
NEST: An Environment for Neural Systems Simulations

Markus Diesmann
Dept. of Nonlinear Dynamics,
Max-Planck Inst. für Strömungsforschung, Göttingen

Marc-Oliver Gewaltig
Future Technology Research,
Honda R&D Europe (Deutschland) GmbH, Offenbach

Abstract

NEST is a framework for simulating large, structured neuronal systems. It is designed to investigate the functional behavior of neuronal systems in the context of their anatomical, morphological, and electrophysiological properties. NEST aims at large networks, while maintaining an appropriate degree of biological detail. This is achieved by combining a broad range of abstraction levels in a single network simulation. Great biological detail is then maintained only at the points of interest, while the rest of the system can be modeled by more abstract components. Here, we describe the conception of NEST and illustrate its key features. We demonstrate that software design and organizational aspects were of equal importance for the success of the project.

1 Introduction

Neuroscience is one of the fields in life science which have recently received increased attention as can be seen from the fact that the 1990's were proclaimed the *Decade of the Brain* by the US Government.

Because of the immense difficulties involved in observing neuronal activity in an intact brain, experimental data exists mostly for single or small numbers of neurons, or for very large populations ($\gg 10^6$ neurons). As a consequence, scientists increasingly employ theoretical and computational methods in order to assess the dynamical properties of individual and populations of neurons and to come up with testable predictions.

Computational neuroscience is, thus, a fast growing research field, dedicated to the investigation of the nervous system with the help of computer simulations. It is important to note, however, that the type of neuronal network model discussed here differs considerably from so-called artificial neural networks (ANN), which have become an important branch of engineering and computer science.

In Computational Neuroscience, simulations are used to investigate models of the nervous system at functional or process levels. Consequently, a lot of effort has been put into developing appropriate simulation tools and techniques, and a plethora of simulation software, specialized for the single neuron or small sized networks is available (e.g. [8, 19]).

Recently, however, there has been growing interest in large scale simulations, involving some 10^4 neurons while maintaining an acceptable degree of biological detail. Thus, there is need for simulation software, possibly parallel, which supports such simulations in a flexible way.

Here, we describe the Neural Simulation Technology (NEST) initiative, a collaborative effort to develop an open simulation framework for biologically realistic neuronal networks. The system is distinguished by at least three features. First, it aims at large structured networks of heterogeneous, biologically realistic elements at different description levels. Second, NEST employs an iterative incremental development strategy, maintaining a running system at any time (see e.g. [9, 7]). Third, the software is developed as a collaborative effort of several research groups. Since NEST is a research tool, it has to continuously adapt to the ever changing demands of the researchers who are using it. Thus, the design and development process of the software is aware of the environment in which the system has to be implemented. The mechanisms supporting the collaboration are an integral part of NEST.

The system has been continuously developed and successfully applied in research over the last few years [11].

2 Basic concepts

In this section, we introduce the conceptual framework on which NEST is based.

A computer simulation is the attempt to investigate the properties of a

(complex) physical system by evaluating the behavior of a simplified model of the system on a computer. We will call such models *computer models*. In contrast to the analytical or numerical examination of mathematical models, computer models often contain algorithmic components. These make a mathematical treatment in a closed form at least difficult, if not infeasible.

A NEST simulation tries to maintain a close correspondence to an electrophysiological experiment. There are two conceptual domains which have to be mapped to the software level:

1. the description of the neuronal system, that is the network components and the network,
2. the description of the experimental setup and protocol.

2.1 *Neuronal systems*

It is a well established idea that information processing in the central nervous system relies on the electric discharges (*action potentials* or *spikes*) of a certain class of cells, the *neurons*, which interact at specialized points of contacts, the *synapses*. A generated spike arrives at one or more target neurons after a delay of a few milliseconds and causes a small change in the neurons' membrane potentials (so-called *post-synaptic potential*).

It is often assumed that if the superposition of sufficiently many post-synaptic potentials at a target neuron reaches a threshold value, the cell will itself generate a spike, however, details vary considerably between different neuron models. More models agree on the concept that the time-course of an action potential can be neglected, and thus the interaction between neurons can be described by the exchange of *point events*.

The standard approach to neural modeling is to consider neurons as the basic network components and describe the network in terms of the neurons, their positions and their projections. This approach can be called bottom up, since the network is built from its basic constituents upwards.

However, the brain is a heterogeneously structured system, composed of numerous areas and nuclei which are distinguished by their anatomical, physiological, and functional properties. Moreover, each of these parts has a specific architecture which is to a large extent genetically determined. The structure of the brain, therefore, conveys information beyond the mere connectivity (i.e. synaptic connections) of its constituents.

Thus, NEST adopts a top-down approach. Networks are regarded as hierarchical structures which may be represented by trees. A network is described in terms of abstract components which may either be atomic or compound.

Atomic network elements may range from sub-neuronal compartments to large populations of neurons, and may, therefore, differ considerably in complexity and degree of abstraction.

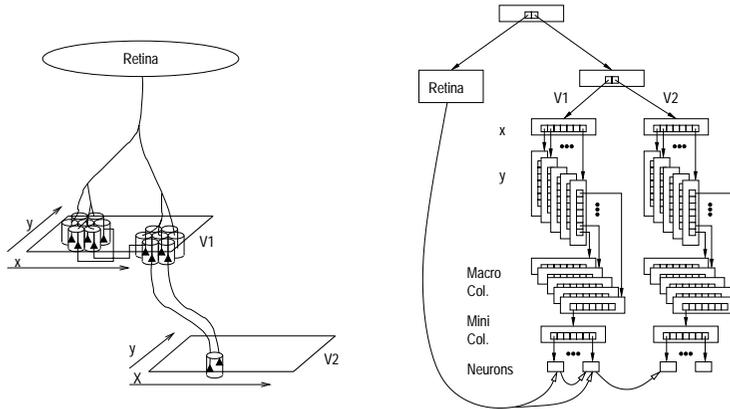


Fig. 1: Example of a structured network model (left) and how it is represented (right).

Compound network elements are defined in terms of other elements and their mutual connections. They may be nested and may, thus, be used to define concepts like areas, macro-columns, and mini-columns. Structural model concepts can therefore be mapped to the simulated network.

Fig. 1 (left) shows an example of a hierarchically structured model (see e.g. [22]). The model network consists of several sub-structures, described at different levels of abstraction: a *retina* and two model brain areas, *V1* and *V2*. The retina sends input to a first visual area *V1* which consists of topographically organized macro-columns. These, in turn, consist of orientation columns. Finally, orientation columns consist of model neurons. The other area, *V2*, has a similar structure.

The right panel shows how the model may be represented in the framework presented here. The tree reflects the *semantic structure* of the model. The *synaptic connectivity* of the network is not shown. Atomic network elements are represented by separate objects as *leaves* of the network tree. Examples are the individual neurons of *V1* and *V2*, or the model retina which represents a large population of neurons in terms of a single atomic element. The model areas *V1* and *V2* and the various columns are examples for compound elements.

2.2 Virtual experiments

In an electrophysiological experiment, a number of different devices are used to stimulate and observe the neuronal system. In our approach, the measurement and stimulation process is an explicit part of the model. Accordingly,

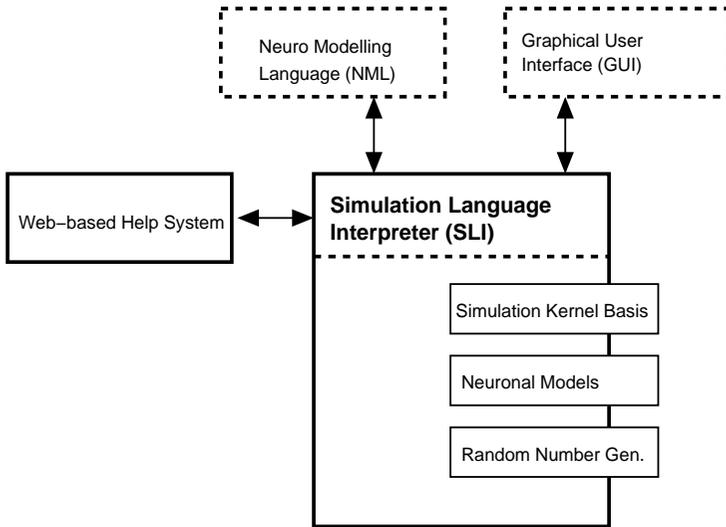


Fig. 2: Structure of the simulation system. Its main parts are a simulation kernel, simulation language interpreter (SLI) and some auxiliary modules. The simulation language interpreter integrates all parts and acts as interface to the user. Additional modules extend functionality.

devices are explicitly mapped to the simulation in order to arrive at *virtual experiments*.

3 Structure of the software

At the top level, NEST consists of a number of independent modules (Fig. 2), with each module contributing some piece of functionality. NEST is written in an object-oriented style (C++). An abstract base class defines a core interface which has to be implemented by each module.

The various modules are combined by a module loader, which is part of a simulation language interpreter (SLI). The interpreter is not only the primary interface to the user, but also provides a set of basic data structures which are used to exchange data between modules.

The simulation language interpreter is a simple stack machine, optimized for the fast processing of high-level commands and objects. It is discussed in detail in section 6.

The second important component of NEST is the simulation kernel. It implements all functionalities to define and simulate large structured networks of biologically realistic neural networks. It provides

1. base classes for neuronal models,
2. derived implementations of important models,
3. a network driver class to manage the temporal update of all simulation elements,
4. a SLI module to provide high-level access to the kernel, and
5. a SLI module to provide a high-level interface to the available models.

Items 1 through 3 are combined in a single C++ library which may also be used without the simulation language interpreter. The simulation kernel is discussed in detail in section 5.

In addition to the simulation kernel, there are a number of other modules which provide important functionality. Some modules solve well-defined mathematical problems like differential equations, some perform high-level operations on arrays and lists or provide random number generators. Other modules implement interfaces with the host operating system or on-line help facilities.

The modular architecture of NEST is an essential ingredient to the aforementioned incremental/iterative development model.

4 Structure of the project

Apart from the technical consideration of the last section, the structure of the project is strongly shaped by the context in which the software needs to be developed and maintained. Important constraints come from the human resources available to the project, the expected life time of individual components, conditions for software development in a scientific environment, and funding.

4.1 *Constraints*

Software which is used as a research tool usually never reaches a steady state. Scientific progress constantly opens up new and unexpected questions which may require improvement and probably partial redesign of software tools. Sometimes a new generation of hardware enables the investigation of a new class of problems (see section 11). In any case, the research tool should be flexible enough to be easily adapted to the new demands of the researcher.

The long time scale of software projects like NEST usually conflicts with two other time scales. First, researchers stay in a laboratory for only 2-5 years. This encourages researchers to write software specifically for their immediate purpose at the expense of reusability. Thus, this software and know-how is lost to the group when the researcher leaves. Second, the time scale on which a particular hardware platform becomes obsolete has dropped to a few years.

The same is true for parts of the software environment, like graphical user interfaces or even operating systems.

4.2 Consequences

These considerations shaped the structure of our simulation tool from the very beginning and resulted in a number of design decisions:

Platform independence The software should not rely on specific hardware or a specific software manufacturer. We achieve this by using open standards where possible, never relying on proprietary software components. In addition the GNU developer tools [32] enable us to design the compile/build process in a platform independent way.

Implementation language An object oriented programming language is used to allow a clear separation of concepts. It helps to hide implementation details, to enhance reusability of code, and helps new users to understand existing code.

Although at the time the project was started (1994) there was no finished standard available, we chose C++ as implementation language. The combination of object oriented programming and static type checking turned out to be an ideal choice for simulation software where the time needed by a simulation run severely limits the feasible research topics.

Cascaded interfaces Changes and extensions of the software are routinely required. To minimize the effort needed to arrive at the desired functionality, NEST provides a cascade of interfaces. The idea is that simple improvements should be simple to implement with little internal knowledge. However, substantial improvements are also possible. Both are supported by programming interfaces at the level of the simulation kernel and the interpreter.

Collaborative effort The large investment required to develop the infrastructure of NEST led us to the conclusion that simulation software should be developed in collaboration with several researchers with the additional benefit that the different interests of researchers automatically tests the generality of new concepts. Moreover, it reduces the risk that the software becomes unusable when one researcher leaves the team. Fig. 3 indicates the research groups which currently participate in the development of NEST.

Open Source All software tools on which NEST depends are freely available under the GNU general public license (GPL). An important insight from

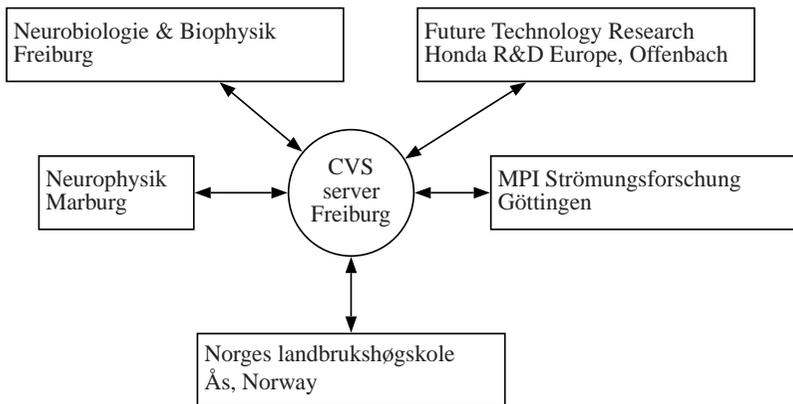


Fig. 3: Distributed development using a common source base. Boxes indicate the locations at which the software is currently developed and used. The circle indicates a server from which the members of the project can update their sources at any time and submit their changes and additions. The system performs automatic version control and detects conflicts between changes made by different users (CVS, [6]).

Open Source projects is that their characteristic rapid speed of progress and the high quality of the software are achieved by developing the software in a distributed team of developers. Thus, distributions of NEST will also be released under the GPL.

Iterative/Incremental development This development model is supported by two principles. First, each researcher implements those parts which he needs for his research. Second, every change and every contribution is immediately submitted to all others. Because each change made by one developer becomes incorporated in the simulation software used by others, errors and unwanted side effects are quickly detected and resolved. Moreover recompilation in different software environments and with different compilers rapidly uncovers non-standard or insecure code.

If one change breaks the simulation project of others and it is not easily possible to resolve the problem, it is always possible to revert to an older version. Thus, the rapid spread of new code between different sites does not affect usability of the software in (scientific) production.

We use the concurrent version system CVS to coordinate this process (see Fig. 3). Using CVS effectively initiates a continuous *review process* which improves the quality of the code and reduces the time taken for errors to be detected and removed. The motivating effect of this policy cannot be overestimated. Each developer is motivated to produce high quality code because he is aware that his contribution is immediately reviewed by others. In our

experience most problems are solved within 1 to 2 days by the exchange of a e-mails and cooperative work on the sources.

Documentation Because of the distributed development and the growing number of developers, an efficient mechanism is required to write and distribute documentation. The measures we have taken are described in section 7.

5 The simulation kernel

5.1 Overview

NEST uses an object-oriented approach. The system to be simulated is conceptually broken down into *components* which are then represented by *classes* at the C++ level. As outlined in section 2, neuronal systems are composed of objects like neurons, synapses, columnar circuits, or whole cortical areas. In a NEST simulation, all these components are called *elements*. Elements share a common interface, but may differ considerably in their purpose, functionality, or their level of abstraction. Different element types may co-exist in the same network. This design also supports the implementation of element types at different levels of abstraction, ranging from sub-neuronal components (e.g. compartments) to supra-neuronal elements (e.g. populations of neurons).

A simulation is then built from the set of pre-defined components which are dynamically (during runtime) combined and configured to represent the neuronal model network. While the components are hard coded in C++, their configuration is usually not. Configuration generally takes place at the level of the simulation language SLI.

The simulation kernel has two main parts. The first part is a set of abstract base classes which provide the starting points for neuronal elements at different abstraction levels. The second part is the simulation driver, the administrative center of the simulation kernel.

5.2 Simulation driver

The simulation driver has four main tasks:

1. administration of the network structure,
2. administration of network elements,
3. organizing the temporal update of each element, and
4. orchestrating the communication between elements.

Networks and elements As mentioned above, network elements fall into two categories: atomic elements and compound elements. The most important compound element is the sub-network. It is used to group arbitrary elements. Since sub-networks may be nested, a network can be thought of as a tree structure with one root sub-network at its base.

The concept of nest-able sub-networks allows the modeler to group functionally related neurons into circuits which may themselves be grouped into higher order circuits. This conceptual grouping introduces a hierarchical structure to the network.

The simulation driver controls the root of this network tree. To the user the network tree is comparable to a directory tree in a file system: sub-networks correspond to *directories* and atomic elements to *files*. The driver provides functions to operate on the network tree similar to the UNIX commands `cd`, `ls`, and `mkdir`, with the difference that network elements are numbered, rather than named. If the user enters a sub-network, the driver will perform any subsequent operation local to this sub-network. If the user creates an element, it will be placed in the current sub-network. Note that GENESIS [8] uses this approach to navigate through the compartments of a model neuron.

Element update After the neuronal model system is configured, the driver is responsible for running the simulation for a specified amount of simulation time. It also controls the global clock of the simulation.

Simulation time proceeds on an evenly spaced grid t_0, t_1, \dots, t_n with

$$t_i := t_0 + i \cdot h$$

where h is the *simulation step size*.

During each time slice, the state of the network is updated by updating each network element. The actual computation performed during update is encapsulated in the respective element class. Different elements may possess very different internal dynamics and may still live in one network. All the driver has to know is that all elements have a common interface which is used during update.

Thus, during each time-slice, the driver iterates over all network elements and calls a specific member function which updates the element's state. After all elements have been updated, the central clock is advanced. This is repeated until the desired simulation time has elapsed.

5.3 Network elements

Computational neuroscience is faced with the problem that there is no standard model neuron; nor is there agreement at which scale neuronal systems

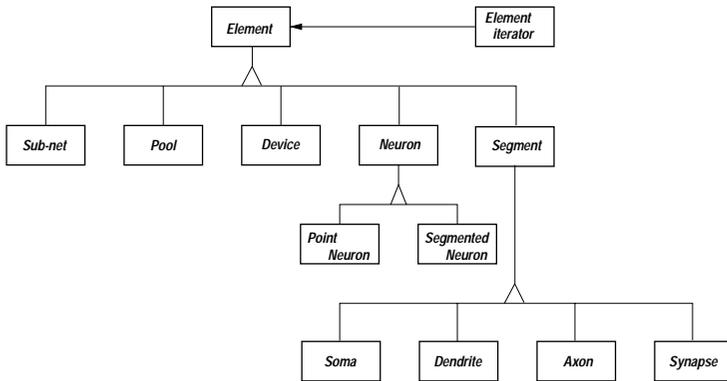


Fig. 4: Base class hierarchy for network elements. All elements are derived from a common base class, the abstract network element. Derived elements add new properties, and inherit the properties of their ancestors. There are elements at the sub- and supra-neuronal level. The distinction between point neurons and segmented neurons maintains the possibility of efficiently simulating “simple” neuron models. Measurement and manipulation devices inherit from the common base class `Device`. See [16] for notation.

should be investigated. This is reflected in the class hierarchy of network elements which are currently used in NEST.

Element hierarchy The range of possible abstractions is covered by deriving several abstract classes from the element base class, each class representing a different range of possible abstractions. Figure 4 shows a simplified excerpt of the element hierarchy.

Configuration interface The element base class defines the minimal interface that each element has to implement. However, neuron models have different parameters and it is not trivial to define a generic interface which is applicable to all implemented neurons and which can be used by the interpreter to access an element’s internal parameters.

One option is that for each element there exists a separate function, taking exactly those parameters required by the element. This leaves the task of remembering the number of parameters and what is worse the correct ordering of the parameters to the user. Moreover, specific functions have to be added to the interpreter whenever a new model class is introduced. Another option is to provide access to all thinkable parameters by separate member functions of base class for network element introducing all the problems and ugliness of a “fat interface” (see discussion in [30]).

The solution to the problem is to use named or keyword parameters [14].

This also opens the possibility to use meaningful names like “Resistance” or “Capacity” for the parameters. This technique is used by a number of interpreted languages where functions can have a large number of arguments, many of them typically having default values. In order to implement a named-parameter interface, the simulation kernel uses the dictionaries (section 6.1) of the SLI interpreter.

Now an element only has to provide two interface functions: one function which takes a dictionary as argument and changes the element’s parameters accordingly; and one function which returns a dictionary with the element’s current parameter settings.

From the interpreter, a single command can extract a dictionary from an element. The user can now operate on this dictionary. A second SLI command transmits the dictionary back to the network object.

5.4 Communication

Elements in a network are able to exchange information. While simple neurons may just exchange point events (spikes), more powerful element types also need to exchange more complex information. NEST provides a generic event mechanism which allows elements to exchange arbitrarily complex objects on demand. In order to ensure causality, all interactions must have a minimal delay of one simulation step.

A connection between two neurons usually involves a synapse. A synapse determines the biophysical properties of a connection. For example, it determines how strongly an incoming spike affects the target neuron.

Spike events As described above, a large class of neuron models communicate by exchanging spikes. During update, a neuron model combines all incoming spikes and determines whether it should itself generate a spike. When a spike is emitted, it is simultaneously sent to all target neurons. The emission of spikes by a single neuron is sparse (in each time slice the probability of spike generation is $\ll 1$). Therefore it is efficient to place the dynamics of synapses on the post-synaptic side of the connection and to let the pre-synaptic neuron notify the post-synaptic site whenever a pre-synaptic spike occurs.

Time driven vs. event driven simulation The fact that in a NEST simulation, neurons communicate by discrete events appears to be ideally suited to a paradigm which is called *event driven simulation* [15]. In such a paradigm, all events carry a stamp with the time when the event should arrive at the target. A scheduler collects all events and queues them until they have to be

delivered. Consequently, it is possible to drive the simulation time according to the time-stamps of the delivered messages. If all messages with stamps $t \leq T$ have been handled, the simulation time may be advanced to time T .

In contexts where the evaluation of each element is expensive compared to the costs of communication, e.g. if each element sends its messages to only a few targets, event driven strategies offer great potential for optimizations.

However, neuronal networks usually consist of a very large number of relatively simple elements with high connectivity (about 10^4 targets per neuron), where each spike event is sent to all targets. In this case an event driven strategy leads to an immense overhead, because a single spike results in some 10,000 event objects which have to be allocated, queued, sorted, delivered and deallocated. Thus, a large number of undelivered events accumulate in the queue.

By contrast, a time driven strategy is useful if the communication is expensive compared to the update of the element. In this case, it is possible to deliver events as they are sent and queue the information where there is enough information to perform optimizations: at the target. Moreover, as the event information is still available at the sender, it is also possible to send an event reference rather than objects. Thus, unnecessary replication of events is not needed.

6 The simulation language interpreter

The simulation language has five basic tasks:

1. to describe the structure of the neural system and the setup of the required manipulation and probing devices,
2. to describe the protocol of the virtual experiment,
3. to provide support for pre- and post-processing of data,
4. to provide a safe user interface with the simulation kernel for interactive sessions, and
5. to provide the interface between kernel and external programs.

This section is concerned with the general properties of the interpreter. An example for a complete simulation script can be found in section 8. We decided to implement a stack machine [5] which interprets a language very similar to PostScript [4]. Such a language is easy to implement and allows us to manipulate high-level objects. At the same time the language is fast and simple enough to serve as the target language for machine generated code. Thus, the simulation language interpreter (SLI) can be used as a virtual machine (section 3). The interpreter required to execute PostScript and how it operates is described in the excellent “Red Book” [4]. Closely following the literature for the implementation of the interpreter has minimized our development

time and reduced the risk of design errors. Several extensions with respect to the PostScript machine were needed to meet our requirements. The most relevant extensions are

- the sizes of arrays, procedures, and strings are dynamic. At any time appending, inserting, or deleting elements can alter the size of a container object. Preallocating memory to increase the performance, e.g. if the size of an array is successively increased by appending elements, is possible.
- optional type checking and operator overloading.

Whereas the basic commands for manipulation of the stacks are the same in SLI and in PostScript, obviously a different set of commands is required for neural simulations than for the description of graphics. The commands introduced to control the simulation kernel are mainly constructors for different network elements, commands for navigating in the network and commands for inspecting and configuring network elements.

Although PostScript provides some commands to manipulate arrays, a much richer set of commands was desired for setting up neural simulations and data pre- and post-processing. The high-level mathematics programming language Mathematica [34] has a consistent set of commands for data manipulation. The style of Mathematica operations fits well to the capabilities of our stack machine. The combination of heterogeneous arrays as universal containers and pure (or anonymous) functions [14] is present in Mathematica as well as in SLI and promotes the definition of powerful operators and a functional programming style. We therefore designed the commands for data manipulation according to the Mathematica model. Doing so, we could again reduce the development time and the risk of design flaws. Many of the current users have several years of experience with Mathematica and can therefore start to work with SLI without having to learn new commands for the same thing. It is one of the principles of NEST that, if later generalization is not severely impaired, basically only those features should be implemented which are immediately used. In particular, all mathematical operations are carried out numerically, although the stack machine itself fully supports the manipulation of symbols.

6.1 *Language properties*

In the scope of the present paper it is neither necessary nor possible to discuss the capabilities of SLI in detail. However, to illustrate the resulting language we briefly summarize a few useful properties and commands

Heterogeneous arrays In particular an array element can itself be an array of arbitrary length. Thus multidimensional data structures of arbitrary shape are supported (with lazy evaluation, copy-on-write).

Dictionaries A data type called dictionary represents a special form of an associative array [14, 20]; array elements are accessed by symbols. Dictionaries have multiple uses in SLI. In fact, similar to PostScript, in SLI all variables and named functions are managed by dictionaries (a named function is nothing else but a pure function assigned to a symbol). SLI dictionaries do not take strings, but symbols as keys for dictionary lookup. When a SLI expression is parsed, the terminal (character) representation of a symbol is transformed into an efficient internal representation which allows fast dictionary lookup. Dictionaries are also used to communicate with the simulation kernel.

Functions as first class values As in some other interpreted languages, it is possible to assemble an expression from a string at runtime and execute it by invoking the full scanner, parser, stack machine sequence. However, in SLI a much more efficient alternative exists. Similar to some other languages (e.g. LISP, see [3] and references therein) SLI procedures are first class values [14]. Therefore it is possible to assemble a procedure with the same efficiency and with the same operators as one assembles an array.

Error handling SLI supports exception handling [14] which allows the user to protect a certain piece of code by a construct similar to the try-catch block of C++. If an exception occurs, the execution stack (call stack) is appropriately unwinded and the innermost exception handler is called. This allows the developer of a simulation to handle errors at locations in the program where sufficient information is still available, and cleanly separate error handling code from the code describing the simulation. The SLI interpreter itself uses the exception mechanism such that application code can be well integrated with existing code implemented in SLI or C++. In normal operation, the interpreter executes code protected by a default exception handler that prints some diagnostics for otherwise uncaught exceptions.

6.2 *Levels of extendability*

The incremental development strategy (section 4) puts strong demands on the extendability of the interpreter. There are three different levels upon which functionality can be added to the interpreter. At each level the developer is presented with an interface which hides as much of the interpreter machinery as possible and requires only localized changes. The top level interface is the SLI language itself. New functions can be defined in individual or collective text files and loaded or discarded as needed. In fact, a large part of the interpreter's functionality is implemented in SLI itself and loaded at start

up. Only a small set of primitive operations is required to drive the stack machine. On the second level, the developer can define new “micro programs” for the stack machine. Operators influencing program control like `loop`, or the operator actually executing a procedure, are typically not implemented by a single C++ function. These micro programs have full access to the execution stack. This is not allowed for SLI programs in order to enforce separation of levels of abstraction. Providing support for writing micro programs allows the developer to implement new control structures efficiently. The third level is the straight implementation of an operator on the C++ level. Important examples are the operators interfacing with the simulation kernel. Arguments need to be taken from the operand stack and appropriately passed to a member function of a kernel object. The interpreter provides an interface that gives the developer access to the stacks and the framework for handling error conditions. The ordering in the three levels does not necessarily reflect the difficulty of implementation (from low to high), but rather the amount of available SLI functionality that can be used (from high to low). The level of micro programs may be considered to be the most difficult. It is a common strategy to implement a new function in SLI (level 1) first. Only when serious performance problems are detected is the function re-implemented in C++ (level 3).

6.3 Implementation issues

We have shown in section 5 that choosing C++ as the implementation language for the simulation kernel proved to be very useful for a number of reasons. Although most of the properties of C++ are also advantageous for the implementation of the interpreter, one aspect constituted a serious challenge. The strength of SLI’s machinery is the heterogeneous nature of its stacks and arrays. This is in stark contrast to the philosophy of static typing in C++ which is so useful in generating high performance code. Considerable design efforts were required to arrive at a solution which is at the same time computationally efficient and comfortable for the developer. In this area of code virtual member functions and templates are extensively used.

6.4 The Language is the protocol

We would like to end the section on the simulation language interpreter with a remark found in [18]. Considering the communication between a process performing some computation and a process providing the GUI, one is tempted to think that some format needs to be defined for the data and requests to be exchanged. Even worse, in this case a considerable amount of code on both sides needs to be devoted to the interpretation of the format. However,

there is a much more elegant solution. If the programs on both ends of the communication channels are executed by interpreters, the sender can simply send its data in the language of the receiver. We have successfully tested this approach using Tcl/Tk [25, 33] as a graphics engine for SLI. The GUI sends requests as SLI statements and the simulation engine responds with Tcl/Tk statements.

7 Documentation

The first documentation appeared as a technical report of the Weizmann Institute of Science in 1995 [11] consisting of some 60 pages. While the basic principles of the simulator remained practically unchanged in the following years, the number of individual commands and their scope has grown swiftly.

Different types of documentation are required for the user and the developer. Both types consist of introductory parts describing concepts of usage and design respectively. In addition, both need a reference part describing individual functionality in the first case, and its implementation in the second. Whereas the introductory parts only change slowly over the life time of the project, the reference parts are subject to much more rapid extension and change. In the NEST project, all documentation is integrated in a web-based (hyper-linked) help-desk. In the present section we focus on the mechanism we developed to generate and maintain the user-level reference documentation. We return to the documentation of source code in section 11.

Encouraging experience was gathered with the style of documentation in Matlab [24], a tool for numerical data analysis and visualization. The startup message of SLI informs the user of the availability of a command, which contacts the web-browser and displays the introductory help-desk page of the documentation. The user has access to an index of SLI functions organized by name, and an index organized by subject. Hyper-links lead to the documentation of individual functions. The documentation page of an individual function, in turn, may provide links to the documentation of related functions. Alternatively, the documentation for function f can be viewed on the console by issuing `/f help`. On an X-terminal, `help` can be configured to present the documentation in a separate window using an application like `xless`.

We observed that functions were routinely commented. Here, the comments in the code primarily served as specification and reminder for the author. In contrast, developing a separate user documentation turned out to be a slow process. The technique used by Matlab overcomes this problem by generating documentation from the source code. If the user issues the command to ask for documentation of a specific function f , the file `f.m` is searched for the first comment, and this comment is returned as documentation of f .

Exploring this technique, we found that the combination of two aspects is essential in motivating developers to write and maintain documentation. First, documentation is written directly in the source code. Second, after typing a single command at the interpreter prompt the new text is directly visible, integrated in the online help.

In our project, functions can be defined in C++ or in SLI. In contrast to Matlab, several functions can be defined in the same file. A mechanism for automated documentation should work homogeneously for both languages. Because of the rapid development cycle, the documentation also contains information about the location of the source file. In addition to the PostScript (and Matlab) comment sign % which protects a single line of comment, SLI also uses the C/C++ style comment block /* ... */ (however, see [29]). This allows us to specify a homogeneous documentation block in both languages which should precede any definition of a SLI operator, implemented in C++ or SLI:

```
/*BeginDocumentation
Name: Flatten - flattens a nested list
Synopsis:
        array          Flatten -> array
        array integer Flatten -> array
Description:
Flatten called with one argument flattens out all
levels of the argument. Flatten called with two
arguments flattens out the first n levels.
Examples:
    [3 [4 [5 [6]]] 7] Flatten --> [3 4 5 6 7]
    [3 [4 [5 [6]]] 7] 1 Flatten --> [3 4 [5 [6]] 7]
    [3 [4 [5 [6]]] 7] 2 Flatten --> [3 4 5 [6] 7]
Author: Gewaltig, Diesmann
...
SeeAlso: Partition
*/
```

The documentation block starts with `BeginDocumentation` to distinguish it from other comments in the file. A set of keywords (e.g. `Name` and `SeeAlso`) structures the documentation.

Our documentation generator works as follows. The command

```
(filename) makehelp
```

extracts the documentation comments of all functions defined in the file and writes them as separate files into a special directory. Information about the name and the location of the source file is appended to the documentation file. In the next step the file is converted to an HTML file. The function

names following keyword `SeeAlso` are used to create hyper-links. Command `makeallhelp` searches the source tree of NEST and calls `makehelp` for all appropriate source files. Subsequently, `makeallhelp` generates an index of all the functions. Similar to the “H1” line of Matlab, the text in the line where keyword `Name` appears following the minus sign is used as a short description of the function in the index.

The index file is then converted to an HTML file, with function names representing hyper-links to the documentation files. HTML files are designed to integrate with the remainder of the HTML documentation of the simulator (appropriate links). When the simulator is built from the sources, in the last step of the make process, the interpreter is started to execute `makeallhelp`. Thus, consistency of the documentation with the sources is automatically enforced. A list of SLI functions ordered by subject is currently maintained manually. It should be noted that the documentation engine is fully implemented in the SLI language.

Since the mechanism described in this section has been in operation the documentation has grown steadily and new functions have practically always been commented. The clearness of the HTML documentation improves the detectability of errors. The availability of the location of the sources increases the probability that errors are removed, and documentation is improved by persons different from the initial developer. Thus, the process of writing user level documentation is integrated into our iterative/incremental strategy of software development.

8 A complete simulation session

In this section we give a complete example of a simulation script illustrating key features of both kernel (section 5) and interpreter (section 6). We simulate a neuron receiving spike input from large pools of excitatory and inhibitory neurons. In each pool the neurons fire randomly at a constant homogeneous spike rate. As a result of this random input, the neuron under study also generates spikes at random times. We are looking for a self-consistent solution, where the firing rate of our neuron equals the firing rate of the neurons in the excitatory pool, using the firing rate of the inhibitory pool as a parameter.

Fig. 5 shows the corresponding simulation script. First an object representing a neuron (described by a particular model `IaFcnNeuron`) is created. A handle for this object is stored in variable `n`. This allows us to configure a parameter of this object (the rise-time of synaptic currents `TauSyn` is set to 0.2 ms). Using the same syntax, an object, `be`, is constructed, representing the pool of excitatory neurons. This pool consists of 16000 neurons, all of them firing at 2.0 Hz. We are not interested in the detailed dynamics of these

```

/n IaF neuron Create def
n
  <<
  /TauSyn 0.2 ms
  >> SetStatus
  (1)

/be BackgroundPopulation Create def
be
  <<
  /rate 2.0 Hz
  /convergence 16000
  >> SetStatus
  (2)

/be_n be n Connect def
be_n
  <<
  /weight 45 pA
  >> SetStatus
  (3)

/bi BackgroundPopulation Create def
bi
  <<
  /rate 12.5 Hz
  /convergence 4000
  >> SetStatus
  (5)

/bi_n bi n Connect def
bi_n
  <<
  /weight -45 pA
  >> SetStatus

/s SpikeDetector Create def
s n ImplantDevice

/OutputRate
{
  /li Set

  bi << /rate li >> SetStatus
  s ResetSpikeTimes
  10000 ms Simulate
  s GetStatus /events get 10000.0 div
} def
(4)
(6)
{OutputRate 0.002 sub} 5.0 15.0 0.0001 FindRoot

```

Fig. 5: Example of a complete simulation script (SLI code presented in two columns). A neuron receives random spike input from an excitatory and an inhibitory population of neurons. The simulation script searches for a fixed point where the output spike rate of the modeled neuron equals the spike rate of the neurons in the excitatory population. Arrows indicate where key features of the interpreter and the kernel-interpreter interaction are used: (1) uniform methods for object creation and configuration, (2) communication with kernel by associative arrays, (3) type safe connector, (4) control passed to kernel, (5) pure function object, (6) functional operator searches for fixed point.

neurons. Therefore, the neurons of the pool are represented by a single object `BackgroundPopulation` describing the collective properties of the pool. Here, we exploit the fact that objects describing components of a neural system at different levels of abstraction can be used simultaneously in a single simulation. Using the two handles `be` and `n`, a connection is established by command `Connect`. Subsequently, the amplitude of the post-synaptic current elicited in `n` by a spike emitted from `be` is specified (`/weight 45 pA`). The same sequence is repeated to create and connect the inhibitory pool, differing from the excitatory pool only in terms of parameter values and the sign of the interaction (`/weight -45 pA`). In the last step of the specification a measurement device (spike detector), `s`, is created, a handle assigned to variable `s`, and the device is implanted into neuron `n`.

The next step is to specify the simulation procedure or protocol. To this end a new SLI function `OutputRate` is defined which takes the firing rate of the inhibitory pool as an argument, and returns the firing rate of neuron `n`. Without discussing the details, we observe that this is achieved as follows. First, the parameter `rate` of `bi` is changed. Then the neuronal dynamics is simulated for 10000 ms. Finally, the number of spike events recorded by detector `s` is read out and converted to a rate.

SLI operator `FindRoot` numerically searches for the root of a function on a specified interval (here `[5.0, 15.0]`) with a predefined precision (here `0.0001`). The function supplied to `FindRoot` is an unnamed (pure) function having a root at the desired output rate 0.002 ms^{-1} .

Thus, we have described an example simulation with a non-trivial interaction between simulation kernel and interpreter. In the search for a self-consistent solution the interpreter repeatedly passes control to the kernel to simulate the neuronal system with altered parameter settings. The efficiency of the kernel in simulating the dynamics, and the flexibility of the interpreter in expressing algorithms are exploited in the same task.

Let the simulation script be stored in a file `searchfxpt.sli`. The fact that the simulation language is interpreted expresses itself in the following interactive session:

```
SLI ] (searchfxpt.sli) run
SLI [1] /x Set
SLI ] 400 ms Simulate
```

The first line carries out the computation we have described above. `SLI]` indicates the prompt of the interpreter in an interactive session. When `FindRoot` terminates it leaves the requested root on the stack. The prompt now reads `SLI [1]`, visualizing that one object is available on the stack. The sequence `/x Set` stores the object in variable `x` and the prompt returns to its initial appearance. The neuronal system created during the execution of `searchfxpt.sli` still exists and in the exact state where the last call of `Simulate` left it. Therefore, we can continue to manipulate and investigate the neural system in interactive fashion (e.g. advance system time by another 400 ms).

9 Applications

The simulation software has already supported a number of publications. Let us discuss the problem of spike synchronization in feed-forward subnetworks to illustrate the different types of simulations that have been performed while investigating such systems. Fig. 6A is a sketch of a feed-forward subnetwork known as a *synfire chain* [2]. Groups of neurons are connected into a chain-like structure. Each neuron receives excitatory input from all neurons in the preceding group. The structure is a subnetwork in the sense that it is embedded in a large random network of excitatory and inhibitory neurons. Thus, in addition to the input from the preceding group, each neuron receives on the order of 10^4 inputs from the remainder of the network. In this setting, the membrane potential of an individual neuron exhibits large random fluctuations (Fig. 6B). Synfire chains were introduced to explain the occurrence

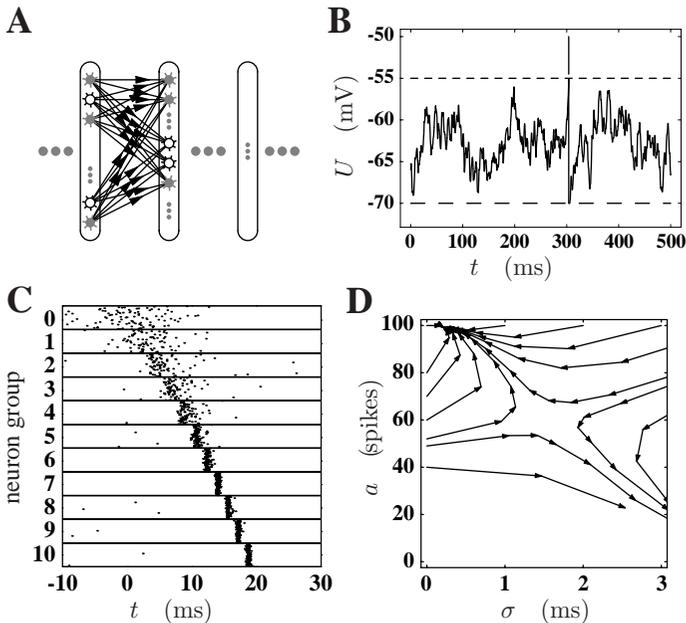


Fig. 6: Examples for the different types of simulations required in investigating a neural system. (A) Feed-forward network of groups of neurons [2]. (B) Simulation of membrane potential fluctuations in a single neuron. Upper dashed line indicates the threshold at which a spike is emitted (a spike occurs at $t \approx 300$ ms). In the absence of any input the membrane potential would reside at resting level (lower dashed line). (C) Spike synchronization in a feed-forward network of thousands of neurons. Panels labeled 1 to 10 contain the spike times (dots) of the neurons in the consecutive neuron groups of A. Panel 0 contains the spikes (centered at $t = 0$ ms) constituting a stimulus applied to group 1. (D) State space constructed for the dynamics in C (assuming 100 neurons per group). Number of spikes a in a packet vertical and temporal spread of spikes σ horizontal. Trajectories are obtained from a deterministic iterative mapping computed from statistics obtained in single neuron simulations. The activity in C corresponds to a trajectory reaching the attractor from the regime of large σ .

of spatio-temporal spike patterns with millisecond precision in the cortex [1]. The underlying question is how precise spike timing can occur, given the large membrane potential fluctuations and the weakness of individual synaptic connections. Investigating the relationship between parameters determining the sub-threshold dynamics of the membrane potential and spike timing connects single neuron properties with network effects. Measures obtained on the two levels need to be simultaneously consistent with experimental results. Consequently, a moderately realistic neuron model needs to be employed [13].

Fig. 6C shows the result of a network simulation where the first group of a chain is stimulated with a broad packet of input spikes. Following the stimula-

tion a packet of spikes travels along the chain. The spike packet synchronizes until some residual spread is reached and is then stably propagated. Such simulations require a large number of neurons but only a simple protocol.

Describing a spike packet by two variables, number of spikes a and temporal spread σ , we were able to construct a two-dimensional transmission function characterizing the single neuron response to spike packet input. Surprisingly, this allowed us to construct a two-dimensional iterative mapping predicting the synchronization dynamics in a synfire chain (Fig. 6D).

We found that there is indeed an attractor for synchronous spiking activity at millisecond precision [12]. Thus, synfire activity seems to be a possible mode of activity in the cortex. To allow numerical state space analysis, the transmission function needs to be available at high precision. Here, only a single neuron needs to be simulated. However, the simulation protocol is complex and requires a large number of iterations at each parameter setting. In a later publication [17] we demonstrated that activity does indeed develop as predicted by our theory, which reduces the thousands of variables of the network to just two variables. In addition, we characterized the variability observed in individual trials not captured by our deterministic iterative mapping.

10 Teaching computational neuroscience

Although the software is primarily used in research, it is also suitable for teaching Computational Neuroscience. Since 1997 we have used our simulation tool in different courses for students of physics and biology. Starting from the material presented in the course, students can continue to explore neuronal dynamics and build more complex systems on their own initiative. Knowledge can be directly applied and deepened in later research. It is unlikely that software specifically developed for teaching is updated as thoroughly and regularly as software used in research. For the teacher, the advantage of using the same software for research and teaching is that with minor pedagogically motivated improvements, the setup for the research can be used in the classroom. In our experience, education in theoretical neuroscience requires teaching at three different levels.

The first level is concerned with the topic of the research: the structure and dynamics of neural systems. At this level the simulator should support investigation of the dynamical properties of the model. The performance of “virtual experiments” should be comfortable (see section 8) and not be obfuscated by technicalities of the simulation. Depending on the course, implementations of the models used and simulation scripts may even be supplied by the teacher.

The second level is concerned with computational physics topics. Here we deal with the question of how a model can be represented in a computer and how the system can be solved accurately and efficiently. This involves standard numerical techniques (e.g. [27]) and techniques specific to neuronal modeling (e.g. [21, 28]).

The third level is concerned with computer science topics. Access to the sources of the simulation software, including the implementation of the infrastructure (e.g. the simulation language interpreter), enables a discussion of standard computer science techniques in a concrete and applied context. Reviewing many lines of code from neuroscientists, we have learned that neuroscientists usually still do not have access to the appropriate computer science literature and are therefore not aware of standard solutions to the problem at hand. This often results in unnecessarily long development times and suboptimal code.

A particular course can focus on any of the three levels or combine aspects of several levels.

11 Discussion

In the present paper we have described a software environment for the simulation of neural systems which has evolved and been heavily used over the course of several years [11] and led to new insights into the dynamics of neural networks (see section 9).

After 25 years of intensive neuroscience research with computers [23], no clear picture has emerged of how simulations of neural systems should be expressed in software and be made communicable and reproducible. Reasons for this situation are manifold: the inhomogeneity of models describing neural systems at different levels of abstraction, the rapid progress in experimental research, the rapid changes in computer technology and the short term funding of research have all contributed. We have argued that due to the design decisions we made (section 3), we arrived at an efficient simulation system that is able to cope with the demands of the coming years. The four most important aspects are:

- the use of object oriented language and technology,
- the possibility of having network elements at different levels of abstraction and different interactions,
- the view that devices like a spike detector and the protocol are an integral part of a simulation, and
- the strict separation into software layers (e.g. kernel, interpreter, graphics).

However, equally important to the success of our project was gaining an understanding about the conditions under which the software is developed,

the resources available, and the risks of a long term project (section 4). The four most important consequences are:

- the software is developed in a collaboration of research groups,
- an iterative/incremental strategy is used avoiding long periods where the software cannot be used,
- free software and open standards are used, and
- mechanisms are devised that support rapid availability of documentation.

The next scientific target is to investigate networks of on the order of 10^5 neurons (see below). Being able to simulate networks of this size enables us for the first time to simulate the volume of cortex from which multiple single unit recordings are usually experimentally obtained. This allows us to make better predictions for spike synchronization and to compare sub- and supra-threshold dynamics in a realistic network environment. Artificial down-scaling of the number of inputs per neuron is no longer necessary.

At the functional level some of our collaborators are interested in modeling large parts of the visual system. This will lead to an increase of network elements on level of abstraction above the single neuron level. At the order of 10^5 neurons, parallel or distributed methods are required to solve network dynamics in appropriate time, and to have sufficient memory available. With a test implementation of a distributed version of the simulation kernel we have already performed preliminary scientific studies. This parallel kernel uses the MPI library for distributed computing [26]. The next task in the development of our software is to re-integrate these parallel methods with the simulation language interpreter, in order to be able to use parallel methods routinely, and perform complex experimental setups and protocols. However, with SMP computers having large homogeneously addressable memory becoming available at reasonable costs an implementation using POSIX threads [10] also becomes an interesting option.

The code base is constantly growing, as is the number of users. Therefore, some of our management procedures will have to be adapted. Until now, no formal bug-tracking mechanism has been introduced.

To get an overview of the code base it is instructive to look at the distribution of code lines and files over main components of the project Tab. 1. In terms of the amount of C++ code, simulation kernel and SLI interpreter seem to be of equal complexity. The SLI support library are the parts of the simulation software implemented in SLI itself. Note again, that the documentation is extracted from the respective SLI or C++ source code (section 7). The figures for simulation kernel, SLI interpreter, and SLI startup library, thus, include both code and documentation.

Especially in the C++ components, large parts of the code are not described by user level documentation, because the basic machinery is only relevant for the developer. In the future it will become increasingly important to introduce

| Part | Lines | Files |
|-----------------------------|--------|-------|
| Simulation kernel | 20 000 | 137 |
| SLI interpreter | 26 000 | 127 |
| SLI support library | 10 000 | 18 |
| Documentation from SLI code | 3 700 | 177 |
| Documentation from C++ code | 4 000 | 249 |

Tab. 1: The code base of NEST.

a process to generate developer level documentation similar to the technique we currently use to generate user level documentation. Doxygen [31] is a candidate tool already used in parts of the project.

Instead of using an adapted general purpose language like SLI, a dedicated *Neuro Modeling Language* could solve a number of problems related to the description of neuronal network simulations. Moreover, such a language could lift the degree of collaboration to a new level, because many teams would use the same language to define their models. Up to the present day no Neuro Modeling Language exists. However, recently considerable research in this direction has started in which we are actively participating (www.neuroml.org). These efforts are promising since they have strong support in the research community.

In the last two years we have managed to install a formal collaboration, the Neural Simulation Technology Initiative (NEST Initiative), between several labs (Fig. 3) which regulates the exchange of concepts and software. In this collaboration the simulation software serves as the reference implementation for new models and methods. Having these resources available has considerably accelerated the development process. The software that is made publicly available (GPL) continues to be distributed under the name SYNOD, the name of the initial version. Some of our results may also be of interest for the development of software tools in other areas of research.

Acknowledgements

This paper documents the results of a collaborative work in which our names are representative for all members of the NEST collaboration (alphabetical by institution): Agricultural University of Ås, Norway: Hans Ekkehard Plesser; University of Freiburg, Germany: Prof. Ad Aertsen, Ulrich Hehl, Carsten Mehring, and Stefan Rotter; Max-Planck Institut für Strömungsforschung, Göttingen: Prof. Theo Geisel; Honda R&D Europe (Deutschland) GmbH, in Offenbach/Main: Prof. Edgar Körner and Andreas Richter; University of Marburg: Rüdiger Kupper.

We thank Prof. Ad Aertsen whose ideas on a *zoom-in simulator* strongly influenced the concepts presented in this paper. He supports and shapes this project since its very beginning.

We gratefully acknowledge the help of Prof. Moshe Abeles, who kindly provided us with the source code and personal notes of his synfire chain simulator. His help has been invaluable in the design of the first version of the simulation kernel.

Partial funding for this project was obtained from the Deutsche Forschungsgemeinschaft (DFG), the German-Israeli Foundation for Scientific Research and Development (GIF), and the Human Frontier Science Program (HFSP).

References

- [1] M. Abeles, H. Bergman, E. Margalit, and E. Vaadia. Spatiotemporal firing patterns in the frontal cortex of behaving monkeys. *Journal of Neurophysiology*, 70(4):1629–1638, 1993.
- [2] Moshe Abeles. *Corticomics: Neural Circuits of the Cerebral Cortex*. Cambridge University Press, Cambridge, 1st edition, 1991.
- [3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. professional computing. Addison-Wesley, 1998.
- [4] Adobe Systems Inc. *The PostScript Language Reference Manual*. Addison-Wesley, 2nd edition, 1991.
- [5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, Reading, Massachusetts, 1988.
- [6] Moshe Bar and Karl Fogel. *Open Source Development with CVS*. Coriolis Group Books, 2nd edition, 2001.
- [7] Grady Booch. *Managing the Object-Oriented Project*. Addison-Wesley, Reading, Massachusetts, 1996.
- [8] James M. Bower and David Beeman. *The Book of GENESIS: Exploring realistic neural models with the GEneral NEural Simulation System*. TELOS, Springer-Verlag, New York, 2nd edition, 1997.
- [9] Frederick P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley Longman, anniversary edition, 1995.
- [10] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Boston, 1997.
- [11] Markus Diesmann, Marc-Oliver Gewaltig, and Ad Aertsen. SYNOD: An environment for neural systems simulations - language interface and tutorial. Technical report, The Weizmann Institute of Science, 76100 Rehovot, Israel, 1995. Technical Report GC-AA/95-3.
- [12] Markus Diesmann, Marc-Oliver Gewaltig, and Ad Aertsen. Stable propagation of synchronous spiking in cortical neural networks. *Nature*, 402:529–533, 1999.
- [13] Markus Diesmann, Marc-Oliver Gewaltig, Stefan Rotter, and Ad Aertsen. State space analysis of synchronous spiking in cortical neural networks. *Neurocomputing*, 38–40:565–571, 2001.
- [14] Raphael A. Finkel. *Advanced programming languages*. Addison-Wesley, Menlo Park, California, 1996.
- [15] Richard M. Fujimoto. *Parallel and distributed simulation systems*. Wiley, New York, 2000.
- [16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1994.
- [17] Marc-Oliver Gewaltig, Markus Diesmann, and Ad Aertsen. Propagation of cortical synfire activity: survival probability in single trials and stability in the mean. *Neural Networks*, 14:657–673, 2001.

- [18] Mark Harrison and Michael McLennan. *Effective Tcl/Tk programming: writing better programs with Tcl and Tk*. Addison-Wesley, Reading, Massachusetts, 1998.
- [19] Michael Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9:1179–1209, 1997.
- [20] Nicolai Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, Reading, Massachusetts, 1999.
- [21] Christof Koch and Idan Segev, editors. *Methods in neuronal modeling: from ions to networks*. A Bradford Book, MIT Press, Cambridge, 2nd edition, 1998.
- [22] Edgar Körner, Marc-Oliver Gewaltig, Ursula Körner, Andreas Richter, and Tobias Rode-
mann. A model of computation in neocortical architecture. *Neural Networks*, 12(7–
8):989–1005, 1999.
- [23] Ronald J. MacGregor and Edwin R. Lewis. *Neural Modeling, Electrical Signal Processing
in the Nervous System*. Plenum Press, New York, 1977.
- [24] MathWorks. *MATLAB The Language of Technical Computing: Using MATLAB*. Natick,
MA, 1998. 24 Prime park Way, Natick, Mass. 01760-1500.
- [25] John K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing. Addison-Wesley,
1994.
- [26] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, California, 1997.
- [27] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Nu-
merical Recipes in C*. Cambridge University Press, 2nd edition, 1992.
- [28] Sefan Rotter and Markus Diesmann. Exact digital simulation of time-invariant linear
systems with applications to neuronal modeling. *Biological Cybernetics*, 81:381–402,
1999.
- [29] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesely, New York, 1994.
- [30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesely, New York, 3rd
edition, 1997.
- [31] Dimitri van Heesch. Doxygen, 1997. <http://www.stack.nl/~dimitri/doxygen>.
- [32] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf,
Automake, and Libtool*. New Riders, 2000.
- [33] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 3rd edition, 2000.
- [34] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press,
Cambridge, UK, 3rd edition, 1996.