIC through an *event handler*. The event handler is a C++ functor given to MUSIC by the application before the simulation starts. The event handler is called by MUSIC when the application calls `tick`(). It is called once for every event delivered.

Some spiking neuronal network models include axonal delays. The MUSIC framework assumes that handling and delivery of delayed spikes occurs on the receiver side. In such a case, the receiver may allow MUSIC to deliver a spike event later than its time stamp according to local time. This *maximal acceptable latency* can be specified for a port during setup.

Application Responsibilities

One goal of MUSIC has been to limit the responsibilities imposed on each application. Here we present a step-by-step list of what an application must do in order to participate in a multi-simulation.

1. **Initiate MUSIC**
   This is done by calling the `Setup` constructor.
2. **Publish ports**
   Data available to be imported and exported is identified by creating named ports.
3. **Map ports**
   MUSIC is informed about where the actual data is located. This includes information about which processor owns each data element. For continuous data it also includes information about where in memory it is stored, while for event data it specifies how to receive events.
4. **Initiate the runtime phase**
   This is done by calling the `Runtime` constructor. At this stage, MUSIC can build the plan for communication between different processes.
5. **Advance simulation time**
   The application must call `tick`() at regular intervals to give MUSIC the opportunity to transfer data.
6. **Finalize MUSIC**
   By calling `finalize`(), all MUSIC communication is terminated.

Pre- and Post-processing

The MUSIC framework provides a uniform interface to access data from various simulators. This allows the development of pre- and post-processing tools, for example for data analysis or visualization, that are independent of data sinks or sources so that they can be re-used in different multi-simulations.

This is exemplified here by a visualizaton tool written for INCF's MUSIC demonstration at the Society for Neuroscience Conference 2008 in Washington. The tool receives events from a MUSIC event port and displays these as changes in size and color of a set of 3D spheres resembling the neurons of a neuronal network. The demonstration is described in Section "A MUSIC Multi-simulation with NEST and MOOSE" and the graphics window of the visualization tool shown in Fig. 13. The communication between the simulator and the visualization tool is set up using the MUSIC configuration file. The visualization geometry, neuron sizes and colors are specified in a separate configuration file. The camera position is automatically adjusted so that all neurons are visible.

MUSIC allows simulators to run independently of each other, in so far as one model might run ahead of another if there is only unidirectional communication between them. To cope with this, the visualization maintains its own internal clock which is a scaled version of the wall-clock time. For example, the visualization can be configured to display the simulation 100 times slower than real time. This makes the visualization independent of the relative execution time of the simulators.

Adapting NEST to MUSIC

NEST (Gewaltig and Diesmann 2007) is a simulator for heterogeneous networks of point neurons or neurons with a small number of electrical compartments. The focus of NEST is on the investigation of phenomena at the network level, rather than on the simulation of detailed single neuron dynamics.

NEST is implemented in C++ and can be used on a wide range of architectures from single- and multi-core desktop computers to super-computers with thousands of processors. It has a built-in simulation language interpreter (SLI), but can also be used from the Python programming language via an extension module called PyNEST. In this article, we use the PyNEST syntax to show the usage of the MUSIC interface in NEST. As Python does not support MPI enabled extensions out of the box, a small launcher script has to be used (see Section B in the online supplementary material). For details on PyNEST and its API, see Eppler et al. (2009).

Implementation of the NEST-MUSIC Coupling

Event sources and sinks that are located in remote MUSIC applications are represented by *proxy* nodes inside of NEST. Two separate classes of proxies are used for inbound and outgoing connections. They are derived from the base class `Node`. This means that they are created and can be connected in the network graph like all other nodes. See online supplementary material, Section A, for a more detailed description of the network representation in NEST.

To make it easier to distinguish *global ids* (NEST's identifiers for nodes) from *global indices* (MUSIC's identifiers for connections on a port, see Section "Ports and Indices"), we use the term *channel* for the concept from MUSIC in the following description and in our implementation.

Three new classes were implemented to exchange events with MUSIC. In addition, several of the existing classes were extended by data structures and algorithms for the necessary book keeping during setup and runtime phase. The following sections contain a description of the components that are involved in the MUSIC interface. See online supplementary material, Section D, for sequence diagrams that explain the interaction of the components.

Sending Events from NEST to MUSIC

The class `music_out_proxy` represents a MUSIC output port and all associated channels in NEST (see Fig. 3). It forwards the events of arbitrarily many nodes to remote MUSIC targets. One instance of this proxy is created in each NEST process for each MUSIC output port.

The name of the corresponding MUSIC output port is set as parameter `port_name` using `SetStatus()`:

```
outproxy = Create('music_out_proxy')
SetStatus(outproxy, {'port_name':
                     'spikes_out'})
```

The events of a node are forwarded to the MUSIC channel that is specified by the parameter `music_channel` during connection setup. It cannot be changed, once the connection is set up. Note that it is not allowed to connect several nodes to the same channel. The following example shows how the connections of five neurons to a MUSIC port are set up:

```
neurons = Create('iaf_neuron', 5)
Connect([neurons[0]], outproxy,
        {'music_channel': 0})
Connect([neurons[1]], outproxy,
        {'music_channel': 1})
Connect([neurons[2]], outproxy,
        {'music_channel': 2})
Connect([neurons[3]], outproxy,
        {'music_channel': 3})
Connect([neurons[4]], outproxy,
        {'music_channel': 4})
```

During connection setup in NEST, the sender checks its compatibility with the receiver by calling its `connect_sender()` function. The first argument for this function is an event of the type the sender wants to send during simulation, which is only used to select the correct variant of `connect_sender()`. The second

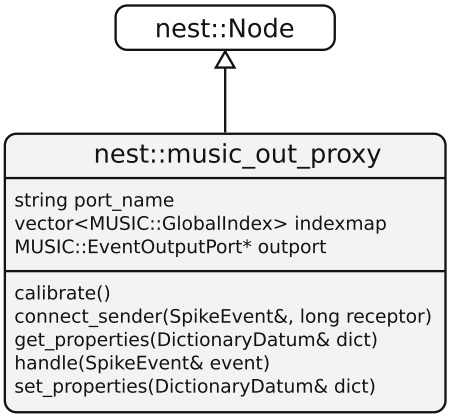


Fig. 3 The UML diagram shows the data members and the functions of the proxy that represents MUSIC output ports in NEST. The new class is shown in *grey*

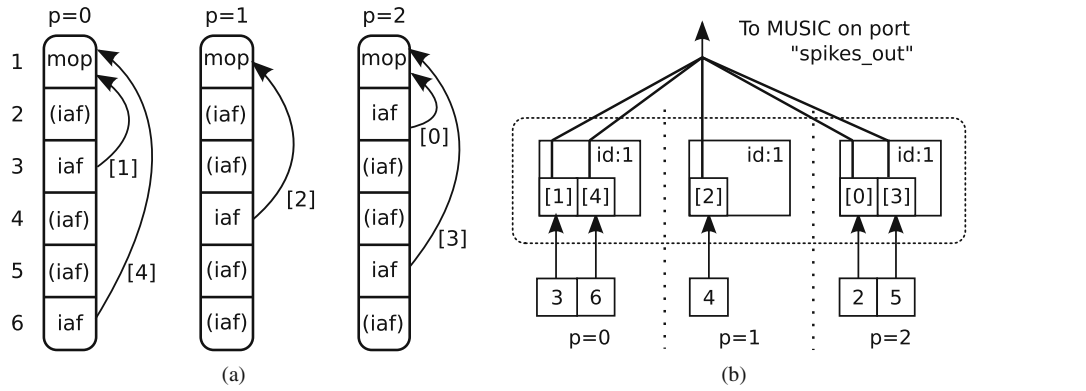


Fig. 4 **a** Nodes in NEST are distributed over the processes ($p = 0, 1, 2$). iaf denotes an integrate and fire neuron, (iaf) denotes a proxy. mop denotes a `music_out_proxy`. MUSIC channels are indicated in *square brackets* for each connection (*arrows*). **b** A sketch of the complete connectivity from the nodes (*lower squares*) over the different channels (*numbers in square brackets*) to MUSIC. The *dashed box* encloses all proxies that belong to one MUSIC output port

argument is an integer which specifies the channel the source wants to connect to, and is used to build the `indexmap`, a list that registers all channels that have to be mapped with MUSIC. The `indexmap` is built separately by each process and therefore only contains local channels.

Figure 4 shows the network in NEST after the above commands were executed using three NEST processes.

Before NEST tells MUSIC to enter the runtime phase, the port has to be mapped. This is done in `calibrate()`, which is called by NEST's scheduler on each node before the start of the simulation. This function executes the following steps:

1. Create a `MUSIC::EventOutputPort`, `outport`. This will trigger an exception if the port name is already used.
2. Create a `MUSIC::PermutationIndex` and initialize it with the data from the `indexmap`. The `PermutationIndex` informs MUSIC about the channels present on a specific process.
3. Use the `PermutationIndex` to map all local channels by calling the function `map()` on `outport`.

Events that are delivered to the proxy are passed to its `handle()` function with the event as argument. This function forwards the spikes directly to the MUSIC output port object `outport`.

Note that as the `music_out_proxy` only acts as a proxy for nodes in NEST, it does not take into account the delay of incoming connections. Synaptic interactions have to be set up in the receiving application.

Receiving Events from MUSIC in NEST

In contrast to MUSIC output ports, which are represented by single `music_out_proxy`s in NEST, inbound connections require two classes: The MUSIC input port is represented by the class `MusicEventHandler` (see Fig. 5). One `MusicEventHandler` is created for each MUSIC

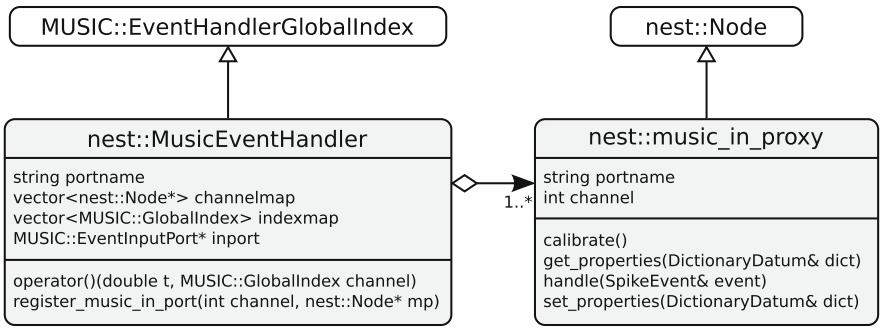


Fig. 5 The UML diagram shows the data members and the functions of the proxy that represents a channel on a MUSIC input port in NEST and its relation to the class that represents the MUSIC input port. New classes are shown in *grey*

input port in each of NEST's processes. Each channel on the port is represented by a separate `music_in_proxy` (see Fig. 5). The reason for this is that NEST's connection mechanism cannot handle different signal origins, but only different target locations on a node. This means that we cannot specify the MUSIC channel during connection setup to a single proxy (cf. Section "Sending Events from NEST to MUSIC"), but we need to set it separately as a parameter for the `music_in_proxy`, which should receive the respective input.

The `MusicEventHandler` maintains a `channelmap`, which maps the global MUSIC channel id to the address of the corresponding proxy. The `channelmap` is built incrementally during the registration of channels by `register_channel`().

Spike sources in remote MUSIC applications are represented in NEST by instances of class `music_in_proxy`. Each instance listens to exactly one channel on a MUSIC input port. This means that several proxies listen to the same port, but to different channels.

After the creation of the proxy, the port name and the channel are set using `SetStatus`(). The port name defaults to *spikes_in* for all `music_in_proxy`s.

```
in_proxies = Create('music_in_
                     proxy', 2)
SetStatus([in_proxies[0]],
          {'music_channel': 0})
SetStatus([in_proxies[1]],
          {'music_channel': 1})
```

Connections from a `music_in_proxy` to other nodes can use any of NEST's built-in connection types. The following listing shows how connections are established using the high-level connection routine `DivergentConnect`() and the basic connection command `Connect`():

```
neurons = Create('iaf_neuron', 4)
DivergentConnect([in_proxy[0]],
                 [neurons[0], neurons[1]])
DivergentConnect([in_proxy[1]],
                 [neurons[1], neurons[2]])
Connect([in_proxy[0]], [neurons[3]],
        model='stdp_synapse')
```

Figure 6 shows the network representation after the above commands were executed in a setup with three NEST processes.

As the proxy itself does not know about MUSIC, we use an indirection via the `Network` class to register the proxy with the event handler for the corresponding port.

In its `calibrate`() function, the proxy registers itself with its channel index and port name with the `Network` class by calling `register_music_in_proxy`(). The network class maintains a mapping of port names to MUSIC event handlers to efficiently find the right one or create a new instance for unknown ports if an input proxy is registrered. Before the start of the simulation, all known input ports are mapped.

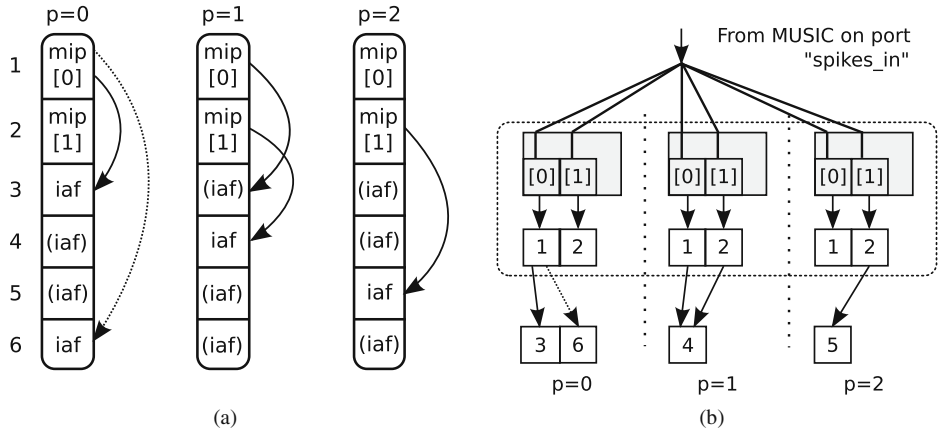


Fig. 6 **a** Nodes in NEST are distributed over the processes ($p = 0, 1, 2$). iaf denotes an integrate and fire neuron, (iaf) denotes a proxy. mip denotes a `music_in_proxy`. The *numbers on the left* indicate the global id of the nodes. MUSIC channel ids are indicated in *square brackets* for each `music_in_proxy`. The STDP connection is indicated by a *dashed arrow*. **b** A sketch of the complete connectivity from MUSIC (*channels in square brackets*) to the MUSIC event handler (*grey rectangles*) to the proxies (*squares labeled 1 and 2*) to the actual target nodes (*lower squares*). The STDP connection is indicated by a *dashed arrow*. The *dashed box* encloses all event handlers and proxies that represent a MUSIC input port

For each incoming spike, MUSIC calls `operator()` on the event handler with the time of the spike and the target channel as arguments. `operator()` creates a new `SpikeEvent` object and passes it directly to the `handle()` function of the proxy associated with the channel. This bypasses the synapse system in NEST and only informs the proxy about a new spike in a remote application. Upon arrival of new events, the `handle()` function immediately calls `Network::send()` to deliver the event to all local targets via the synapse system.

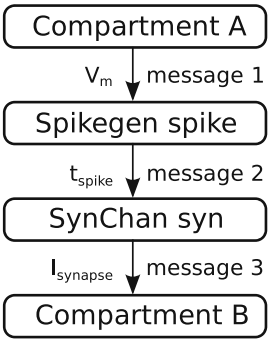


Fig. 7 Illustration of the MOOSE messaging structure. Two compartments are connected by a synapse

Adapting MOOSE to MUSIC

MOOSE (Multiscale Object Oriented Simulation Environment, available at `http://moose.ncbs.res.in`) is a simulator which enables the development and simulation of biologically detailed models of neuronal and biochemical networks. It is a multiscale simulator, as it lets a modeller build a model by coupling components from different levels of detail—from single molecules to whole neurons. It achieves this by coordinating calculations between specialized numerical engines which are suited for each level of detail.

To discuss how MUSIC compatibility was added to MOOSE, it will be helpful to outline how MOOSE functions. MOOSE inherits an object-oriented framework from the GENESIS simulator (Bower and Beeman 1998) for describing models and simulations. In this framework, the user sets up a simulation by putting together the right building blocks ("MOOSE objects"), which are instances of the respective MOOSE classes.

Objects in MOOSE communicate with each other by means of "messages". A message is a persistent connection between two objects, which allows them to exchange information during a simulation. An example of messaging is shown in Fig. 7 which depicts how one can model synaptic transmission in MOOSE. A `SpikeGen` object called `spike` monitors the membrane potential `Vm` of the presynaptic compartment `A`, via the message labelled `message 1`. When this membrane potential crosses a certain threshold, `spike` interprets it as an action potential and sends the spike time to the `SynChan` (short for "Synaptic Channel") object called `syn`. This triggers the opening of the synaptic channel, and `syn` sends the synaptic current to the postsynaptic compartment `B`, via the message `message 3`.

MOOSE provides a user- and developer-friendly framework to run parallel simulations on a cluster. It hides MPI-based communication behind an interface so that sending and receiving information to and from foreign objects looks the same to the developer as with local objects. For the user, the design is such that a serial simulation script can be run in parallel right away, without any changes. In particular, for inspecting and manipulating objects and their fields and messages, the same script commands work in serial and in parallel operation. During object creation, a load-balancer decides which process the object should be created on.

Implementation of the MOOSE-MUSIC Coupling

New Classes

Five new MOOSE classes were created to allow MOOSE to exchange spike times with MUSIC:

- `Music`—This is a singleton class with exactly one instance automatically created at the start of a MOOSE session.
  This object is responsible for making most of the basic MUSIC API calls in the correct order. This includes appropriate initialization and finalization by managing the MUSIC `Setup` and `Runtime` objects. Also, during a simulation, this object calls MUSIC's `tick()` function periodically, separated by a user-specified time interval.
  While this `Music` object carries out the above without user intervention, it also provides an interface which the user can use to create new MUSIC ports for sending and receiving spike-event information.
- `InputEventPort`—An instance of this class is created when the user calls a function of the above `Music` class to declare readiness to receive spike-event information. Upon creation, this object finds out the width of the corresponding MUSIC port by making a MUSIC API call. A corresponding

number of instances of the `InputEventChannel` class (described next) are then created.

- `InputEventChannel`—Instances of this class act as proxies within MOOSE of the spike-generating entities in the sending application. They receive spike-time information relayed by MUSIC and recreate the original spike-train by emitting the spike-times locally. Hence, within MOOSE, they appear as bona fide spike-generating objects which can connect to, e.g. a `SynChan` object, and send spike messages just like a `SpikeGen` object can.
- `OutputEventPort`—This class is analogous to the `InputEventPort`, and is instantiated by the user when MOOSE should act as a spike-generating application. Like before, an instance of this class creates the same number of instances of the following `OutputEventChannel` as is its own width.
- `OutputEventChannel`—Objects of this class can receive spike-time messages from other MOOSE objects, like a `SynChan` object can. Upon receiving a spike, an `OutputEventChannel` object passes it on to MUSIC, which forwards it to interested applications.

Note that if the user creates a port of width m in a parallel simulation with n processes, then m channel objects (i.e., instances of `InputEventChannel` or `OutputEventChannel`) will have to be distributed among the n processes. An algorithm is built into the port classes (i.e., `InputEventPort` and `OutputEventPort`) to carry out this distribution, without the need for the user's knowledge. At present, this algorithm is simple: the list of m is divided into n blocks of size approximately equal to m/n, and channels within each block are created on a separate node. In the future, this algorithm can be improved by placing the channel objects in the same process as the objects they connect to.

Interfacing with MUSIC in MOOSE

With the above classes at hand, it is simple for a user to incorporate MUSIC sources and sinks in a simulation. The user carries out the following steps to set up MOOSE-MUSIC communication:

- Specifying `tick`() rate—The user provides a time interval which is used to call MUSIC's `tick`() function periodically. This is done by setting up a "clock" with the desired time interval as its clock-rate, and attaching it to the instance of the singleton `Music` class. See online supplementary material, Section E, for an example MOOSE script, which has commands to carry out all steps mentioned here.
- Adding ports—The user declares the ports through which MOOSE can receive and send data via MUSIC. One script command has to be issued for every port added.
- Connecting MUSIC with the model—With the above two steps done, the user has MUSIC sources and sinks available as native MOOSE objects. From here on, it is intuitive for a user to route MUSIC-originating and MUSIC-destined data to- and from desired entities in a model. This is done by simply adding messages, in the usual MOOSE fashion, between objects representing MUSIC, and objects constituting the model. Note that in adding a message, the user need not do anything special if the source and destination objects are situated in different processes, since MOOSE will carry

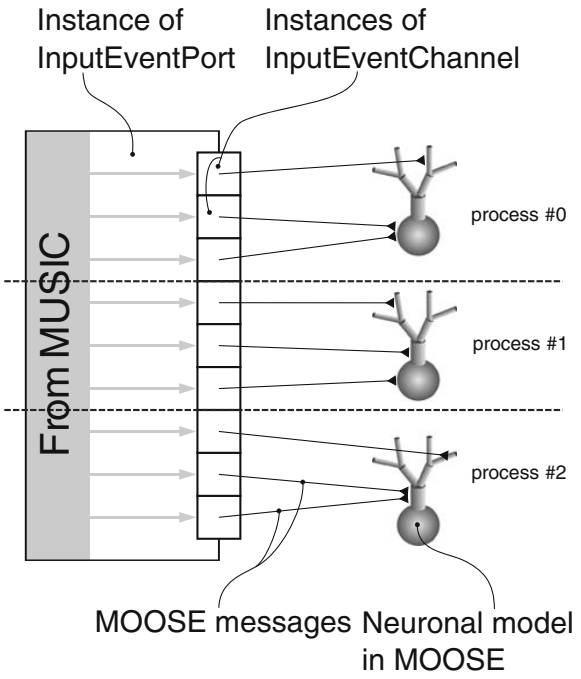


Fig. 8 A model in MOOSE receiving spike-time information from MUSIC. An object of type `InputEventPort` handles spike-times relayed by MUSIC. Objects of type `InputEventChannel` act as proxies for the spike-generating entities in the foreign application. The proxies forward the spikes to targets in the model via messages. Note that it is possible for a message to connect a proxy and its target even if both are in separate processes. It is most efficient, however, if they are on the same process

out the correct setup internally. This situation is depicted in Fig. 8.

Performance and Application

The adaptation of NEST and MOOSE to MUSIC allows us to test the performance of MUSIC and to apply the framework to a multi-simulation connecting two very different models. We test the performance of MUSIC in two typical multi-simulation examples: (1) an asymmetric multi-simulation benchmark with one large-scale model that is connected bidirectionally to a second program that runs on a single process (see Fig. 9a) and (2) a symmetric multi-simulation benchmark with MUSIC connecting two large-scale models each running on multiple processes (see Fig. 9b). For simplicity, we use NEST for the benchmarks presented here. As a complete application example, we present a multi-simulation that connects two very different models: a cortical network model based on integrate-and-fire units in NEST and a striatal network model based on multi-compartmental units with Hodgkin-Huxley formalism in MOOSE.

In Section "Benchmarking MUSIC with a Cortical Network Model in NEST" we describe consecutively the model definition and the performance of the cortex model, the asymmetric multi-simulation benchmark and the symmetric multi-simulation benchmark. Section "A MUSIC Multi-simulation with NEST and MOOSE" then describes the multi-simulation connecting the cortex and the striatum network and shows simulation results.

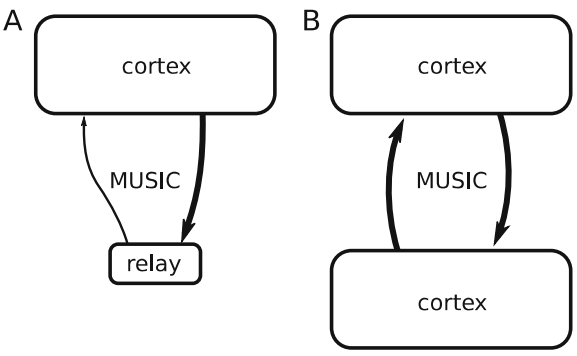


Fig. 9 Benchmark models. **a** Asymmetric benchmark model consisting of one large-scale cortex model and a single process relay model. Inter-model communication via MUSIC is bidirectional but asymmetric, mainly from the cortex model to the relay model. **b** Symmetric benchmark model consisting of two interconnected large-scale cortex models. Communication via MUSIC is symmetric between the two models

Benchmarking MUSIC with a Cortical Network Model in NEST

We use a layered cortical network model (Potjans and Diesmann 2008) in NEST in order to assess the performance of MUSIC for large-scale simulations. It consists of 80,000 integrate-and-fire units divided among four layers (2/3, 4, 5 and 6) and around 0.3 billion synapses. Each layer contains one excitatory (e) and one inhibitory (i) population. Populations are connected randomly with layer- and type-specific connection probability. In all benchmark simulations, we use static synapses. The integration step size is 0.1 ms and the minimal delay in the network is `min_delay` = 0.8 ms. The network exhibits asynchronous irregular activity with layer- and type-specific firing rates for stationary, homogeneous background input. The mean firing rates range from below 1 Hz to maximally 8 Hz (Potjans et al. 2009).

Performance of the Cortex Model

From the computational perspective, simulating this model is a rather lightweight job on modern compute clusters. We simulate the model with NEST on a ×86 cluster consisting of 23 nodes: each node is equipped with two AMD Opteron 2834 Quad Core processors with 2.7 GHz clock speed and running Ubuntu Linux. The nodes are connected via InfiniBand; the MPI implementation is OpenMPI 1.3.1. Our simulation setup first distributes the processes to nodes, resulting in a single process per machine and InfiniBand port up to 23 processes. Figure 10a shows the computing time per second of biological time as a function of the number of cores on this system: black squares show the data for the default installation of NEST, gray diamonds when linking NEST during compilation to MUSIC. The overlap of the data points shows that the performance of NEST is not impeded when using the MUSIC communicator; the performance is the same when NEST is compiled by default with the configure switch `--with-music`. In both cases, the simulation time of the layered cortex model scales supralinearly up to 20 cores and linearly up to 24 cores, yielding a simulation time of only 8 s per second of biological time. A further increase of the number of processors still improves the simulation time; using 48 cores results in a simulation time of around 5 s. The suboptimal scaling when increasing the number of processes from 24 to 32 is due to limited memory bandwidth that comes into play when multiple processes run on a single compute node. Beyond 32 processes the scaling is again close to optimal linear scaling. This simulation represents the

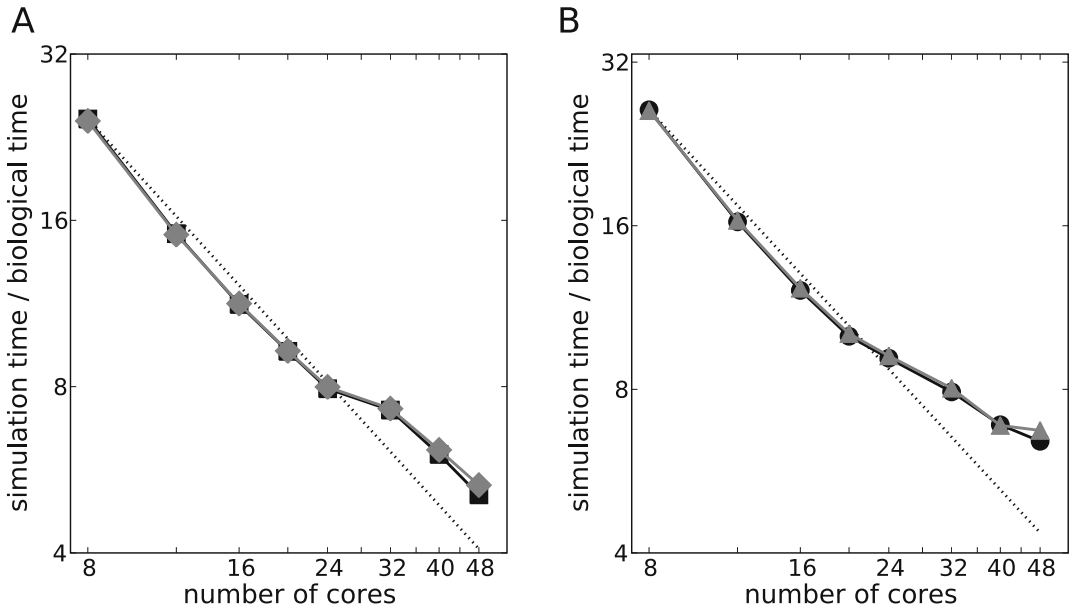


Fig. 10 Performance of the layered cortical network model and the asymmetric multi-simulation benchmark. **a** Computing time per second of biological time as a function of the number of compute cores. *Gray diamonds* show the performance of the cortex model simulation when NEST is compiled without MUSIC, *black squares* the performance when compiled with MUSIC. The *dotted line* indicates the expectation for linear speed-up. **b** Computing time per second of biological time of the asymmetric multi-simulation benchmark. The number of cores corresponds to the number of cores used for the cortex model without the additional core for the relay network. The shown data corresponds to $N_{\mathrm{MUSIC}} = 8$ (*black circles*) and $N_{\mathrm{MUSIC}} = 71,000$ (*gray triangles*); the *dotted line* gives the expectation for linear speed-up

control simulation for the asymmetric multi-simulation benchmark.

Asymmetric Multi-simulation Benchmark

The asymmetric multi-simulation benchmark consists of two models implemented in NEST: the cortex model and a basic relay model (see Fig. 9a). The models are coupled bidirectionally via MUSIC, i.e. both models send/receive spike events to/from the other model. Basically, we record spikes from any population in the layered network and transmit them to the relay model. The relay model takes few of the transmitted spikes and sends them back to the sender population. The communication is layer- and type-specific: we transmit the spikes to/from any population in the cortex model separately.

The relay model is kept minimal. It consists of one `parrot_neuron` for every population in the cortex model; this neuron immediately emits a spike for every spike it receives. The `parrot_neuron`s receive a subset of the spike trains transmitted from its population in the cortex model from the corresponding `music_in_proxy`s in the relay model and sends its spikes to the corresponding `music_out_proxy`. Therefore we always have a fixed number of eight MUSIC channels transmitting spikes from the relay model to the cortex model.

The implementation of the benchmark requires changes to the cortex model script and the relay script. But as the communication with MUSIC is carried out by nodes—`music_out_proxy`s and `music_in_proxy`s—the multi-simulation NEST scripts do not differ fundamentally from scripts describing stand-alone simulations. We create and connect `music_out_proxy`s and `music_in_proxy`s for any population in the model as described in Sections "Sending Events from NEST to MUSIC" and "Receiving Events from MUSIC in NEST", respectively. In addition, we have to set the acceptable latency with the `SetAcceptableLatency()` command. Care has to be taken in order to arrive at consistent parameters in the NEST scripts and the corresponding MUSIC script of the multi-simulation. On the level of a multi-simulation, these parameters are the number of MUSIC channels per population going from the cortex model to the relay model and vice versa—we call the total number of efferent MUSIC channels of the cortex model N_{MUSIC}.

On the level of the NEST scripts, we account for the asymmetry of a single `music_out_proxy` with many `music_channel`s per population on the one hand and many `music_in_proxy`s with the corresponding `music_channel`s on the other hand.

Altogether, the asymmetric multi-simulation benchmark extends the stand-alone cortex model by the following parameters:

- $N^{i,x}_{\mathrm{MUSIC}}$: number of efferents of the cortex model per population ($i \in \{2/3, 4, 5, 6\}$, $x \in \{\mathrm{e}, \mathrm{i}\}$). $N_{\mathrm{MUSIC}} = \sum_{i,x} N^{i,x}_{\mathrm{MUSIC}}$ corresponds to the number of MUSIC channels from the cortex model to the relay model and therefore also to the number of `music_in_proxy`s in the relay model
- $k^{i,x}_{\mathrm{MUSIC}}$: number of connections between the $N^{i,x}_{\mathrm{MUSIC}}$ `music_in_proxy`s and the corresponding `parrot_neuron` in the relay model.

Figure 10b shows the performance of the asymmetric multi-simulation benchmark for $N_{\mathrm{MUSIC}} = 8$, $k^{i,x}_{\mathrm{MUSIC}} = 1$ (black circles) and for $N_{\mathrm{MUSIC}} = 71,000$, $k^{i,x}_{\mathrm{MUSIC}} = 4$ (gray triangles). For better comparison, we give here the number of cores used for the cortex model, the relay model is simulated on one additional core.

We find that the additional costs due to the MUSIC interfaces and due to the communication of the two models via MUSIC are very small. The multi-simulation scales supralinearly up to 20 cores and the relative increase in simulation time is well below 10% of the simulation time of the control simulation without MUSIC. Further increasing the number of cores still improves the simulation time below 7 s; only when using 48 cores, the additional costs lead to an earlier onset of the saturation of the simulation time. The excellent performance holds for the minimal case where we only transmit the spikes of a single neuron from every population, but also when transmitting almost all spikes created by the cortex model: We do not observe a dependence of the number of MUSIC channels/the number of transmitted spikes for this benchmark.

Symmetric Multi-simulation Benchmark

The symmetric multi-simulation benchmark increases the demands on software and hardware considerably. It consists of two reciprocally connected cortex models (see Fig. 9b). Each model connects, as in the asymmetric multi-simulation benchmark, via in total N_{MUSIC} MUSIC channels to the other model. The incoming spike trains project on $k^{i,x}_{\mathrm{MUSIC}}$ neurons of the corresponding population. We choose very low synaptic weights for the connections between the two models in order to not interfere with the dynamics of the layered network. The random connectivity of the networks requires MUSIC to route events not only between single machines but rather in an all-to-all fashion.

The implementation of this benchmark does not require any changes to the cortex model script with respect to the asymmetric multi-simulation benchmark. The only changes affect the MUSIC script, configuring two interconnected and equally sized NEST simulations of the same model.

Figure 11a shows the performance of the symmetric multi-simulation benchmark. The given number of cores corresponds to the multi-simulation with both models. The control simulation for this benchmark (black squares) is defined by the multi-simulation of both cortex models without any connections between the models ($N_{\mathrm{MUSIC}} = 0$). Only the first data point (16 cores) corresponds to the situation with a single process per InfiniBand port. Still, we observe linear scaling up to 24 cores. Beyond this, the simulation time scales up to 96 cores, yielding a simulation time of 6 s per second of biological time.

The minimal benchmark ($N_{\mathrm{MUSIC}} = 8$, $k^{i,x}_{\mathrm{MUSIC}} = 1$, dark gray diamonds) also exhibits excellent scaling, but the simulation time increases by 1.3 ± 0.4 s. This difference in simulation time, however, does not show a clear dependence on the number of cores. Communicating 1,000 spike trains for every population ($N_{\mathrm{MUSIC}} = 8,000$, $k^{i,x}_{\mathrm{MUSIC}} = 1,000$, light gray circles) results in an additional increase of 1.3 ± 0.4 s, again with excellent scaling and no clear dependence of the increase in simulation time of the number of cores.

In order to understand this increase in simulation time, we simulate the symmetric multi-simulation benchmark for various values of N_{MUSIC} (keeping $k^{i,x}_{\mathrm{MUSIC}} = 1,000$ constant) and convert the number of MUSIC channels with the measured firing rates of the different populations in the cortex model into the MUSIC spike rate, the total number of spikes that is transmitted in one biological second from one cortex model to the other. Figure 11b shows the simulation time per second of biological time as a function of this MUSIC spike rate for a fixed number of cores (data obtained with 32 cores in the multi-simulation is shown in black, with 64 cores in dark gray). The dashed lines indicate the control multi-simulations with two unconnected cortex models ($N_{\mathrm{MUSIC}} = 0$). While the asymmetric multi-simulation benchmark is independent of the MUSIC spike rate (see above), the simulation time does depend on the MUSIC spike rate for the symmetric multi-simulation. Note, however, that this number is representing the spike rate transmitted via MUSIC

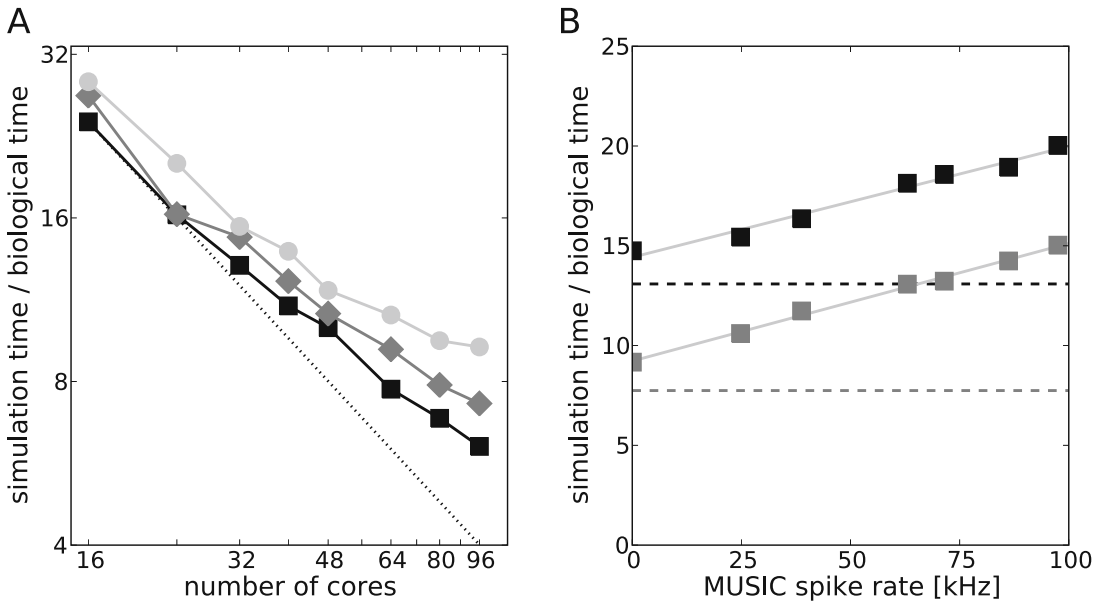


Fig. 11 Performance of the symmetric multi-simulation benchmark. **a** Simulation time per second of biological time as a function of the total number of compute cores for both network models. *Black squares* show the performance of the control ($N_{\mathrm{MUSIC}} = 0$), *dark gray diamonds* the benchmark's performance for $N_{\mathrm{MUSIC}} = 8$ and *light gray circles* for $N_{\mathrm{MUSIC}} = 8,000$. The *dotted line* indicates the expectation for linear speed-up of the control. **b** Simulation time per second of biological time as a function of the MUSIC spike rate. Simulations with 32 cores are indicated in *black*, with 64 cores in *dark gray*. *Dashed lines* indicate the control ($N_{\mathrm{MUSIC}} = 0$) and *squares* show the data for the symmetric multi-simulation benchmark with *light gray lines* showing the corresponding linear fits

in every direction and that the information has to be routed to all processes running the corresponding cortex model. We find that the simulation time increases linearly with the MUSIC spike rate (light gray lines indicate the linear fit). For 32 cores, the simulation time increases by 0.56 s when increasing the MUSIC spike rate by 10 kHz. For 64 cores, this number is only slightly increased to 0.6 s/10 kHz in the MUSIC spike rate.

A MUSIC Multi-simulation with NEST and MOOSE

A MUSIC multi-simulation was performed by connecting the layered cortical network model in NEST to a striatal network model in MOOSE. Activity of both simulations were visualized using the tool described in Section "Pre- and Post-processing" (see Fig. 12).

For the live demonstration, we reduced the size of the layered cortical network model to 8,000 neurons. The output consisted of spike events generated in the excitatory population of layer 5 that were exported through a MUSIC port.

The striatal network model was built using multi-compartmental units with Hodgkin-Huxley formalism and consisted of ten striatal medium spiny projection neurons with 189 compartments each (Wolf et al. 2005; Hjorth et al. 2008) and ten fast spiking interneurons with 127 compartments each (Hellgren Kotaleski et al. 2006). The cell models were ported from NEURON

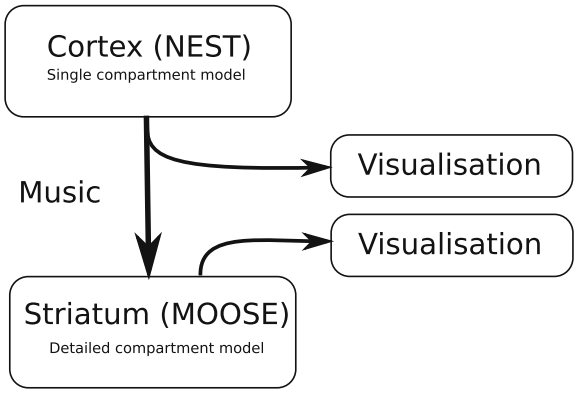


Fig. 12 Schematic of run-time interoperability for a cortico-striatal model. The cortical model simulated in NEST uses MUSIC to send spikes to the striatal model in MOOSE. In addition, two visualization processes receive the spike information from both NEST and MOOSE

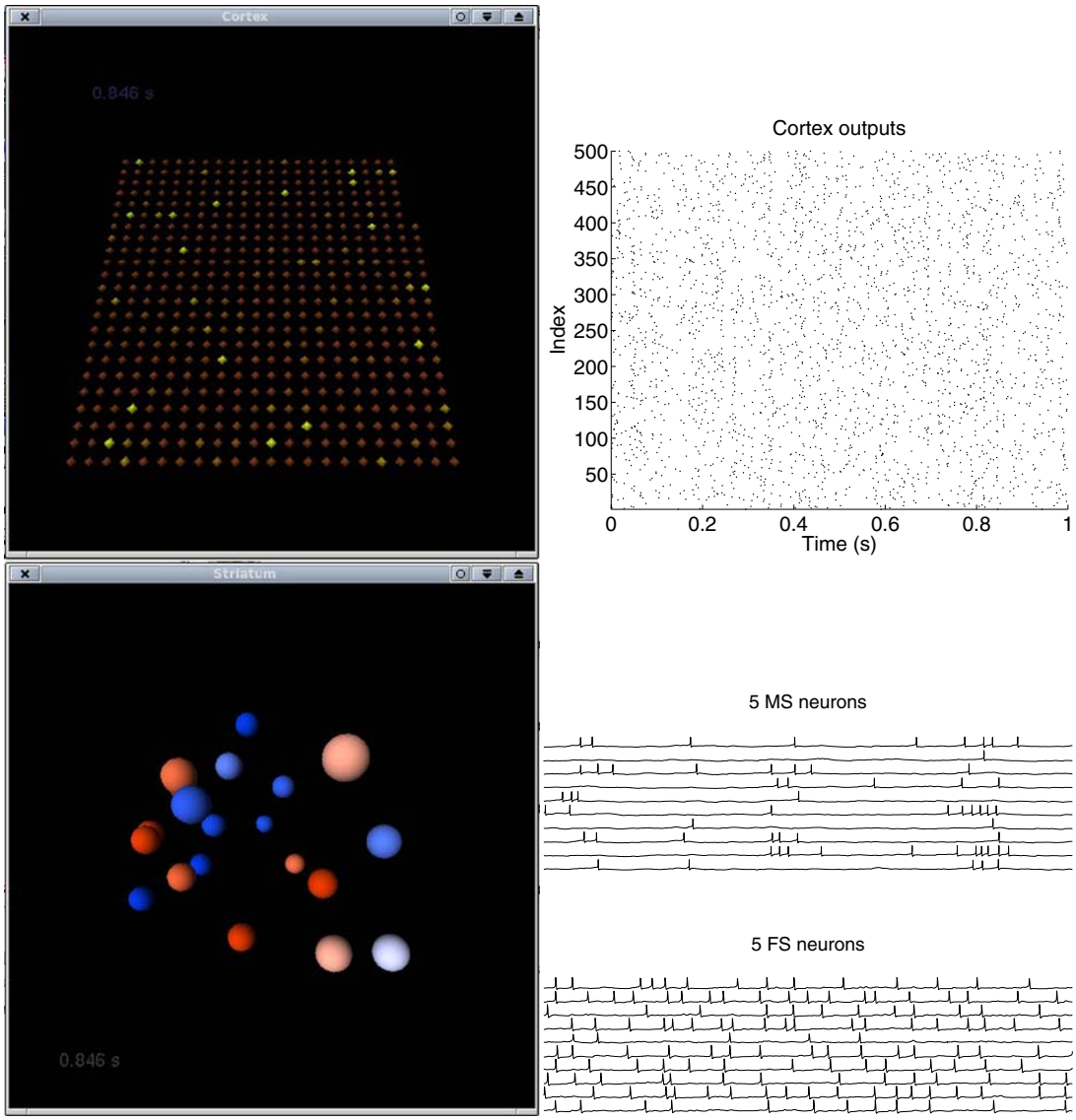


Fig. 13 Results from the multi-simulation described schematically in Fig. 12. To the *left*, two window captures from 3D visualizations of the cortex and striatum model are shown. In the *upper half* of the figure, 500 outputs from the cortex model in NEST are visualized on a planar grid, the radii and intensity of the color of the neurons increase when they spike. In the *lower part*, 10 MS (*red*) and 10 FS (*blue*) neurons in the striatal network are visualized in the same manner. To the *right* are a raster plot of the cortical activity and voltage traces for the MS and FS neurons

and GENESIS, respectively, to MOOSE. In this reduced version of the striatum for the MUSIC demonstration, no GABAergic connections were included between the two neuron populations. A MUSIC input port delivered spike events to both populations.

A short MUSIC configuration file described the multi-simulation and specified connections between the cortex output port and the striatum input port, and also connections from both models to one instance each of the visualization tool. Figure 13 shows captures of windows from the simulation tool instances together with simulation results from each model.

Since the MUSIC API enforces independence between the applications, the multi-simulation could be built from the cortex model and the striatum model without changes to their simulation scripts in other

respects than the creation of MUSIC ports and the addition of the cortico-striatal projection on the receiver side. Spike events from NEST could easily be routed to MOOSE as well as a visualization process without further changes to the simulation scripts.

Discussion

The multi-simulation described in the previous section is a demonstration of how MUSIC can promote interoperability between models written for different simulators and how these can be re-used to build larger model systems. Alternative approaches to run-time interoperability are object-oriented frameworks (as illustrated by MOOSE itself; see Cannon et al. 2007) and using a common standard model description language.

Object-oriented frameworks provide APIs for services such as solvers, scheduling of events and communication, while specialized modules correspond to entities in the neuronal model. In comparison, the MUSIC API is slim, essentially only providing what is necessary to support communication through MUSIC ports. In a sense, the approach of MUSIC is orthogonal to that of an object-oriented framework, implying that these approaches can, in fact, be combined, as illustrated by the MOOSE simulation in this article. Writing a module for an object-oriented framework usually means a commitment to that framework. On one hand, the object-oriented framework lifts some of the burden of implementation by providing services. On the other hand, it will only be possible for the module to communicate with other modules in the same object-oriented framework. In contrast, any simulator or tool supporting the MUSIC interface can be connected to the rest of the set of tools supporting MUSIC. In fact, any module written for an object-oriented framework which supports MUSIC will also be possible to connect to such tools supporting MUSIC.

One example of a framework targeting a similar problem domain as MUSIC is the component-based extension framework by King et al. (2009). This framework provides three APIs, one for a *compute engine*, exemplified by a specially compiled version of NEURON (Carnevale and Hines 2006), a *message-bus component*, allowing the encapsulation of a spike communication algorithm, and, a *monitoring, analysis and control component*. This framework could, as MUSIC, be used to set up multi-simulations and promote interoperability and re-use of existing components. While both solutions are non-exclusive in the sense that they could potentially co-exist with each other and/or other communication frameworks, MUSIC does not require the re-organization of an existing simulator into a library providing the compute engine API and is in this way less invasive. Also, MUSIC abstracts connectivity at two levels, as ports and as shared global indices within ports, thereby making it possible to easily connect pluggable components into different configurations specified by a configuration file. The component engine API leaves the handling of the lowest level of connectivity entirely to the user (in the form of NEURON "gids"). This creates dependencies between the configurations of components of a multi-simulation so that the re-use of a tool requires a different mapping of gids.

The approach of a common standard model description language, such as PyNN (Davison et al. 2009) or NeuroML (Crook and Howell 2007; Crook et al. 2007), enables the same model description to be used with different simulators. This circumvents the difficulty of reimplementing models when moving them from one software to another. This approach also has the strength that it makes the model future-proof. But even in the presence of such a standard, we cannot combine two models of different kinds (for example a model based on integrate-and-fire units and a model based on Hodgkin-Huxley formalism) if our favorite software does not support both forms of modeling. Ultimately, we must recognize the value in specialized tools optimized for a particular purpose. A scripting language environment such as Python can bind tools together by loading them as libraries (Ray and Bhalla 2008). MUSIC is another alternative. Thus, we again see that MUSIC should be seen as providing orthogonal functionality.

Apart from interoperability, MUSIC also provides efficient communication between parallel applications enabling multi-simulation of large-scale neuronal systems.

One of the strengths of the MUSIC API design is that it allows for establishing a deterministic communication schedule which removes the need for handshaking. This has also been exploited in the implementation. The downside of this design choice is that new MUSIC ports cannot be added once the simulation is run, as MUSIC cannot change back to the setup phase once the runtime was entered. It is conceivable, though, that a future version of the standard could allow for changes to the communication graph during simulation without requiring handshaking during communication.

The adaptation of NEST to MUSIC was straightforward. The changes were not extensive and fell naturally into the existing structure of the code. MUSIC concepts

such as ports were mapped to NEST proxies, MUSIC events could be routed by the proxies into the standard spike event delivery mechanisms. In the NEST simulator kernel, only five of the existing compilation units were affected: the scheduler and the units for MPI communication, network administration, scripting language binding and error handling. New compilation units were added for the MUSIC event handler and the NEST representations of MUSIC ports (`music_out_proxy`, `music_in_proxy`). The handling of the MUSIC `Setup` and `Runtime` objects was encapsulated in NEST's `Communicator` class.

NEST implements an error handling strategy based on C++ exceptions. Several new exception classes have been added to be used upon errors related to MUSIC. Unfortunately it is not possible to recover from errors during a MUSIC multi-simulation, as interactive simulations are not supported. Therefore NEST uses the function `MPI_Abort`() to quit the simulator upon errors. This also quits all remote applications.

Very little had to be changed in MOOSE to adapt it to MUSIC, and the changes here fit naturally into the MOOSE code structure. The five classes mentioned in Section "New Classes" were defined in fewer than 1000 lines of C++ code, and no changes were made in the basic MOOSE infrastructure. This was possible due to the compact MUSIC API, and was facilitated by the modular design of MOOSE.

Where possible, MOOSE allows the user to handle MUSIC related errors. For example, the user can inspect the `isConnected` field on MOOSE objects representing MUSIC ports, and choose to quit the simulation, or continue without MUSIC communication, in case a port was found to have not been connected successfully. At present, MPI exceptions are left unhandled by MOOSE, causing MOOSE to abort in case of errors at the MPI level.

While MUSIC supports communication of events, continuous values and messages, currently only spike event communication has been implemented in NEST and MOOSE.

The asymmetric multi-simulation benchmark (Section "Asymmetric Multi-simulation Benchmark", Fig. 10) shows that linking with MUSIC and using the MUSIC communicator does not affect performance. It also shows that normal communication loads through MUSIC ports do not add significantly to simulation time. In order to test performance under heavy communication load, the symmetric multi-simulation benchmark (Section "Symmetric Multi-simulation Benchmark", Fig. 11) provided a situation where every MUSIC channel in one application communicates spikes to neurons on every MPI process in the other application. MUSIC adapts both the spatial and temporal communication scheme to the topology of the multi-simulation, but the pilot implementation of the MUSIC library only uses pair-wise MPI `Send`() and `Receive`(). For a uni-directional one-to-one projection this would mean communication in one step at longer intervals. For the symmetric benchmark this instead implies a complete pair-wise exchange at every min-delay (minimum axonal delay between the applications). This partly accounts for the difference between no connectivity (black squares) and $N_{\mathrm{MUSIC}} = 8$ (dark grey diamonds) in Fig. 11a. However, the linear dependence on the number of spikes transmitted via MUSIC in Fig. 11b can also be attributed to the additional load due to the collection and delivery of spikes by `music_out_proxy`s and `music_in_proxy`s in NEST. While the pair-wise communication gives most efficiency for multi-simulations that do not require all-to-all communication, a future version of the library could switch to the use of, for example, `Allgather`() when the number of inter-process communication pairs are of O(#processes). Another interesting development would be to use non-blocking communication over the MUSIC library inter-communicators between `tick`() calls, during the time when the application is computing or communicating.

We conclude that MUSIC fulfills the design goal that it should be simple to adapt existing simulators to use MUSIC. In addition, since the MUSIC API enforces independence of the applications, the multi-simulation could be built from pluggable component modules without adaptation of the components to each other in terms of simulation time-step or topology of connections between the modules. Preliminary results from benchmarks of two reciprocally connected large-scale versions of the layered cortical network model (one magnitude larger than the model simulated in this article) also indicate good performance and scaling behavior. We would like to encourage the community to continue building on a sharable base of MUSIC-enabled simulators and tools for the easy construction of multi-simulations.

Acknowledgements Partially funded by DIP F1.2, BMBF Grant 01GQ0420 to the Bernstein Center for Computational Neuroscience Freiburg, EU Grants FP6-2004-IST-FETPI-015879 (FACETS) and FP7-HEALTH-2007-A-201716 (SELACT), the Next-Generation Supercomputer Project of MEXT (Japan), and the Helmholtz Alliance on Systems Biology (Germany). The MUSIC standard and software is provided and supported by the International Neuroinformatics Coordinating Facility (INCF).

References

Albus, J. S., Bekey, G. A., Holland, J. H., Kanwisher, N. G., Krichmar, J. L., Mishkin, M., et al. (2007). A proposal for a decade of the mind. *Science, 317*(5843), 1321.

Bower, J. M., & Beeman, D. (1998). *The book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2nd Ed.). New York: Springer.

Brette, R., Rudolph, M., Carnevale, N. T., Hines, M. L., Beeman, D., Bower, J. M., et al. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience, 23*, 349–398.

Cannon, R. C., Gewaltig, M.-O., Gleeson, P., Bhalla, U. S., Cornelis, H., Hines, M. L., et al. (2007). Interoperability of neuroscience modeling software: Current status and future directions. *Neuroinformatics, 5*(2), 127–138.

Carnevale, N. T., & Hines, M. L. (2006). *The NEURON Book*. U.K.: Cambridge University Press.

Crook, S., Gleeson, P., Howell, F., Svitak, J., & Silver, R. A. (2007). MorphML: Level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics, 5*, 96–104.

Crook, S. M., & Howell, F. W. (2007). XML for data representation and model specification in neuroscience. *Methods in Molecular Biology, 401*, 53–66.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics, 2*, 1–10.

Djurfeldt, M., Ekeberg, Ö., & Lansner, A. (2008a). Large-scale modeling—a tool for conquering the complexity of the brain. *Frontiers in Neuroinformatics, 2*, 1–4. doi:10.3389/neuro.11/001.2008.

Djurfeldt, M., & Lansner, A (2007). Workshop report: 1st INCF workshop on large-scale modeling of the nervous system. Nature Precedings. Available from http://dx.doi.org/10.1038/npre.2007.262.1.

Djurfeldt, M., Lundqvist, M., Johansson, C., Rehn, M., Ekeberg, Ö., & Lansner, A. (2008b). Brain-scale simulation of the neocortex on the BlueGene/L supercomputer. *IBM Journal of Research and Development, 52*, 31–42.

Ekeberg, Ö., & Djurfeldt, M. (2008). Music—multisimulation coordinator: Request for comments. Available from Nature Precedings http://dx.doi.org/10.1038/npre.2008.1830.1.

Ekeberg, Ö., & Djurfeldt, M. (2009). *MUSIC—Multi-Simulation Coordinator, users manual* (1st Ed.). Stockholm, Sweden: INCF, Karolinska Institutet, Nobels väg 15 A, SE-171 77, February 2009. http://software.incf.org/software/music.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., & Gewaltig, M. (2009). PyNEST: A convenient interface to the NEST simulator. *Frontiers in Neuroinformatics, 2*, 12. doi:10.3389/neuro.11.012.2008.

Gewaltig, M.-O., & Diesmann, M. (2007). NEST (Neural Simulation Tool). *Scholarpedia, 2*(4), 1430.

Hellgren Kotaleski, J., Plenz, D., & Blackwell, K. T. (2006). Using potassium currents to solve signal to noise problems in inhibitory feedforward networks of the striatum. *Journal of Neurophysiology, 95*(1), 331–341.

Hjorth, J., Zilberter, M., Oliveira, R. F., Blackwell, K. T., & Hellgren Kotaleski, J. (2008). Gabaergic control of backpropagating action potentials in striatal medium spiny neurons. *BMC Neuroscience, 9*(Suppl 1), P105. doi:10.1186/1471-2202-9-S1-P105.

King, J. G., Hines, M., Hill, S., Godman, P. H., Markram, H., & Schürmann, F. (2009). A component-based extension framework for large-scale parallel simulations in NEURON. *Frontiers in Neuroinformatics, 3*, 1–11.

Potjans, T. C., & Diesmann, M. (2008). Consistency of *in vitro* and *in vivo* connectivity estimates: Statistical assessment and application to cortical network modeling. In *Soc. Neurosci. Abstr.* (Vol. 38, pp. 16.1). Washington, DC, U.S.A.

Potjans, T. C., Fukai, T., & Diesmann, M. (2009). Implications of the specific cortical circuitry for the network dynamics of a layered cortical network model. *BMC Neuroscience, 10*(Suppl 1), P159.

Ray, S., & Bhalla, U. S. (2008). PyMOOSE: Interoperable scripting in python for MOOSE. *Frontiers in Neuroinformatics, 2*, 6. ISSN 1662-5196, URL: http://www.ncbi.nlm.nih.gov/pubmed/19129924, PMID: 19129924. doi:10.3389/neuro.11.006.2008.

Schemmel, J., Fieres, J., & Meier, K. (2008). Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008* (pp. 431–438).

Wolf, J. A., Moyer, J. T., Lazarewicz, M. T., Contreras, D., Benoit-Marand, M., O'Donnel, P., et al. (2005). NMDA/AMPA ratio impacts state transitions and entrainment to oscillations in computational model of the nucleus accumbens medium spiny projection neuron. *Journal of Neuroscience, 25*(40), 9080–9095.

Bibliography

Abeles, M. (1991). *Corticonics: Neural Circuits of the Cerebral Cortex* (1st ed.). Cambridge: Cambridge University Press.

Adobe Systems Inc. (1999). *Postscript language reference manual* (third ed.). Reading, MA: Addison-Wesley.

Aho, A. V., Sethi, R., & Ullman, J. D. (1988). *Compilers, principles, techniques, and tools*. Reading, Massachusetts: Addison-Wesley.

Albus, J. S., Bekey, G. A., Holland, J. H., Kanwisher, N. G., Krichmar, J. L., Mishkin, M., Modha, D. S., Raichle, M. E., Shepherd, G. M., & Tononi, G. (2007). A proposal for a decade of the mind. *Science 317*(5843), 1321.

Alexandrescu, A. (2001). *Modern C++ Design*. Boston: Addison-Wesley.

Barlow, H. B. (1972). Single units and sensation: a neuron doctrine for perceptual psychology? *Perception 1*, 371–394.

Bednar, J. A. (2009). Topographica: building and analyzing map-level simulations from Python, C/C++, MATLAB, NEST, or NEURON components. *Frontiers in Neuroinformatics*. doi:10.3389/neuro.11/008.2009.

Bi, G.-q., & Poo, M.-m. (1998a). Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *J. Neurosci. 18*, 10464–10472.

Bi, G. Q., & Poo, M. M. (1998b). Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *J Neurosci 18*, 10464–10472.

Binzegger, T., Douglas, R. J., & Martin, K. A. C. (2004). A quantitative map of the circuit of cat primary visual cortex. *J. Neurosci. 39*(24), 8441–8453.

Booch, G. (1996). *Managing the Object-Oriented Project*. Reading, Massachusetts: Addison Wesley Longman.

Bower, J. M., & Beeman, D. (1997). *The Book of GENESIS: Exploring realistic neural models with the GEneral NEural SImulation System* (2 ed.). New York: TELOS, Springer-Verlag-Verlag.

Braitenberg, V., & Schüz, A. (1998). *Cortex: Statistics and Geometry of Neuronal Connectivity* (second ed.). Berlin: Springer-Verlag-Verlag.

Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., & Yergeau, F. (2000). Extensible markup language (XML) 1.0. *W3C recommendation 6*.

Brette, R., & Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J Neurophysiol 94*(5), 3637–3642.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris Jr., F. C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., El Boustani, S., & Destexhe, A. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *J Comput Neurosci 23*(3), 349–398.

Brodmann, K. (1905). Beiträge zur histologischen Lokalisation der Grosshirnrinde. *J. Psychol. Neurol.*.

Brooks, F. P. (1995). *The Mythical Man Month. Essays on Software Engineering* (anniversary ed.). Addison Wesley Longman.

Brüderle, D., Grübl, A., Meier, K., Mueller, E., & Schemmel, J. (2007). A software framework for tuning the dynamics of neuromorphic silicon towards biology. In *Proceedings of the 2007 International Work-Conference on Artificial Neural Networks (IWANN'07)*, Volume LNCS 4507, pp. 479–486. Springer Verlag.

Brüderle, D., Müller, E., Davison, A., Muller, E., Schemmel, J., & Meier, K. (2008). Establishing a novel modeling tool: A Python-based interface for a neuromorphic hardware system. *Frontiers Neuroinf. this volume*.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci. 8*(3), 183–208.

Burks, A. W., Warren, D. W., & Wright, J. B. (1954). An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation 8*(46), 53–57. `http://www.jstor.org/stable/2001990`.

Cannon, R., Gewaltig, M., Gleeson, P., Bhalla, U., Cornelis, H., Hines, M., Howell, F., Muller, E., Stiles, J., Wils, S., , & De Schutter, E. (2007). Interoperability of neuroscience modeling software: current status and future directions. *Neuroinformatics 5*, 127–138.

Carnevale, N. T., & Hines, M. L. (2006). *The NEURON Book*. Cambridge University Press.

Casagnes, A., Moren, J., & Shibata, T. (2010). Integration of visuo-motor network models by music. In *Proceedings of the 2nd Biosupercomputer symposium: Toward integrative understanding of the live phenomena, current state and the future*. RIKEN.

Churchland, P. S., & Sejnowski, T. J. (1988). Perspectives on cognitive neuroscience. *Science 242*(4879), 741–745.

Connors, B. W., & Long, M. A. (2004). Alectrical synapses in the mammalian brain. *Annual Review of Neuroscience 27*(1), 393–418.

Crook, S., Beeman, D., Gleeson, P., & Howell, F. (2005). XML for model specification in neuroscience: An introduction and workshop summary. *Brains, Minds, and Media 1:bmm228*, (urn:nbn:de:0009–3–2282).

Crook, S., Gleeson, P., Howell, F., Svitak, J., & Silver, R. A. (2007). MorphML: level 1 of the NeuroML standards for neuronal morphology data and model specification. *Neuroinformatics 5*, 96–104.

Crook, S. M., & Howell, F. W. (2007). XML for data representation and model specification in neuroscience. *Methods Mol Biol 401*, 53–66.

Davison, A., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., & Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics 2*. doi:10.3389/neuro.11.011.2008.

Davison, A. P., Hines, M., & Muller, E. (2010). Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience*. doi:10.3389/neuro.01/036.2009.

Dayan, P., & Abbott, L. F. (2001). *Theoretical Neuroscience*. Cambridge: MIT Press.

De Schutter, E. (2008). Why are computational neuroscience and systems biology so separate? *PLoS Comput Biol 4*(5), e1000078. doi:10.1371/journal.pcbi.1000078.

Diesmann, M., & Gewaltig, M.-O. (2002). NEST: An environment for neural systems simulations. In T. Plesser & V. Macho (Eds.), *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, Volume 58 of *GWDG-Bericht*, pp. 43–70. Göttingen: Ges. für Wiss. Datenverarbeitung.

Diesmann, M., Gewaltig, M.-O., & Aertsen, A. (1995). SYNOD: an environment for neural systems simulations. Language interface and tutorial. Technical Report GC-AA-/95-3, Weizmann Institute of Science, The Grodetsky Center for Research of Higher Brain Functions, Israel.

Djurfeldt, M., Ekeberg, Ö., & Lansner, A. (2008). Large-scale modeling—a tool for conquering the complexity of the brain. *Front. Neuroinform. 2*(1).

Djurfeldt, M., Hjorth, J., Eppler, J. M., Dudani, N., Helias, M., Potjans, T. C., Bhalla, U. S., Diesmann, M., Kotaleski, J. H., & Ekeberg, Ö. (2010). Run-time interoperability between neuronal simulators based on the music framework. *Neuroinformatics 8*. doi:10.1007/s12021-010-9064-z.

Djurfeldt, M., Johansson, C., Ekeberg, Ö., Rehn, M., Lundqvist, M., & Lansner, A. (2005). Massively parallel simulation of brain-scale neuronal network models. Technical Report Technical Report TRITA-NA-P0513, KTH, School of Computer Science and Communication Stockholm, Stockholm.

Djurfeldt, M., & Lansner, A. (2007). Workshop report: 1st incf workshop on large-scale modeling of the nervous system. *Nature Precedings*. doi:10.1038/npre.2007.262.1.

Djurfeldt, M., Lundqvist, M., Johansson, C., Rehn, M., Ekeberg, Ö., & Lansner, A. (2008). Brain-scale simulation of the neocortex on the BlueGene/L supercomputer. *IBM J Research and Development 52*, 31–42.

Dubois, P. F. (2007). Guest editor's introduction: Python: Batteries included. *Computing in Science and Engineering 9*(3), 7–9.

Dudani, N., Ray, S., Siji, G., & Balla, U. S. (2009). Multiscale modeling and interoperability in MOOSE. In *Proceedings of the Eighteenth Annual Computational Neuroscience Meeting: CNS*2009*. Poster presentation.

Eccles, J., Fatt, P., & Koketsu, K. (1954). Cholinergic and inhibitory synapses in a pathway from motor-axon collaterals to motoneurones. *The Journal of Physiology 126*(3), 524.

Ekeberg, Ö., & Djurfeldt, M. (2008). MUSIC — Multisimulation Coordinator: Request For Comments. Available from Nature Precedings `http://dx.doi.org/10.1038/npre.2008.1830.1`.

Ekeberg, Ö., & Djurfeldt, M. (2009). *MUSIC — Multi-Simulation Coordinator, Users Manual* (1st ed.). Karolinska Institutet, Nobels väg 15 A, SE-171 77 Stockholm, Sweden: INCF. `http://software.incf.org/software/music`.

Encyclopædia Britannica (2007). *Encyclopaedia Britannica*. Chicago: Encyclopaedia Britannica.

Eppler, J. M. (2006). A multithreaded and distributed system for the simulation of large biological neural networks. Master's thesis, Albert Ludwig University Freiburg.

Eppler, J. M., Helias, M., Muller, E., Diesmann, M., & Gewaltig, M. (2009). PyNEST: a convenient interface to the NEST simulator. *Front. Neuroinform. 2*, 12.

Eppler, J. M., Kupper, R., Plesser, H. E., & Markus, D. (2009). A testsuite for a neural simulation engine. In *Proceedings of the 2nd INCF Congress of Neuroinformatics*. doi:10.3389/conf.neuro.11.2009.08.042.

Finkel, R. A. (1996). *Advanced programming languages*. Menlo Park, California: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesely.

Gerstner, W., & Kistler, W. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gewaltig, M.-O. (2009). Self-sustained activity in networks of integrate and fire neurons without external noise. In *Conference Abstract: Bernstein Conference on Computational Neuroscience*. Frontiers in Computational Neuroscience. doi:10.3389/conf.neuro.10.2009.14.051.

Gewaltig, M.-O., & Diesmann, M. (2007). NEST (Neural Simulation Tool). *Scholarpedia 2*(4), 1430.

Gleeson, P., Steuber, V., & Silver, R. A. (2007). neuroConstruct: A tool for modeling networks of neurons in 3D space. *Neuron 54*, 219–235.

Goodman, D., & Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics 2*. doi:10.3389/neuro.11.005.2008.

Gross, J., & Yellen, J. (1999). *Graph Theory and its Applications*. CRC Press.

Gütig, R., Aertsen, A., & Rotter, S. (2003). Analysis of higher-order neuronal interactions based on conditional inference. *Biol. Cybern. 88*(5), 352–359.

Hammarlund, P., & Ekeberg, O. (1998). Large neural network simulations on multiple hardware platforms. *J. Comput. Neurosci. 5*(4), 443–459.

Harrison, M., & McLennan, M. (1998). *Effective Tcl/Tk programming: writing better programs with Tcl and Tk*. Reading, Massachusetts: Addison-Wesley.

Helias, M., Rotter, S., Gewaltig, M., & Diesmann, M. (2008). Structural plasticity controlled by calcium based correlation detection. *Front. Comput. Neurosci. 2*(7), doi:10.3389/neuro.10.007.2008.

Helias, M., Rotter, S., Gewaltig, M.-O., & Diesmann, M. (2009). Self-sustained cell assemblies in structurally plastic networks. *8th Göttingen meeting of the Neuroscience Society*.

Hellgren Kotaleski, J., Plenz, D., & Blackwell, K. T. (2006). Using potassium currents to solve signal to noise problems in inhibitory feedforward networks of the striatum. *J Neurophysiol 95*(1), 331–341.

Hines, M., & Carnevale, N. T. (1997). The NEURON simulation environment. *Neural Comput. 9*, 1179–1209.

Hines, M., Davison, A. P., & Muller, E. (2009). Neuron and python. *Frontiers in Neuroinformatics 3*. doi:10.3389/neuro.11/001.2009.

Hjorth, J., Zilberter, M., Oliveira, R. F., Blackwell, K. T., & Hellgren Kotaleski, J. (2008). Gabaergic control of backpropagating action potentials in striatal medium spiny neurons. *BMC Neuroscience 9*(Suppl 1), P105. doi:10.1186/1471-2202-9-S1-P105.

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol. (Lond) 117*, 500–544.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. USA 79*, 2554–2558.

Hucka, M., Finney, A., Sauro, H. M., Bolouri, H., Doyle, J. C., Kitano, H., Arkin, A. P., Bornstein, B. J., Bray, D., Cornish-Bowden, A., Cuellar, A. A., Dronov, S. Gilles, E. D., Ginkel, M., Gor, V., Goryanin, I. I., Hunter, P. J., Juty, N. S., Kasberger, J. L., Kremling, A., Kummer, U., Le Novère, N., Loew, L. M., Lucio, D., Mendes, P., Minch, E., Mjolsnes,

E. D., Nakayama, Y., Nelson, M., Nielsen, P., Sakurada, T., Schaff, J. C., Shapiro, B. E., Shimizu, T. S., Spence, H. D., Stelling, J., Takahashi, K., Tomita, M., J., W., & Wang, J. (2002). The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics 19*(4), 524–531.

Izhikevich, E. (2003). Simple model of spiking neurons. *IEEE Transactions on neural networks 14*(6), 1569–1572.

Kandel, E. R., Schwartz, J. H., & Jessel, T. M. (2000). *Principles of Neural Science* (4 ed.). New York: McGraw-Hill. ISBN 978-0838577011.

Kerr, R., Bartol, T., Kaminsky, B., Dittrich, M., Chang, J., Baden, S., Sejnowski, T., & Stiles, J. (2008). Fast Monte Carlo simulation methods for biological reaction-diffusion systems in solution and on surfaces. *SIAM journal on scientific computing: a publication of the Society for Industrial and Applied Mathematics 30*(6), 3126.

King, J. G., Hines, M., Hill, S., Godman, P. H., Markram, H., & Schürmann, F. (2009). A component-based extension framework for large-scale parallel simulations in NEURON. *Frontiers in Neuroinformatics 3*, 1–11.

Kirsch, J. (2010). Skript zum kurs "das menschliche gehirn – ein mal- und bastelkurs". Technical report, Bernstein Center for Computational Neuroscience Freiburg.

Knuth, D. E. (1998). *The Art of Computer Programming* (Third ed.), Volume 2. Reading, MA: Addison-Wesley.

Kohonen, T. (1984). *Self-Organization and Associative Memory*. Springer-Verlag.

Kötter, R., Bednar, J. A., Davison, A., Diesmann, M., Gewaltig, M.-O., Hines, M., & Muller, E. (Eds.) (2009). *Frontiers in Neuroinformatics: Special topic on Python in Neuroscience*. Frontiers Research Foundation. `http://frontiersin.org/neuroinformatics/specialtopics/8`.

Kunkel, S., Potjans, T. C., Abigail, M., & Markus, D. (2009). Simulating macroscale brain circuits with microscale resolution. In *Proceedings of the 2nd INCF Congress of Neuroinformatics*. doi:10.3389/conf.neuro.11.2009.08.044.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 558–565.

Lazzaro, J. P., Wawrzynek, J., Mahowald., M., Sivilotti, M., & Gillespie, D. (1993). Silicon auditory processors as computer peripherals. *IEEE Transactions on Neural Networks 4*, 523–528.

Lendvai, B., Stern, E. A., Chen, B., & Svoboda, K. (2000). Experience-dependent plasticity of dendritic spines in the developing rat barrel cortex in vivo. *Nature 404*, 876–881.

Lewis, B., & Berg, D. J. (1997). *Multithreaded Programming With PThreads*. Upper Saddle River: Sun Microsystems Press.

Looie (2008). http://en.wikipedia.org/wiki/File:Vertebrate-brain-regions.png.

MacGregor, R. J. (1987). *Neural and Brain Modeling*. San Diego: Academic Press.

Markram, H., Lübke, J., Frotscher, M., & Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science 275*, 213–215.

Markram, H., Wang, Y., & Tsodyks, M. (1998). Differential signaling via the same axon of neocortical pyramidal neurons. *Proc. Natl. Acad. Sci. USA 95*(9), 5323–5328.

Martin, R. C., Riehle, D., & Buschmann, F. (Eds.) (1998). *Pattern languages of program design 3*. Reading, MA: Addison–Wesley.

MathWorks (2002). *MATLAB The Language of Technical Computing: Using MATLAB*. Natick, MA. 3 Apple Hill Drive, Natick, Mass. 01760-2098.

McConnell, S. (2004). *Code Complete: A practical handbook of software construction* (2nd ed.). Redmond, Washington, USA: Microsoft Press.

Message Passing Interface Forum (1994). MPI: A message-passing interface standard. Technical Report UT-CS-94-230.

Migliore, M., Cannia, C., Lytton, W. W., Markram, H., & Hines, M. (2006). Parallel network simulations with NEURON. *J Comp Neurosci 21*, 119–223.

Morrison, A., Aertsen, A., & Diesmann, M. (2006). Spike-timing dependent plasticity in balanced random networks. In *Computational Neuroscience Meeting*, Edinburgh, U.K., pp. 77.

Morrison, A., Aertsen, A., & Diesmann, M. (2007). Spike-timing dependent plasticity in balanced random networks. *Neural Comput. 19*, 1437–1467.

Morrison, A., Diesmann, M., & Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike-timing. *Biol. Cybern. 98*, 459–478.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., & Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput. 17*(8), 1776–1801.

Morrison, A., Straube, S., Plesser, H. E., & Diesmann, M. (2007). Exact subthreshold integration with continuous spike times in discrete time neural network simulations. *Neural Comput 19*, 47–79.

Muller, E., Kremkow, J., Davison, A., & Brette, R. (2009). Workshop on python in neuroscience at the 18th annual computational neuroscience meeting.

Natschläger, T. (2003). CSIM: A neural Circuit SIMulator. Technical report.

Nicholls, J. G., Martin, R. A., Wallace, B. G., & Fuchs, P. A. (2001). *From neuron to brain* (4 ed.). Sunderland, MA: Sinauer Associates.

Nordlie, E., Plesser, H. E., & Gewaltig, M.-O. (2009). Towards reproducible descriptions of neuronal network models. doi:10.3389/conf.neuro.11.2008.01.086.

Nowak, L. G., Azouz, R., Sanchez-Vives, M. V., Gray, C. M., & McCormick, D. A. (2003). Electrophysiological classes of cat primary visual cortical neurons in vivo as revealed by quantitative analyses. *J Neurophysiol 89*(3), 1541–1566.

Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Professional Computing. Reading Massachusetts: Addison-Wesley.

Pecevski, D., Natschläger, T., & Schuch, K. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with python. *Front. Neuroinform. 3*(11), doi:10.3389/neuro.11.011.2009.

Plesser, H. E. (2005). Nest express report. Technical report, Department of Mathematical Sciences and Technology, Norwegian University of Life Sciences.

Plesser, H. E., Eppler, J. M., Morrison, A., Diesmann, M., & Gewaltig, M.-O. (2007). Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In A.-M. Kermarrec, L. Bougé, & T. Priol (Eds.), *Euro-Par 2007: Parallel Processing*, Volume 4641 of *Lecture Notes in Computer Science*, Berlin, pp. 672–681. Springer-Verlag.

Potjans, T. C., & Diesmann, M. (2008a). Consistency of *in vitro* and *in vivo* connectivity estimates: statistical assessment and application to cortical network modeling. In *Soc. Neurosci. Abstr.*, Volume 38, Washington, DC, U.S.A., pp. 16.1.

Potjans, T. C., & Diesmann, M. (2008b). Integration of anatomical and physiological connectivity data sets for layered cortical network models. *BMC Neuroscience 9*(Suppl 1), P60.

Potjans, T. C., & Diesmann, M. (2009). An integrated data set for layered cortical network models: dynamical implications of specific connectivity. in preparation.

Potjans, W., Morrison, A., & Diesmann, M. (2009). A spiking neural network model of an actor-critic learning agent. *Neural Comput. 21*, 301–339.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *COMPUTER 33*, 23–29.

Rall, W. (1964). Theoretical significance of dendritic trees for neuronal input-output relations. In R. F. Reiss (Ed.), *Neural Theory and Modeling*. Palo Alto: Stanford University Press.

Ranson, S. W. (1920). *The Anatomy of the Nervous System: from the Standpoint of Development and Function*. W. B. Saunders.

Ray, S., & Bhalla, U. (2008). PyMOOSE: interoperable scripting in Python for MOOSE. *Frontiers Neuroinf. 2*, 6. doi:10.3389/neuro.11.006.2008.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organisation in the brain. *Psychological Review 65*, 386–408. Reprinted in Anderson:Neurocomputing1.

Schemmel, J., Brüderle, D., Meier, K., & Ostendorf, B. (2007). Modeling synaptic plasticity within networks of highly accelerated I&F neurons. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE Press.

Schemmel, J., Fieres, J., & Meier, K. (2008). Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008*, pp. 431–438.

Schmolesky, M. (2000). `http://webvision.umh.es/Webvision/VisualCortex.html`.

Schrader, S., Gewaltig, M. O., Körner, U., & Körner, E. (2009). Cortext: a columnar model of bottom-up and top-down processing in the neocortex. *Neural Networks 22*(8), 1055–1070. doi:10.1016/j.neunet.2009.07.021.

Shepherd, G. (1991). *Foundations of the neuron doctrine*. Oxford University Press.

Shepherd, G. M. (1988). *Neurobiology* (2 ed.). New York: Oxford University Press.

Stewart, T., Tripp, B., & Eliasmith, C. (2009). Python scripting in the Nengo simulator. *Frontiers in Neuroinformatics 3*. doi:10.3389/neuro.11.007.2009.

Stroustrup, B. (1997). *The C++ Programming Language* (3 ed.). New York: Addison-Wesely.

Stufflebeam, R. (2008). `http://www.mind.ilstu.edu/curriculum/neurons_intro/neurons_intro.php`.

Surachit (2007). `http://en.wikipedia.org/wiki/File:EmbryonicBrain.svg`.

Szentágothai, J. (1978). The Ferrier lecture, 1977: the neuron network of the cerebral cortex: a functional interpretation. *Proceedings of the Royal Society of London. Series B, Biological Sciences 201*(1144), 219–248.

Tam, A., & Wang, C. (2000). Efficient scheduling of complete exchange on clusters. In *13th International Conference on Parallel and Distributed Computing Systems (PDCS 2000)*, Las Vegas.

Tanenbaum, A. S. (1999). *Structured Computer Organization* (4 ed.). Upper Saddle River: Prentice Hall.

Thomson, A. M., & Bannister, P. (2003). Interlaminar connections in the neocortex. *Cereb. Cortex 13*, 5 – 14.

Thomson, A. M., & Deuchars, J. (1994). Temporal and spatial properties of local circuits in neocortex. *Trends Neurosci 17*, 119–126.

Trachtenberg, J. T., Chen, B. E., Knott, G. W., Feng, G., Sanes, J. R., Welker, E., & Svoboda, K. (2002). Long-term in vivo imaging of experience-dependent synaptic plasticity in adult cortex. *Nature 420*, 788–794.

Tsodyks, M., Pawelzik, K., & Markram, H. (1998). Neural networks with dynamic synapses. *Neural Comput. 10*, 821–835.

Tuckwell, H. C. (1988). *Introduction to Theoretical Neurobiology*, Volume 1, Chapter 3, The Lapique model of the nerve cell, pp. 85–123. Cambridge: Cambridge University Press.

van Rossum, G. (2008). Python/C API Reference Manual. `http://docs.python.org/api/api.html`.

Villarreal, M. R. (2007). `http://en.wikipedia.org/wiki/File:Complete_neuron_cell_diagram_en.svg`.

Vogels, T. P., & Abbott, L. F. (2005). Signal propagation and logic gating in networks of integrate-and-fire neurons. *J. Neurosci. 25*(46), 10786–10795.

Wall, L., Schwartz, R. L., & Potter, S. (1996). *Programming perl* (2nd ed.). Sebastopol CA, USA: O'Reilly & Associates, Inc.

Wils, S., & De Schutter, E. (2009). STEPS: modeling and simulating complex reaction-diffusion systems with python. *Frontiers in Neuroinformatics 3*. doi:10.3389/neuro.11/015.2009.

Wilson, G. (2006). Where's the real bottleneck in scientific computing? *American Scientist 94*, 5–6.

Wolf, J. A., Moyer, J. T., Lazarewicz, M. T., Contreras, D., Benoit-Marand, M., O'Donnel, P., & Finkel, L. H. (2005). NMDA/AMPA ratio impacts state transitions and entrainment to oscillations in computational model of the nucleus accumbens medium spiny projection neuron. *J Neurosci 25*(40), 9080–9095.

Wolfram, S. (2003). *The Mathematica Book* (5 ed.). Wolfram Media Incorporated.

Woll, A. (2007). Performance analysis of an MPI- and thread-parallel neural network simulator. Master's thesis, Norwegian University of Life Sciences.

Zeigler, B. P., Praehofer, H., & Kim, T. G. (2000). *Theory of Modeling and Simulation* (Second ed.). Amsterdam: Academic Press.