Chapter 5

SpikeNNS simulator: design and implementation

The development in the last decade of a third generation of neural networks (Maass, 1997) has required the design of dedicated simulation environments. Simulation platforms either for the detailed modeling of individual neurons, such as Neuron (Hines and Carnevale, 1995) and Genesis (Bower and Beeman, 1998) or dedicated to large–scale neural simulations, such as SpikeNet (Delorme et al., 1999), Infernet (Sougné, 1999), NSL (Weitzenfeld et al., 1999) have been created in recent years. Due to the large attention given to realistic modeling of the structure of biological neurons, the most powerful and flexible simulators in existence today implement detailed neural models. As a result, these simulators become more appropriate for the modeling of single neuron behavior and less suitable for large-scale modeling of cognitive phenomena.

The aim of the modeling work presented in this thesis is to provide an example of how cortical processes, such as movement planning or visuomotor mapping, are grounded in the neural substrate. In Section 4.3 it was argued that a simplified neural model, such as the Spike Response Model (SRM) (Gerstner, 1999) corresponds best to our modeling goals. To date, SRM has been implemented only as part of simulations that are dedicated to specific tasks (Gerstner, 1999; Lerchner, 2001). Therefore, in the absence of a flexible simulation platform that can be adapted to our modeling goals, the approach taken in this thesis was to create a library of functions that model networks of SRM neurons. This library was implemented as an extension of the general purpose–simulator Stuttgart Neural Network

Simulator (SNNS) (Zell et al., 1992).

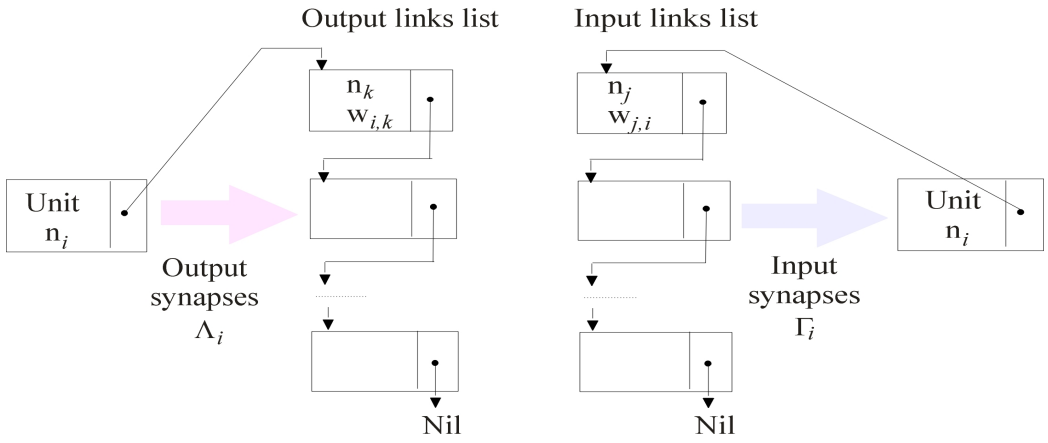
This chapter presents the implemented system, entitled SpikeNNS. We begin in Section 1 by describing the implementation of the neural model introduced in Chapter 4, Section **??**. When modeling large–scale networks of spiking neurons with plastic synapses, the time and memory efficiency of the simulation became essential issues in the design of the simulator (Jahnke et al., 1999; Mattia and Del Giudice, 2000). In the first section, we present several versions of the neural model implementation, aimed at decreasing the computational workload per neural unit. Section 2 focuses upon how learning with networks of spiking neurons is implemented. In this part, we deal with efficiency issues at the network level, partially solved by implementing a discrete event–driven strategy. The issue of high neural activity patterns is addressed, when spike events on the order of 10^5 are generated in the network. Several strategies are proposed that lead to a highly efficient event–driven algorithm that reduces the simulation time up to 20 times compared to the baseline algorithm. Finally, Section 3 outlines the configurable features of the SpikeNNS simulator.

5.1 Implementation of SpikeNNS neural model

The neural model implemented in SpikeNNS is a simplified version of the Spike Response Model (Gerstner 1999), described by the set of Equations 4.2, 4.3, and 4.5 (Section 4.3). This section presents the effective implementation of the neural model and focuses upon the time–efficiency of the simulation.

5.1.1 Design considerations

The implementation of a neural model requires consideration of several design issues. One issue concerns the type of the synaptic connectivity employed. This falls into two categories: (1) the *regular connectivity* follows some deterministic rules, giving rise to patterns such as receptive fields or topographical projections; (2) the *non-regular connectivity* consists of sparse, probabilistic connections. In SpikeNNS both categories of connection patterns are implemented. However, while regular connections are relatively simple to configure and manage efficiently with respect to the time and memory load, sparse connections are more difficult to deal with. The difficulty concerns both the generation of the probabilistic pattern of synapses, and the efficient management of such a pattern (see Section 5.3.2).



(a) Sender oriented scheme

(b) Receiver oriented scheme

Figure 5.1: Data structures used to store the connections list. (a) A sender-oriented connectivity stores the weights and addresses of the output connections Λ_i (b) In the receiver-oriented scheme the weights and addresses of input links Γ_i are stored.

A commonly used method in representing the sparse connectivity pattern is to create a list of connections, one for each neuron n_i. Each item in the list contains the weight of the connection and the address n_j of the pair unit. If the address stored denotes the target neuron n_j to which n_i sends spikes, then the scheme is called *sender–oriented* (Figure 5.1a). When the list stores the addresses of the input nodes, that is, those neurons from which n_i receives spikes, the scheme is referred to as *receiver–oriented* (Figure 5.1b). Each of these schemes demands different processing of spike generation and distribution.

In a receiver–oriented scheme it is the job of the receiver unit to handle the incoming spikes and to set the transmission delays. Whenever a unit computes its activation it checks all its input synapses to see whether a spike has been emitted on any of them. If this is the case, then it sets the delay of the spike transmission (see lines 3-6 in Algorithm 1). It is important that the algorithm ensures that the delay for each spike is computed only once. Alternatively, in the sender-oriented scheme, the distribution of spikes and the generation of synaptic delays are managed by the emitting neuron.

A number of simulators implementing the traditional rate–coding neural model use the receiver-oriented scheme (i.e., SNNS). The reason for this is that scheme can be advantageous in the case of a continuous neural output function, which causes constant firing activity on synapses. However, this is not the case with spiking neurons activity. In Sections

5.1.2 and 5.1.3 the appropriateness of each connectivity scheme for the implementation of the spiking neural model is discussed.

Another design issue which has to be considered at the neural level is the method by which the integration of the membrane potential is realized (Equation 4.2, Section 4.3). A straightforward way to compute the sum term in Equation 4.2 consists of the following. Whenever an update of the unit activation is requested, the values of all previous postsynaptic potentials coming from all presynaptic nodes are computed, integrated and added to the refractory kernel. The optimization of this method is possible if we consider that the depolarization of the postsynaptic potential ϵ follows a deterministic function (see Equation 4.3). Therefore, its value at time t can be expressed as a function of its previous value at $t - \Delta t$. This idea is explored in Algorithm 3 in Section 5.1.3.

Besides considering *how* we integrate a neuron activity, we have to decide *when* we do this integration. Solutions to this problem reside in using either a synchronous, time-driven integration of unit activity or an asynchronous integration. Both strategies are implemented and compared with respect to their efficiency for simulation of pulsed neural networks.

5.1.2 Activation and output functions - version 1

The first version of the Spike Response Model (Section 4.3) in SpikeNNS was created using the SNNS receiver-oriented connectivity scheme. Equations 4.2, 4.3 and 4.5 have been implemented within the body of two functions: the *Activation* function (ACT_Spike) and the *Output* function (OUT_Spike). A straightforward (non-optimal) implementation of these functions is shown in Algorithm 1. Note that the Output function could have been incorporated in the Activation function, but is kept separate to ensure compatibility with the implementation of these functions in SNNS.

To get a quantitative measure of the complexity of the algorithms presented in this section, we estimate the scaling of the number of operations required per algorithm, with the variables of the simulation process. The main variables are: T, the time of the simulation; N, the number of neurons in the network; S, the synapses per neuron; F_j, the set of non-negligible firing times on each synapse. Other variables that affect the number of the operations executed will be introduced in the following subsections. Note that our measure NO is given by the number of operations in the performance critical parts (i.e., such as loops) and does not reflect the exact number of low-level operations required in the implementation. In the

Algorithm 1 ACT_Spike v1 -receiver oriented

Require: node i, current time t
Ensure: activation value $V_i(t)$
1: $V_i(t) = 0$
2: **for all** $j \in \Gamma_i$ **do** {presynaptic nodes to i}
3: t_j^l = get_spike(j) {receiver unit checks for new spike from this unit}
4: **if** $t_j^l \notin \mathrm{F}_j$ **then**
5: d_j^l = noisy_delay(length(w_{ji})) {generate delay}
6: insert_spike (t_j^l, F_j) {insert the spike in the set of firing times}
7: **end if**
8: **for all** $t_j^k \in F_j$ **do** {all previous firing times of node j}
9: $PSP_j^k(t) = \epsilon(t - t_j^k - d_j^k)$ {compute kth postsynaptic potential}
10: $V_i(t) = V_i(t) \; + \; PSP_j^k(t) \cdot w_{ji}$ { compute weighted sum of PSPs}
11: **end for**
12: **end for**
13: $REF(t) = \eta(t - t_i^{(f)})$ {calculate refractory period}
14: $V_i(t) = V_i(t) \; - REF(t)$ {subtract REF from the PSPs aggregate}
15: **if** $V_i(t) \geq \theta$ **then** {if the neuron fires}
16: $t_i^{(f)} = t$ {record firing time}
17: **end if**
{No propagation of emitted spike is done by the sender unit}

Algorithm 2 OUT_Spike

Require: activation $V_i(t)$
Ensure: output value $O_i(t)$
1: **if** $V_i(t) \geq \theta$ **then** {the neuron fires}
2: $O_i(t) = 1$ {emit a signal, set output to 1}
3: **else**
4: $O_i(t) = 0$ {no signal}
5: **end if**

case of Algorithm 1, the critical computational effort is represented by the computation of the ϵ kernel (i.e., given by an exponential function) for F_j times, for each input synapse in S (lines 8-11). This entails $NO \approx S \cdot |F_j|$. In addition, checking the firing activity of each input synapse (lines 3-7, Algorithm 1) needs a supplementary effort that scales with S. Then, the estimated number of operations for Algorithm 1 is:

$$NO_1 \approx S \cdot (|F_j| + 1). \tag{5.1}$$

5.1.3 Acceleration of the activation function - version 2

Let us now consider the sender-oriented connectivity scheme. In Algorithm 1, the receiver unit checks the activity of *all* incoming synapses at each integration step (lines 2-7). By contrast, in a sender–oriented strategy there is no need for this additional operation. Whenever a spike is generated, it becomes the task of the emitting unit (or of the main simulation engine) to compute the propagation delays for all output synapses. Because the propagation is performed only once for each spike, the uniqueness of the delay value per spike is ensured. The sender–oriented scheme is implemented in Algorithm 3.

Algorithm 3 ACT_Spike v2 - sender oriented

Require: node i, current time t, previous integration time t_{int} and previous weighted sum $PSP_i(t_{int})$
Ensure: activation value $V_i(t)$
1: $x = t - t_{int}$ {elapsed time from last integration}
2: $PSP_i(t) = PSP_i(t_{int}) \cdot exp(-x/\tau_m)$ {decay the sum of past PSPs}
3: **for all** w_{ji}^k with $t_j^k > t_{int}$ **do** {only spikes that occurred since last integration}
4: $\quad PSP_j^k(t) = \epsilon(t - t_j^k - d_j^k)$
5: $\quad PSP_i(t) = PSP_i(t) + PSP_j^k(t) \cdot w_{jj}$ {add new weighted PSP to the sum}
6: **end for**
7: $V_i(t) = PSP_i(t)$ {set the voltage to the aggregate of inputs}
8: $REF(t) = \eta(t - t_i^{(f)})$ {compute refractory period}
9: $V_i(t) = V_i(t) + REF(t)$ {subtract refractoriness }
10: **if** $V_i(t) \geq \theta$ **then** {if the neuron fires}
11: $\quad t_i^{(f)} = t$ {record firing time}
12: **end if**
{Note that propagation of spikes is handled by the main simulation engine (lines 14-17 in Algorithm 4)}

Algorithm 1 can be further improved through the judicious choice of the method used to integrate the membrane potential. The new version of the algorithm exploits the fact that between any two firing moments, a neuron depolarization has a deterministic evolution. Rather than computing the sum over all the postsynaptic potentials at each time step, the contribution of the previous spikes effects on the node i is stored and it is decayed every time when a new potential has to be added. Thus, at the current time moment t the previous sum computed at time t_{int} is decayed by the formula $PSP_i(t) = PSP_i(t_{int}) \cdot exp[-(t - t_{int})/\tau_m]$, where τ_m is the time membrane constant (line 2 in Algorithm 3).

Apart from decreasing the simulation time this method has the advantage of minimizing the memory load. Thus, for the receiver–oriented scheme the transmission delays d_j^k of

all spikes in F_j emitted on each synapse j in the network have to be stored. For instance, in the case of a network with $1k$ units and 10% connectivity, where the last 10 spikes are stored for each synapse, the additional memory load is on the order of 10^5 floats. Instead, in the sender-oriented scheme only two values are kept within each neuron structure: the last integration moment t_{int} and the corresponding decayed sum $PSP_i(t_{int})$.

As a result of the optimizations implemented, the computational effort of the Algorithm 1 is reduced in Algorithm 3 at $NO \approx aS \cdot bF_j$, where a and b are two variables which range between 0 and 1. The term bF_j represents the number of spikes released on a single synapse between two integration moments. The value of the parameter b depends on the time resolution of the simulation. By considering an algorithm that increments simulation time in fine clock steps (e.g., 1 ms) (see Section 5.2.1) or which integrates the neuron's activity for each new spike received on a synapse (see Section 5.2.1) than bF_j is 1. This means that no more than one new spike per synapse has to be processed at any call of the activation function. The term aS represents the percent of synapses on which new spikes are emitted between two integration moments. Hence, the estimated number of operations for computing a single unit in a time bin reduces to

$$NO_2 \approx aS. \tag{5.2}$$

with a a variable dependent on the network activity. The differences in performance between the two versions of the *Activation* function (Algorithm 1 vs. 3) becomes more obvious when these algorithms are employed for the integration of a large number of units over a long time interval (see Section 5.2.1).

The algorithms presented above illustrate in turn, how receiver– and sender–oriented schemes may be implemented separately. However, by using the SNNS libraries, it has been possible to construct a mixture of these algorithms that uses both sets of data structures. This is, because SNNS implements a receiver-oriented scheme, to which we added, rather than replacing (i.e., for compatibility reasons the default connectivity could not be deleted) structures for the sender–oriented connections.

By doing this, we found that the most time efficient method for integrating spikes may be a composite version of the two algorithms. That is, during a first time period when the input patterns are applied, values associated with these spikes are stored in the input synapses, using the receiver–oriented structures. These spikes are integrated only once at an arbitrary time moment T_{int}, by using the first algorithm (see description of time coded input patterns

in Section 4.3.4). After first integration is done at T_{int}, further spikes are integrated one at a time, using the second algorithm. The advantage of using the first algorithm to store and process the values of the input patterns spikes, is that it avoids the generation and handling of a spike event for each firing of an input unit. In this way it is possible to significantly reduce the number of events the algorithm has to deal with. Nevertheless, this is not a portable solution, because it makes use of two lists of connections that can introduces a significant memory load. For spiking neuron simulations the sender-oriented connectivity is the best choice from the two schemes.

What remains to be explained for Algorithm 3 is how it computes the values of the incoming postsynaptic potentials (lines 3 and 4). This is because, by implementing a sender–oriented connectivity solely, a unit does not have access to the values of the input synapses (i.e., the weights and spikes delays). Consequently, it does not know how to calculate the postsynaptic potentials. A possible solution is to parse the output connection structures of all units in the network to find a neuron input synapses. However, this is an exhaustive, time consuming solution. An alternative is to create a data structure for each spike emitted on a synapse that stores the necessary values: the addresses of the emitting and target units n_i, n_j, the firing presynaptic time t_j^k, the transmission delay d_j^k and the synaptic weight w_{ji}. This solution, of a spike-oriented nature, in contrast with the synapse-oriented scheme, leads us naturally into an event-driven approach to the simulation (see Section 5.2.1 below). The complete description of how these spikes are created and delivered is given in Section 5.2.2.

5.2 Simulation of networks of spiking neurons

Up to this point, we focussed upon the simulation of one time step of neural behavior. This consists of computing the neuron's membrane potential by integrating the effects of the incoming spikes (i.e., the postsynaptic potentials) with the effect of the neuron's own spikes (i.e., the refractoriness). In this section, we address the issue of information processing and learning within a network of spiking neurons during a certain time interval.

An essential implementation aspect in building discrete simulations concerns the way the simulation time is progressed. Performing a simulation means to mimic the occurrence of events (i.e., spikes) as they evolve in time and recognizing their effects as represented by states (i.e., network activity) (Ferscha and Tripathi, 1994). The simulation of one basic time step for a whole network is called a *time slice* (Jahnke et al., 1995). Most commonly each time

slice involves four operations:

- **Activity propagation.** Spikes from input or hidden units are propagated through the hidden layers of the network.
- **Integration.** Each unit in the network calls the *Activation* function and integrates its inputs.
- **Output.** If a unit fires, the spike time is recorded and the signal is managed according with the implemented strategy (see Sections 5.1.2, 5.1.3).
- **Learning.** Adaptation of the synaptic weights can follow each pattern application or wait until all input patterns are applied. Information concerning learning (e.g., spikes timings, frequency of spiking) may be recorded for each time slice.

In this section, we present a number of algorithms for the simulation during a time period T, of the four operations described above. The first three of them, namely the activity propagation, the integration and the output are implemented in the function PROPAGATE_Spike. Learning is applied after a time coded pattern is presented to the network and the activity is propagated through the hidden layers for a certain time interval. A general training framework is described in the LEARN_Spike function. Note that when computing with spiking neurons, a single time coded pattern actually consists of a set of input signals, each applied as a distinct sub-pattern at a certain time moment (see description of input patterns application in Section 4.3.4). Hence, we bear in mind that the PROPAGATE_Spike function applies only a time coded pattern, consisting of several sub-patterns, while the entire training set is learned using the LEARN_Spike function.

5.2.1 Continuous vs. event–driven protocols

Currently, two kinds of discrete simulation are distinguished with respect to the way simulation time is progressed: *time driven* simulation and *event driven* simulation. In a *time driven* simulation, time is advanced in steps of a constant size Δt (Ferscha and Tripathi, 1994). The choice of Δt influences the simulation accuracy. That is, ticks short enough to guarantee the required precision generally imply longer time simulation. The continuous, time–driven protocol appears to be the *de facto* standard for detailed neural modeling (Bower and Beeman, 1998). This is because, it ensures the time resolution needed for the integration of the differential equations describing the model.

On the other hand, large–scale simulations that involve learning in plastic synapses and long training procedures are faced with important time and memory efficiency issues. These are usually tackled through the use of simplified neural models and event–driven strategies (Watts, 1994; Mattia and Del Giudice, 2000; Claverol et al., 2002). Efforts have also been made to implement simulations on parallel computers (Fujimoto et al., 1992; Brettle and Niebur, 1994; Jahnke et al., 1995) or to create dedicated hardware (Schoenauer et al., 1998; Elias and Northmore, 1999).

Continuous time algorithm

A straightforward implementation of a continuous–time algorithm for the simulation of the network activity during a training step is presented in Algorithm 4.

Algorithm 4 PROPAGATE_Spike - time driven

Require: current pattern number p, simulation time st, time out t_{out}

1: $ct = st$ {set the time clock}
2: **while** $ct < t_{out}$ **do** {while current time less than time out}
3: **for all** $l \in Input(p)$ **do** {all input units}
4: **if** $l(ct) = 1$ **then** {if there is an input value at the current moment}
5: out^l = OUT_Identity($l(ct)$) {send a pulse}
6: **for all** $i \in \Lambda_l$ **do** {output synapses to hidden nodes}
7: d_i^l = noisy_delay(input_delay) {set delay for input spike from l to i}
8: **end for**
9: **end if**
10: **end for**
11: **for all** $i \in$ Net **do**
12: act^i = ACT_Spike(i, ct) {call activation function}
13: out^i = OUT_Spike (act^i)
14: **if** out^i **then** {propagate spike to postsynaptic nodes}
15: **for all** k $\in \Lambda_i$ **do** {all postsynaptic nodes to i}
16: d_i^k = noisy_delay(length(w_{ik})) {compute spike delay}
17: **end for**
18: insert(i, LearningNodes) {synapses of firing units are subject to learning}
19: **end if**
20: **end for**
21: $ct+ = \Delta t$ {increment the current time contor}
22: **end while**

Let us consider the computational effort demanded by the execution of the Algorithm 4, when the activation functions described in Section 5.1 are used in turn (in line 12). Thus, by using the first version of the activity function from Algorithm 1, to compute a network with

N neurons, S synapses per neuron, during a time interval T with a time step Δt, a number of operations are required:

$$NO_3 \approx \frac{T}{\Delta t} \cdot N \cdot S \cdot (|F_j| + a)). \tag{5.3}$$

The first term in the sum accounts for the computational effort needed to integrate the activities of all units (lines 11-13, Algorithm 4). The second term estimates the operations required for the distribution of spikes that occur in a time slice, at a network activity a (lines 14-17, Algorithm 4). The value of a is defined as the average number of spikes in a time slice divided by the number of neurons and takes values in the range of $[0, 1]$. When using the second version of the activity function (Algorithm 3), both computational complexities scale with the network activity a. The number of operations reduces to:

$$NO_4 \approx \frac{T}{\Delta t} \cdot a \cdot N \cdot S \cdot 2. \tag{5.4}$$

These two estimations of the number of operations, neglect the computational effort required for the distribution of the input spikes (lines 3-10, Algorithm 4). This is mainly because, input units have an identity activation function that does not require the computation of a response kernel and the total number of input patterns is small compared to the number of times the network activity during a training step.

The number of operations implemented by Algorithm 4 is higher when using the first version of the activation function. This is, because at each time step, the activity of every unit is updated by computing the ϵ values for all presynaptic firing times in $|F_j| \approx 10$. Conversely, the second version of the algorithm decreases the number of operations, by reducing the computational effort per neuron to the integration of one spike from each active synapse aS in the network.

Both implementations of Algorithm 4 operate in a synchronous way. They integrate all neurons N at each time step $T/\Delta t$ that determines their poor scaling with the network size and the simulation time. Moreover, the global integration of the network activity may not be necessary, due to the fact that at each time moment only a small number of neurons receive a new spike. An observation can be made here, with respect to the differences between the simulation of spiking neurons and rate–coding neurons. Learning based on a continuous neural model makes use of the integration of network activity at each time step. This is because, the output synapses are constantly active due to the continuous output function of the neurons. By contrast, the output of spiking neurons are discrete, rare events, whose transmission is affected by axonal latencies and which reach the target units at certain time

moments. A simulation which discretizes the computation of unit states (e.g, activities) at spike-event occurrences seems more appropriate for a pulsed neural network simulation.

Event driven algorithm

An alternative to time driven simulation is to implement an event driven strategy (Ferscha and Tripathi, 1994). Instead of advancing the simulation time in a continuous way and processing events synchronously at each clock tick, the integration of a unit activity can be performed in an asynchronous way, triggered by the reception of one or several spike-events (Watts, 1994; Grassmann and Anlauf, 1998; Mattia and Del Giudice, 2000). In this section we describe Algorithm 5 that implements a basic event-driven strategy for the computation of the network activity during a training step.

Algorithm 5 PROPAGATE_Spike - event driven

Require: current pattern number p, simulation time st, time out t_{out}
1: $ct = st$ {set the time clock}
2: $SL = nul$ {init spike list}
3: **while** $ct < t_{out}$ **do** {while current time less than time out}
4: **if** $SL = nul \vee ct \geq$ time(next_subpattern(p)) **then**
5: apply_inputs (next_subpattern(p)) {call lines 3-10 Algorithm 4}
6: **end if**
7: e = first_event(SL)
8: $ct = e.time$ {simulation time becomes the time of the current spike }
9: **if not** nul_event(e) **then**
10: $i = e.target$ {get the spike's target unit}
11: act^i = ACT_Spike(i, ct) {integrate activation for target unit i}
12: out^i = OUT_Spike(act^i)
13: SL = remove_first_event(e, SL) {delete event e from the list}
14: **if** out^i **then** {propagate spike to postsynaptic nodes}
15: **for all** k $\in \Lambda_i$ **do** {all postsynaptic nodes to i}
16: d_i^k = noisy_delay(length(w_{ik}))
17: e = create_spike_event($k, ct + d_i^k$)
18: SL = insert_event_order(e, SL)
19: **end for**
20: insert(i, LearningNodes) {synapses of firing units are subject to learning}
21: **end if**
22: **end if**
23: **end while**

The core of Algorithm 5 consists in processing spiking events from a chronologically ordered list SL (lines 7- 13). Whenever a unit fires a spike, a new event e is created and inserted

in order in the spike list SL (lines 17-18). Each new spike is fully characterized by a time stamp representing the delivery moment and the index of the target unit. The input patterns are applied when the event list becomes empty or when the current time of the simulation exceeds next sub-pattern time stamp (lines 4-5).

The *simulation engine* of the event–driven algorithm is presented in Figure 5.2. It continuously takes the first event from the event list (i.e., the one with the lowest timestamp) and delivers it to the target unit. This process continues until a time out moment is reached or no further events have occurred. The most important task of the event-driven engine is to allow an asynchronous processing of the events, while preserving the time order of the events. Doing this, makes the parallelization of the simulation possible, because it permits events to be processed by different processors, while the time order is ensured (Mohraz et al., 1997; Jahnke et al., 1999). The most common procedure for achieving this goal is to maintain a global ordered list of events and whenever a new spike is generated, it is inserted in order in the list. Alternative methods have been proposed, mainly by creating and updating several lists instead of one, in order to reduce the excessive increase of a global list length (Mattia and Del Giudice, 1999; Claverol et al., 2002). In Section 5.2.2 it is presented our solution to this problem.

Evaluation of performances

The event-driven algorithm presented above leads to a significant decrease in the number of operations required computing the network activity during a training cycle. By using the efficient version of the activation function ACT_Spike2 in Algorithm 5, the simulation of the network activity in a time period T requires a number of operations:

$$NO_5 \approx \frac{T}{\sigma t} \cdot a \cdot N \cdot S \cdot (a + 1 + \log(length(SL))) \tag{5.5}$$

Here, σt represents the time resolution used in the generation of noisy delays and input signals (i.e., note that is different from Δt step). The first term represents the computational effort employed by the integration of all units that receive spikes at a certain moment of the simulation (lines 7-12, Algorithm 5). Note that it scales with a^2. Second term is the computational effort for the spike distribution (lines 14-17, Algorithm 5). Compared to the previous versions, this algorithm brings an additional computational effort required by the insertion in order of the new spikes in the event list (line 18, Algorithm 5). The length of the list scales with $T/\sigma t \cdot a \cdot N \cdot S$.

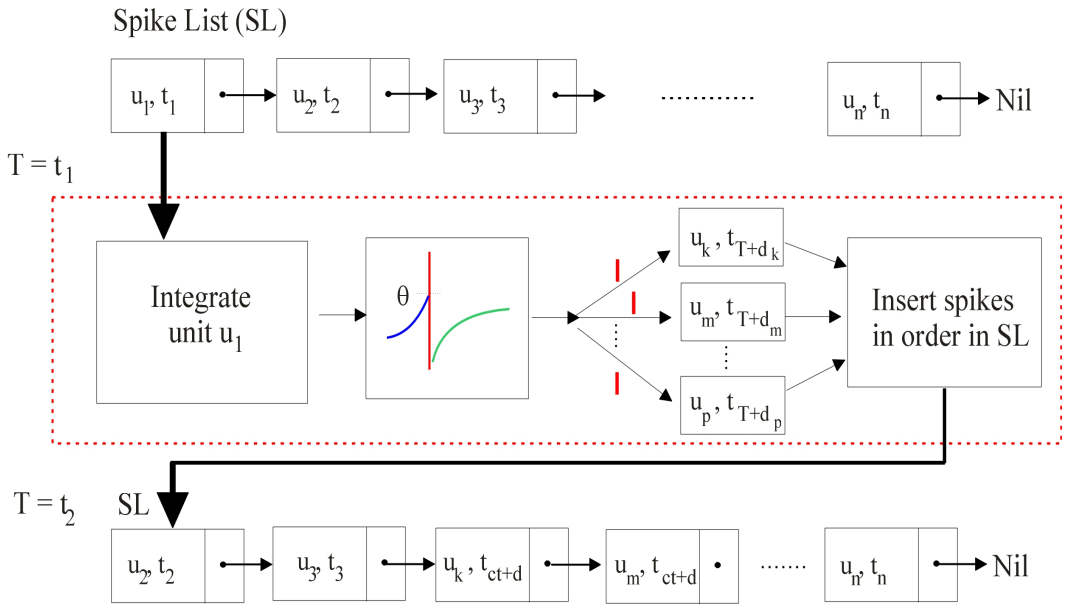


Figure 5.2: Event-driven simulation engine. At $t1$ the spike list SL contains events sorted in a chronological order. The engine takes the first spike in the list (i.e., the oldest) and delivers it to the target unit. This integrates its activity and possibly, emits a spike. The emitting unit computes the delays and creates the corresponding spike structures that are further inserted in order in SL. The process is repeated at time $t2$ with the new configuration of the spike list.

One can see that compared with Algorithm 4 (Eq 5.4) the number of operations on Algorithm 5 is reduced by two factors. First, in each time bin only a percent a of the total number of units in the network are computed. This causes a decrease in the number of operations in the performance critical part that concerns the integration of units activities, by a factor of a^2. Second, the occurrence of the integration time instants is not fixed to a time clock, but is given by the time resolution σt and the network activity a. The number of operations in Equation 5.5 can be further decreased in two ways. Firstly, by decreasing the percent of spikes which are computed in a time slice and secondly, by increasing the σt value. Both issues are discussed in Section 5.2.2.

In order to compare the performance of the event-driven and time-driven policy, the Algorithms 4 and 5 have been implemented to train a pulsed self-organizing network with plastic lateral synapses. Note that, in the remainder of this chapter, the continuous time Algorithm 4 implemented with the activation function ACT_Spike1 will be used as the basicline measure. The network architecture and the learning procedure are described in detail in Section 6.1. In a nutshell, the organization process consists in the training a self-organizing

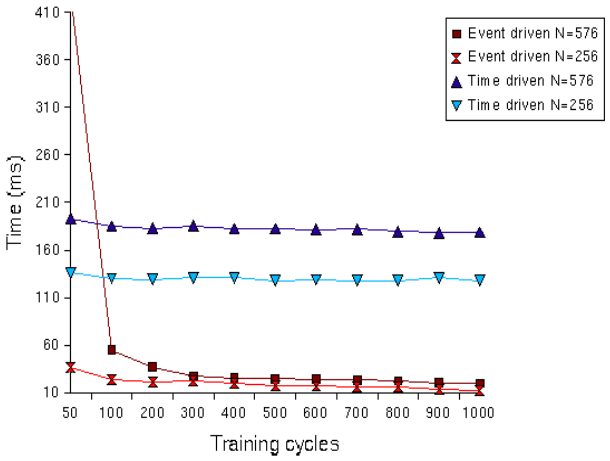


Figure 5.3: Computational times per training cycle for the event and time driven algorithms, for two network sizes N = 256, 576 units.

feature map of spiking neurons with a Mexican–Hat shape of lateral connectivity, to encode 12 directions of movement. Two networks have been used, of dimensions $N = 256$ and $N = 576$ units. The size of the test maps was limited by the low dimension of the set of input patterns (i.e., 12x20 patterns) which would cause the failure of the self-organization process in larger maps (see Section 6.1).

In a self–organization process, the lateral feedback system is used as a basic mechanism to modify over time the form of the emergent activity pattern. Given an untrained map, the neural activity starts out spreading over a large part of the network. That is in our case, up to 30%-50% of the network. In a few hundred iterations of the learning procedure, the network response to one stimulus converges to a stable activity bubble that includes a relatively small set of firing units. It becomes evident that an event-driven algorithm can benefit from the localization of the network activity, by updating units only within the active area. In contrast, a continuous time procedure computes the entire network activity in a time-stepped fashion.

Figure 5.3 shows the computational times per training cycle when the Algorithms 4 and 5 are used for the task described above. Results illustrate the good scaling of the event-driven algorithm with the simulation time, when the level of activity in the network (i.e., a value) decreases. In opposite, the time spent in the synchronous simulation (first two graphics in Figure 5.3) scales poorly with the reduction in the network activity, but it is strongly correlated with the network dimension. The scaling of event-driven algorithm is

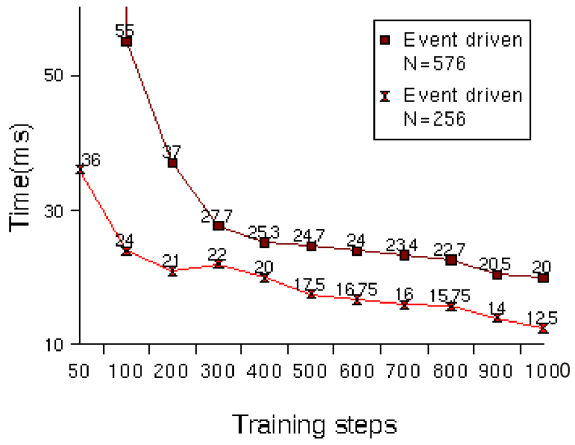


Figure 5.4: Scaling of the event-driven algorithm with the change in the network activity. Computational times per cycle are presented for first 1000 training steps, when activity decreases from 100 HZ (on first 100 steps) to 33 Hz at the end.

better illustrated with Figure 5.4. Note that the network activity decreases from 100 Hz on the first 100 cycles to 30 Hz after 1000 cycles.

These results suggest that the event-driven algorithm represents for a self-organization process with spiking neurons, a more efficient simulation framework than the time-driven approach. Time-efficiency represents a crucial issue in the simulation of self-organizing networks. That is, because the mapping accuracy of a Kohonen network depends upon the dimension of the training sample and the number of learning steps performed (Kohonen, 1995). Similarly, the simulation time of a pulsed SOM can increase significantly, unless time-efficient strategies are employed.

5.2.2 Optimization of the event–driven algorithm

On Figure 5.3 (left upper side) one notes that the event-driven algorithm applied to the training of the larger network (N= 576 units) performs more poorly than the continuous approach. This behavior is recorded during the first hundred learning cycles when the network exhibits neural activity with frequency up to 100 Hz. The inefficiency of the event-driven approach in the conditions of high neural activity arises from a poor management of the spike–event list. The main factor is represented by the insertion operation that scales with the number of events in the list. The observation about the limitations of the event-driven strategy is consistent with those of other authors (Schoenauer et al., 1998; Mattia and Del

Giudice, 2000). They have suggested that an event-driven simulation is suitable only for spiking neural networks with low activity. Most generally, the real neurons fire at low frequencies, however it is also common that they exhibit bursts of spikes with a high frequency of oscillation (Kandel et al., 2000). Therefore, we need to be able to simulate this type of neural behavior, without a tremendous increase of the simulation time. This section proposes an event-driven algorithm that scales well with the increase in the network activity.

In this section, two mechanisms are proposed for the efficient simulation of networks that exhibit activity patterns with high frequencies. Both of them address the most time-expensive process in the event-driven policy, namely the management of the event list.

Multiple spikes. A straightforward implementation of an asynchronous algorithm generates for aN firing neurons with S synapses per neuron a maximum number of $aN \cdot S$ action potentials in each time slice (Equation 5.5). Our idea was that, instead of creating a distinct event for each spike, to accumulate in a single structure all presynaptic spikes that would reach the target unit at the same time moment and to deliver them together. We define the concept of a $multiple\ spike$, which is a data structure that stores in a list, all synaptic weights m that deliver a spike to a certain neuron i at the time moment t. A similar concept was previously formulated by Schoenauer et al. (1998) and defined as weight caching.

By implementing the multiple spike concept, the overall computational load per time slice reduces by $1/m$. This means that we have fewer spikes to integrate less to distribute and less to insert in the spike list. In the most favorable scenario, m can equal S. Consequently, the time of the simulation scales well with the increase in the network activity. On the worst case, m can represent just a low percentage from S. In this case, the method does not bring a significant improvement on the time performance of the algorithm. The main parameters which affect the value of m are: the topology of local connections and the time resolution used in the generation of input signals and delay values σt. A possible trade-off might be to use a large time resolution (e.g., $\sigma t = 1$ ms). This will increase the probability of spikes to be delivered at the same time, hence it will favor their aggregation in multiple spikes and will decrease the length of the event list.

The ordered-insertion problem. The use of multiple spikes for the aggregation of pulses to be delivered at the same time to a target unit is aimed at reducing the length of the event list. However, there are cases when trying to aggregate spikes does not make any difference and there is no significant reduction of the length of the list. This faces us with the problem of how large dynamic data structures can be managed efficiently. It was mentioned above,

that the most expensive operation in the management of the event structure is represented by the ordered insertion (line 18 in Algorithm 5). That is, because the complexity of this operation scales with the length of the list. As the list grows (i.e., in the range of 10^5 events for a frequency of 100 Hz), so does the search time required by the ordered insertion. A basic improvement, which results in speeding the simulation up to four times, can be achieved if the list is searched selectively from the start or from the end, depending on the new spike time stamp.

Several implementations of event-driven strategies for the simulation of pulsed neural networks have been developed in recent years (Watts, 1994; Mattia and Del Giudice, 2000; Claverol et al., 2002; Delorme and Thorpe, in press). Perhaps one of the most efficient (and ingenious) solutions for the management of the event structure was proposed in Mattia and Del Giudice (2000). Authors suggested the use of not only one event list, but of several FIFO queues, each of them associated to a fixed axonal delay value. The neural model accounts for the existence of a discrete set D of ordered delays for spike transmission (e.g., a maximum number of 16 delays have been implemented). Synapses of the neurons in the network are organized in matrix-structured layers, each layer corresponding to one delay value. When an event is generated it is not inserted in a single global list, but it is directed to the synaptic queue corresponding to its transmission delay. Because in the same queue all spikes share the same transmission delay, the spike generated first will be the oldest (top of the queue) and the latest generated spike will be the last in the queue (end of the queue). Accordingly, the data structure needs no sorting operation and the insertion is done in $O(1)$ complexity.

However, the picture is completely different when a neural model accounts for noisy delays. That is, because a recently generated spike with an associated high transmission delay must be delivered to the target unit *later* than an older spike that has a short synaptic delay. In this case, the time stamp of an event is dependent on both the spike time and the noisy delay. The algorithm proposed by Mattia and Del Giudice (2000) has a very low complexity when used for learning with plastic synapses and theoretically, the solution suggested is very ingenious and highly efficient. Its drawback is that it employs a crucial simplification of the neural model, consisting in the limitation of the synaptic delay to a fixed number of values.

Formal neural models apply several simplifications to the detailed structure of the biological neuron (see Section 4.2.1), but noise in the synaptic transmission is preserved by almost all simplified models (Gerstner, 1999; Mass, 1999). This is, because noisy delays are an essential

factor in the optimal tuning of the neural response to the stimulus attributes, hence they play a crucial role in the development of neural selectivity and in the self-organization of the cortex (see Section 6.1). We consider that keeping the noisy delays is a *must* for a simplified neural model and the solution proposed in Mattia and Del Giudice is acceptable only for a limited set of applications, where noisiness can be discarded.

Quick sorting of an unordered pool. Our solution to an efficient management of the spike list structure consists in eliminating the insertion overhead. Instead of performing the ordered insertion, we just add the spike to an unordered pool of spikes, an operation of complexity $O(1)$. Since the events have to be processed in chronological order, at constant time intervals, the simulation engine stops processing spikes, and selects from the pool and sorts chronologically, those events which will be computed in the next interval. The time window during which processing of events takes place continuously is referred to as T_{window}. It is similar to the *safe window* concept used in parallel simulations to guarantee the temporal correctness of the algorithm (Ferscha and Tripathi, 1994). Figure 5.5 shows the simulation engine of the algorithm.

The selection of spikes to be processed in the next time window is performed using a *quick sort* algorithm. Most importantly, the sorting algorithm is run only over a small percentage q from the elements in the pool, namely those whose time mark is within the next processing interval (see Figure 5.5). This is realized by setting the first pivot point of the quick sort procedure to $t + T_{\text{window}}$. By doing this, after the first iteration of the algorithm the pool is "shuffled" and only the elements with a time stamp less than $t+T_{\text{window}}$ are sorted (lines 23-27 in Algorithm 6). As a consequence, instead of dealing with the total number of insertion operations proportional to

$$NO_6 \approx \frac{T}{\sigma t} \cdot a \cdot N \cdot S \cdot \log(\frac{T}{\sigma t} \cdot a \cdot N \cdot S) \tag{5.6}$$

we have an insertion operation of complexity $O(1)$ and an additional computational effort to sort the pool given by

$$NO_7 \approx \frac{T}{T_{\text{window}}} \cdot q \cdot N_p \cdot \log(q \cdot N_p). \tag{5.7}$$

where N_p is the number of spikes in the pool. The value of the interval T_{window} is choose by considering two aspects. On one hand, the value of this variable is set so that it ensures a correct chronological processing of the spike-events from the pool. To obtain this, each processing interval cannot be larger than the minimum synaptic transmission delay. As the value T_{window} grows, so does the risk of integrating more recent spikes before older ones.

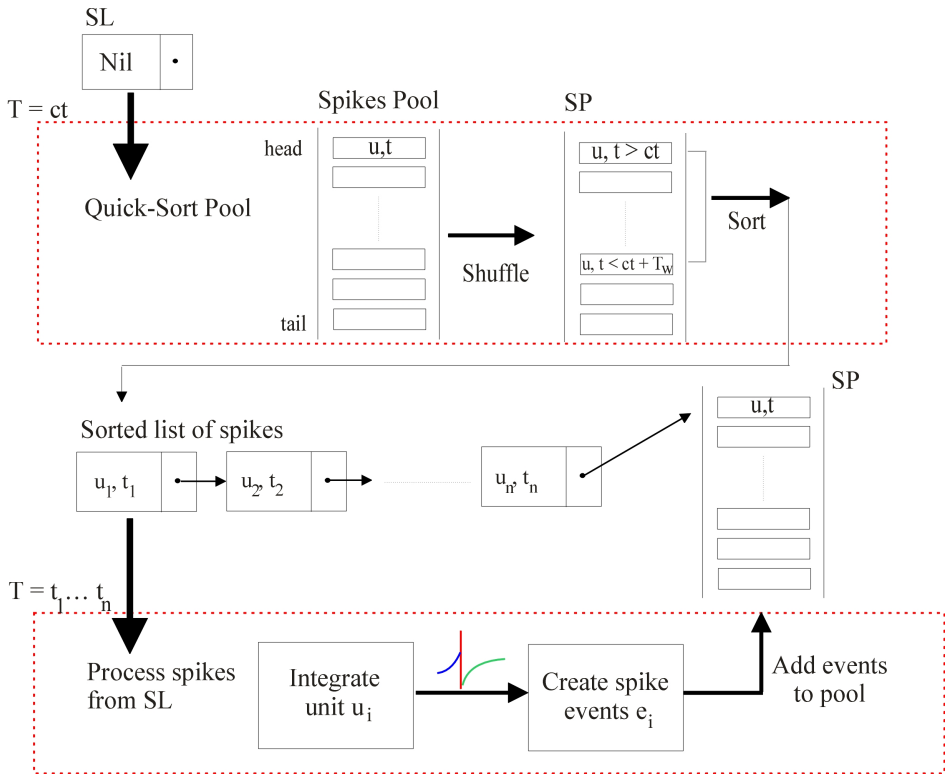


Figure 5.5: The event-driven simulation engine based on the quick-sort strategy of an unordered pool. Whenever the spike list SL becomes null, that is, when all spikes in the current time window T_{window} have been processed, the quick-sort algorithm is run. Note that it is run only upon a fraction of the elements in the spikes pool, namely those with a time stamp between the current time t and $t + T_{\text{window}}$. Sorted spikes are stored in SL and are processed by the simulation engine (i.e., integration, propagation, output). Any new spikes generated are simply added at the end of the pool.

On the other hand, too fine a value of T_{window} increases the chances that the sort algorithm will fail. That is, if the sorting interval is too small, the algorithm does not find not even a single event or it returns a very small number of events with time stamps within this interval (see lines 23-27 in Algorithm 6). Moreover, a large time window is desirable because it decreases the number of times when sorting is required. The implementation of the quick-sorting strategy and of the multiple spike concept, give the final version of the PROPAGATE_Spike function in Algorithm 6.

The implementation of Algorithm 6 for the simulation of a number of cognitive models indicates that it represents an efficient solution for networks with high frequencies of neural activity and when the synaptic delays have equally distributed values. In our simulations (Sections 6.1, 6.2), the T_{window} interval was set to 2 ms and σt to 0.1 ms. This means that

Algorithm 6 PROPAGATE_Spike - quick sorting strategy

Require: current pattern number p, simulation time st, time out t_{out}

1: $ct = st$ {set the time clock}
2: $PL = nul, SL = head(PL)$ {SL points to the head of the spike pool}
3: **while** $ct < t_{out}$ **do** {while current time less than time out}
4: **if** SL = $nul \ \vee \ ct \geq$ time(next_subpattern(p)) **then**
5: apply_inputs (next_subpattern(p)) {call lines 3-10 Algorithm 4}
6: **end if**
7: e = first_event(SL)
8: $ct = e.time$ {simulation time becomes the time of the current spike }
9: **if not** nul_event(e) **then**
10: $i = e.target$ {get the spike's target unit}
11: act^i = ACT_Spike(i, ct) {integrate activation of unit i}
12: out^i = OUT_Spike(act^i)
13: SL = remove_first_event(e, SL)
14: **if** out^i **then** {propagate spike to postsynaptic nodes}
15: **for all** k $\in \Lambda_i$ **do** {all nodes postsynaptic to i}
16: d_i^k = noisy_delay(length(w_{ik}))
17: e^m = **multiple_spike(**$k, ct + d_i^k$**)** {create or update a multiple spike}
18: PL **= add_event_pool(**e^m, PL**)**
19: **end for**
20: insert (i, LearningNodes) {synapses of firing units are subject to learning}
21: **end if**
22: **else if** $PL \neq$ nil **then** {if SL is nul and spike pool PL is not empty}
23: $T_{\text{pivot}} = ct + T_{\text{window}}$ {get the next processing window}
24: $PL\,[center] = T_{\text{pivot}}$ {set the first pivot point for quick sorting}
25: **while** $SL = nul \wedge T_{\text{window}} < time\_out$ **do**
26: SL = sort_pool(PL, T_{pivot}) {sort spikes with time less than T_{pivot}}
27: $T_{\text{pivot}} = ct + T_{\text{window}}$ {increase the window}
28: **end while**
29: **end if**
30: **end while**

the insertion operation complexity in Equation 5.6 is 20 times higher in comparison with the sorting effort from Equation 5.7. Moreover, the value of the percent p is given by the number of spikes to be sorted within a 2 ms interval, hence it is low compared to the total number of spikes in the pool. Together, these factors make the sorting computational effort to remain low and almost independent of the pool size, N_p.

Moreover, we can complete now the implementation of the activation function ACT_Spike2 (Section 5.1.3) by specifying the way a neuron integrates the presynaptic spikes. It was pointed out there, that without a receiver-oriented connectivity each spike must store the complete information needed for the computation of the weighted postsynaptic potentials

by the target unit. In the event-driven Algorithm 6, the *Activation* function is called whenever a neuron receives a multiple spike. At that very moment, when the spike reaches the unit, the postsynaptic potential (PSP) is set to the maximum value (e.g., 1 in SpikeNNS). Therefore, no other computations are needed, excepting to multiply the PSP by the connection weight, which is stored in the multiple spike structure.

With these clarifications, lines $3-6$ in function ACT_Spike2 become:

1: **for all** $w_{ji}^k \in$ MultipleSpikeStructure **do**
2: $PSP^k(t) = 1$
3: $dPSP_i(t) += PSP^k(t) \cdot w_{ij}^k$
4: **end for**

Evaluation

We conclude the description of the event-driven algorithm with the evaluation of its time-efficiency. First, we want to illustrate the gradual improvement in the performance of a basic event-driven implementation when each of the above methods is incorporated. We compare the version presented in Algorithm 4 with three improved algorithms: (1) the implementation of multiple-spikes at a time resolution $\sigma t = 0.1$ms; (2) multiple-spikes at time resolution $\sigma t = 1$ms; and (3) the quick-sort algorithm with multiple-spikes at $\sigma t = 0.1$ms (i.e., Algorithm 6). The performances of the algorithms were compared with respect to how well each of them realize the management of the spike-events structure. Our measure of the computational effort is defined by the time required to compute (e.g., integrate activities, output spikes, propagate pulses and handle the spike-list structure) a certain number of units, in this case $N_k = 1000$ units.

The findings presented in Figure 5.6 reveal a gradual increase in the algorithm performance of up to 20 times when the above strategies are added one by one to the basic event-driven implementation. Note that if the multiple spike strategy is applied to a series of events generated with a fine time resolution (i.e., 0.1 ms) the probability of spikes to accumulate is low. Consequently, it leads to an average improvement in performance by 30%. Only when the time resolution is increased to 1 ms does the method prove really efficient. The best performing algorithm consists of a combination of multiple spike strategy and quick-sorting of the unordered pool. It shows an almost linear scaling with the increase in the network activity and most importantly, this is obtained at a time resolution of 0.1 ms. The

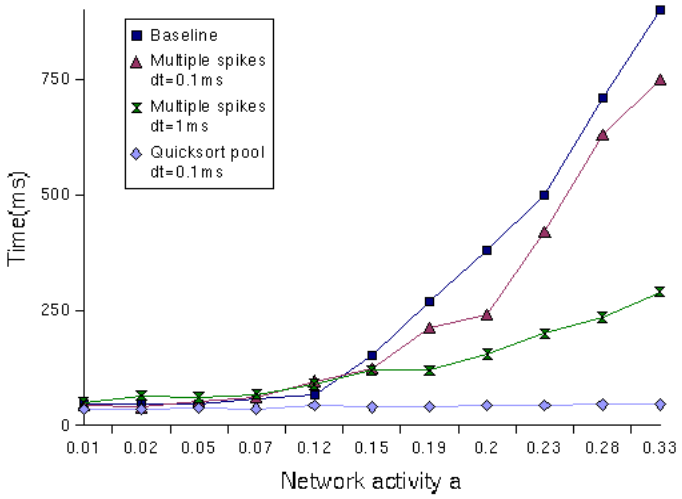


Figure 5.6: Computation times for updating 1000 units vs. levels of network activity, when different event handling methods are applied. The network activity is measured as the average number of spikes in one ms divided by the total number of neurons for N=576.

complexity of Algorithm 6 is shown in Table 5.1.

In the second part of the evaluation, we intend to compare the time performance of Algorithm 6 with the times reported in Mattia and Del Giudice (2000) (i.e., further referred in the text as the layered-delay algorithm). This comparison is facilitated by the fact that both algorithms have been applied to spike-driven learning processes of similar complexities. The quick-sorting strategy has been implemented to apply learning in a self-organization feature map of spiking neurons, with plastic excitatory and inhibitory lateral synapses (see the description of the learning procedure in Section 6.1). The learning procedure was run only for a small number of cycles while the network activity was high (i.e., 100 Hz).

Results reported for the layered-delay algorithm are the time per neuron needed to complete the simulation of one second of neural time in networks with different sizes (i.e., up to 15k units). The algorithm is applied for learning with integrate-and-fire neurons, when the spiking frequencies are maintained constant. Thus, the excitatory neurons frequency is $F_e = 2$ Hz and inhibitory neurons frequency $F_i = 4$ Hz. The connectivity rate, when the network size is varied, represents a constant percent 10% from the total number of units.

To obtain a similar measure of the computational effort required to integrate one neuron activity in our algorithm we proceed as follows. First, we calculate the number of neurons

(1) Continuous time v1.	$NO \approx \frac{T}{\Delta t} \cdot N \cdot S \cdot (\lvert F_j \rvert + a)$, with a $\in [0, 1]$
(2) Continuous time v2	$NO \approx \frac{T}{\Delta t} \cdot N \cdot S \cdot a \cdot 2$
(3) Event driven - basic	$NO \approx \frac{T}{\sigma t} \cdot N \cdot S \cdot a \cdot (a + 1 + \log(\frac{T}{\sigma t} \cdot N \cdot S \cdot a))$
(4) Event driven - optimal	$NO \approx \frac{T}{\sigma t} \cdot N \cdot \frac{S}{m} \cdot a \cdot (a + 1) + \frac{T}{T_{\text{window}}} \cdot q \cdot N_p \cdot \log(q \cdot N_p))$ with $m \in [1, S]$, $q \in [0.05, 0.25]$ and $N_p = \frac{T}{\sigma t} \cdot N \cdot \frac{S}{m} \cdot a$

Table 5.1: Computational effort for four algorithms, estimated by the number of operations required in the implementation of the performance critical parts: (1) continuous time algorithm (Algorithm 4, Section 5.2.1) with activation function version 1 (Algorithm 1, Section 5.1.2); (2) continuous time algorithm (Algorithm 4) with improved activation function version 2 (Algorithm 3, Section 5.1.3); (3) basic event-driven algorithm (Algorithm 5, Section 5.2.1); (4) time-efficient version of the event driven algorithm, with multiple spikes and quick-sorting of the pool (Algorithm 6, Section 5.2.2).

which fire during 1s of simulated time. This is given by $N_f = N_e \cdot F_e + N_i \cdot F_i$, where N_e, N_i are the number of excitatory, and respectively inhibitory neurons, in a network of a given size. Next, we run the Algorithm 6 on a set of networks with similar sizes and connectivities as the one tested in Mattia and Del Giudice (2000). During training of these networks, we compute the time per neuron required for the simulation of all operations entailed by the firing of same number of neurons, N_f. The reason for this measure of evaluation is that the two algorithms run on networks that show activities with significantly different frequencies, hence the computational efforts per time slice cannot be compared. The results of the comparison are presented in Table 5.2.

The main strength of the Mattia and Del Giudice algorithm resides in the layered structure of 4 up to 16 delay values. This was designed particularly for computational simplicity, even if it was done at the expense of the biological plausibility. Moreover, the simulation time per neuron obtained with the layered-delay algorithm (Table 5.2) is based on an average spiking frequency of 2.5 Hz. This means that in one second of simulation time, each neuron fires less than three times. In the case of the quick-sorting algorithm, time per neuron was computed when a number of N_f neurons fired at a frequency of 100 Hz. The difference in the frequencies is essential when comparing the algorithms. For instance, in the case of a network with $N = 4k$ units a spiking frequency of 100 Hz generates a pool of events of

N (1k units)	**N_f** (1k units)	**Layered delays** (ms) 2.5 Hz	**QuickSort pool** (ms) 100 Hz
0.5	1.0	0.45	1.0
1	2.4	0.8	2.5
2	5.0	1.4	3.1
3	7.0	2	4.3
4	9.5	2.7	5.2
5	12.0	3.2	6.3

Table 5.2: Execution times per neuron vs. size of the network N, when N_f neurons fire. For the layered–delays algorithm we refer to the Mattia and Del Giudice (2000) algorithm. Note that the average firing rate in the layered delays simulation is 2.5 Hz, whereas the execution times for the Quick-Sorting algorithm are recorded for a neural activity of approx. 100 Hz.

the order of $220,000$ elements. Hence, our algorithm manages to keep the simulation time approximately twice as long as the layered-delay algorithm with 30 times higher frequency. And most importantly, it still manages to preserve the essentials of neural behavior (i.e., noisy synaptic transmission).

5.2.3 Learning framework

So far in this section, we described the implementation of the PROPAGATE_ACT function that is used to apply and propagate a time-coded input pattern through the network. The final task is to implement the training function LEARN_Spike that specifies how the network learns a complete set of input patterns.

The training procedure consists of a loop that executes a series of actions. It takes a new input pattern from the set (lines 2, 12 Algorithm 7) and it applies it to the network. It propagates the network activity until a reference time $time_out$ is reached or until there are no more spikes to process (see PROPAGATE_ACT in Section 5.2.1). Then, it updates the synapses of those neurons selected for learning during the training step (i.e., the firing neurons). Learning is applied with a frequency L_f, that can be 1 or any number up to the training set dimension. Before the beginning of a new training step, the network activity can be reset or preserved, as a function of the parameter a_f. In the simulations described in chapter 6, the network activity was reset during training after the presentation of each input pattern, and

it was preserved in the testing phase.

Algorithm 7 LEARN_Spike

Require: pattern set P, learning frequency L_f
1: setup_simulation_times ($t_{start}, t_{out}, t_{int}, ct$)
2: cp = first_pattern(P) {get first pattern}
3: **while** not_nul (cp) **do** {for all input patterns}
4: PROPAGATE_ACT(cp)
5: **if** $number(cp)\,\%\,L_f = 0$ **then** {time to apply learning }
6: **for all** $i \in$ LearningNodes **do**
7: Learning_Function(i) {apply learning to the synapses of neuron i }
8: **end for**
9: **end if**
10: reset_network_activity(a_f) {reset or keep neurons activities}
11: update_simulation_times (t_{out}, t_{int}, ct)
12: cp = next_pattern(P) {get next pattern}
13: **end while**

The Learning_Function (line 7 in Algorithm 7) can implement any type of synaptic adaptation rule. Several rules for learning with spiking neural networks have been described in Section 4.4. At the present, the existent learning rules in SpikeNNS implement a self-organization learning framework used for the simulation of the cognitive models described in Sections 6.1 and 6.2. Hebbian and anti-Hebbian plasticity for excitatory synapses and a type of correlation-based learning for the inhibitory synapses have been also implemented (see Section 6.2). Future work is aimed at extending the number of learning functions with supervised rules and other forms of spike-timing dependent synaptic plasticity.

5.3 Description of simulator features

5.3.1 Stuttgart Neural Network Simulator

The scope of the modeling work carried out in this thesis is to provide an illustration of how a cognitive phenomena, such as visuomotor mapping or movement planning, can be grounded at the neural level. In Section 4.3 it was argued that a simple, rather than a detailed neural model, is more appropriate for our modeling goals. We have described the implementation of a neural model, namely the Spike Response Model (Gerstner, 1999). The reasons we did not use a preexisting simulator for pulsed neural networks and choose to build our own implementation are twofold. Firstly, the most powerful and user-friendly

simulators for spiking neurons existent today are based on detailed models of the neuron structure: Neuron (Hines and Carnevale, 1995), GENESIS (Bower and Beeman, 1998). This makes them less suitable for the simulation of large-scale cognitive phenomena. Secondly, freely available simulators of simplified neural models are restricted as applicability, since they have been created for specific modeling issues and run on certain hardware platforms. For instance, INFERNET (Sougné, 1999) is a simulation framework created particularly to explore cognitive binding and inference, and which works on Macintosh machines.

Therefore, it is rather difficult to adapt a simulation environment created to support the development of a family of models in one domain to another domain study. What seemed to be the best solution was to use a general-purpose simulator that can provide good visualization facilities and the possibility of code re-use. It was decided to take a simulation environment designed for the traditional, continuous valued neurons and to extend it to deal with the more biologically realistic spiking neurons. The simulator chosen was the Stuttgart Neural Network Simulator (available at http://www-ra.informatik.uni-tuebingen.de/SNNS/).

Stuttgart Neural Network Simulator (SNNS) is a software simulator, currently available for Unix and Windows platforms, developed since 1990 at the Institute for Parallel and Distributed High Performance Systems (IPVR) at the University of Stuttgart (Zell et al., 1992). It supports arbitrary network topologies, it is highly configurable and includes a relatively large number of learning procedures, starting with backpropagation algorithms, ART maps, Kohonen networks and ending up with time-delay and recurrent networks. The graphical user interface (Figure 5.7) offers a 2D/3D representation of the neural networks and allows an user-friendly control of the kernel during the simulation run. The sources of the implementation in C for Unix platform are freely available, and can be easily extended with user-defined libraries. SNNS is a widely distributed neural network simulator and its use and extension is technically supported by the SNNS team and by the SNNS discussion mailing list.

The advantages of using the SNNS framework for the development of our pulsed neural network libraries were twofold. First, we benefit from the substantial functionality of the simulator and second, we can make the modeling extensions publicly available to the substantial SNNS user community. However, there are also disadvantages when adapting a simulator built for processing with rate coding neurons for the simulation of spiking neurons behavior. Some of the aspects of how an efficient simulation of pulsed neural network differs from a traditional neural network simulation have been emphasized in Section 5.2.1. The main

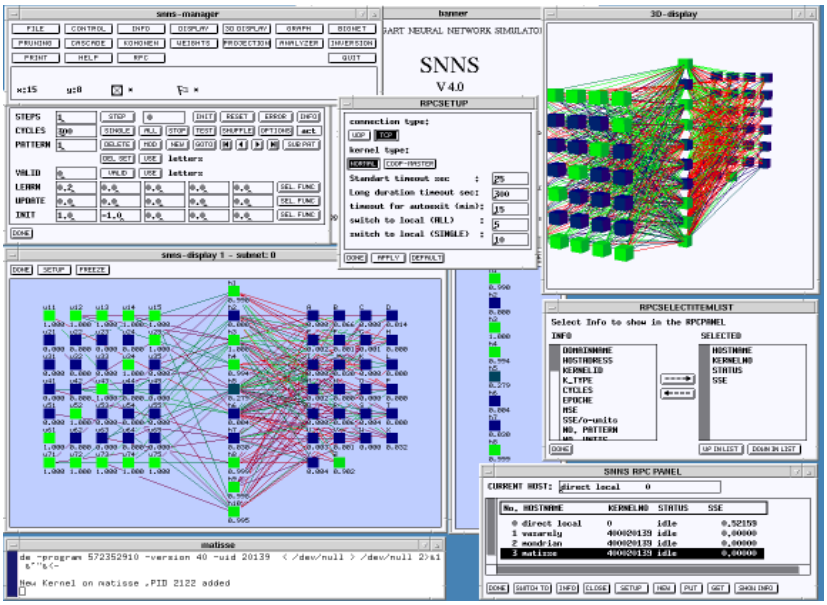


Figure 5.7: User interface (XGUI) of the Stuttgart Neural Network Simulator.

design considerations regard the type of the connectivity scheme (i.e., sender-oriented vs. receiver-oriented) and the type of simulation engine (i.e., synchronous vs. asynchronous). The first implementation of the Spike Response Model using the SNNS data structures indicated that the original kernel libraries are not very appropriate for the simulation of spiking neurons (see Sections 5.1.2, 5.2.1). Consequently, we had to re-engineer a number of functions in the SNNS kernel and create dedicated data structures suited to the specifics of computing with spiking neurons. Another aspect that we had to consider in the simulator design, was the compatibility of the spiking neural functions with the SNNS own functions. Thus, the names, parameters, return codes or functionality of the new procedures have been designed in accord with SNNS conventions. At some point, this aspect constrained the way functions were built.

The extension of the SNNS environment to support simulations with spiking neurons was named SpikeNNS. This extension needed the re-engineering of some parts of the kernel, as well as the addition of dedicated functions, in order to allow computation with spiking neurons. The library of spiking neurons functions consists of: the implementation of the Spike Response Model (SRM), the event-driven simulation engine, functions for the application of time-coded input patterns, routines for learning with spiking neurons. Additional

functions are required for initialization, testing and gathering firing statistics. The sources of SpikeNNS simulator will be freely available at http://cortex.cs.may.ie/tools/SpikeNNS.html.

5.3.2 Setting up a simulation with SpikeNNS

The process of setting up a simulation in SpikeNNS involves a number of steps:

- First, the specification of a prototype of the basic computational unit is required. The parameters of the neural model are specified from the SNNS configuration file. They are discussed below.
- Next, the connectivity of the network is created. This can be done either using the graphical interface available in SNNS for building networks, or by generating it with SpikeNNS functions. Parameters for the configuration of three types of a connectivity are given in Table 5.4 and described below.
- Finally, an initialization procedure is run to set the specific values of the spiking neurons simulation.

Initialization of neural parameters

The behavior of a spiking neuron is described by Equations 4.2, 4.3, and 4.5 (Section 4.3) and depends on a set of parameters, which can be defined by the user. These parameters are enumerated in Table 5.3 and have been described in detail in Section 4.3. The initialization of these values and other spike-specific parameters is done with the procedure INIT_Spike. At initialization, each hidden neuron in the network is assigned a noisy threshold computed by the formula:

$$\Theta = max\Theta - R(\Delta\Theta), \tag{5.8}$$

where R is a function that returns a random number uniformly distributed between 0 and $\Delta\Theta$. When a spike is emitted on a synapse, a noisy delay is generated as a function of the synapse length l:

$$d = l + G(\Delta d), \tag{5.9}$$

where l is given by the Euclidean distance between the connected neurons and G is a Gaussian distribution with mean 0 and standard deviation Δd. The firing time of a neuron is

Neural parameters
$max\Theta$ - maximum neural threshold
$\Delta\Theta$ - threshold spread value
τ_m - membrane time constant
u, v - refractory period parameters
F_j - number of spikes stored / neuron
Δd - noise factor in the generation of synaptic delays
Δf - noise in the firing time value
N_e - percent of excitatory neurons
c_e, c_i - connectivity rates of excitatory and inhibitory neurons
Δc - random factor in connections number

Table 5.3: The configurable parameters of the neural model.

affected by noise with Gaussian distribution with mean 0 and Δf standard deviation, according with the formula:

$$t^{f'} = t^f + G(\Delta f). \tag{5.10}$$

At initialization, each unit is probabilistically defined as either excitatory or inhibitory, so that an average number N_e of excitatory neurons will exist in the network. If the users chooses to implement a sparse connectivity pattern (see Section 5.3.2), than the number of lateral connections per neuron is specified by the formula

$$c = C \cdot N + G(\Delta c), \tag{5.11}$$

where N is the total number of neurons in the network and $C = c_e \vee c_i$ is the rate of connectivity per excitatory, or inhibitory neuron respectively. The Gaussian distribution has the mean 0 and standard deviation Δc.

Creation of a connectivity pattern

SNNS is a simulation environment designed for the modeling of classical neural networks. Hence, it does not address in particular, the problem of creating biologically realistic connectivity patterns. Most readily, the network topology can be set up to full connectivity between any two layers or between any clusters of units, defined by their coordinates (x, y) in the network. By contrast, the topologies of biologically inspired networks usually are

based on sparse probabilistic connections or regular synaptic patterns (see below). That is, because in the cerebral cortex each neuron is coupled to a reduced number of other neurons, in a non-random fashion (Braitenberg and Schuz, 1998). Cortical synapses within the same layer most commonly connect cells with similar neural response, while projections between layers convey information from one stage to another in a topographical manner (Kandel et al., 2000; see also Section 2.2.4).

SpikeNNS offers a number of configurable functions, which can be used to generate three types of connectivity patterns, independently of the neural model used (see parameters in Table 5.4):

- *Intra-layer sparse connectivity*, with a short-range distribution of the excitatory connections and long-range distribution of inhibitory synapses. There is experimental evidence that a Mexican-Hat shape of connectivity with inhibition surrounding excitation is biologically plausible and it favors the process of lateral and afferent weights self-organization (see Section 2.2.4).
- *Inter-layers topographical projections*. In the cerebral cortex sets of neurons which convey information from one processing stage to another are usually projecting in a topographical manner (see the retina-LGN-V1 projections, Kandel et al., 2000).
- *Afferent receptive fields*, which convey signals from a distinct area of the input space to a cortical (e.g., hidden) neuron (Kandel et al., 2000).

In SpikeNNS, connections are defined by their weight w and the synaptic length l that determines the value of the spike transmission delay (see Equation 5.9). The synapse weight is computed by the formula:

$$w = max\, w - R(\Delta w), \tag{5.12}$$

where R is a function that returns a random number uniformly distributed between 0 and Δw.

Sparse connectivity. To create within a target layer n, a sparse, centered-surround connectivity pattern, a number of steps are involved (see parameters in Table 5.4). First, for every neuron i in n, the probability of being connected with any other neuron j in the same layer is computed. The probability to create a synapse depends on the sign of the source neuron (i.e., excitatory or inhibitory neuron), the distance to the target neuron and the connectivity

Type	Connectivity parameters
Sparse intra-layer connectivity	n - target layer $max\ w$, Δw - maximum weight and deviance value σ_e, σ_i - connectivity decay parameters
Topographical inter-layers projections	s, t - source and target layer $max\ w$, Δw - maximum weight and deviance value L_p - sets the size of projection for topographical connections bi - sets on the bidirectional connectivity between s and t
Receptive afferent fields	i, t - input and target layer $max\ w$, Δw - maximum weight and deviance value $d_i, \Delta d$ - delay and deviance value for input connections R_f - sets the radius of the receptive field m - sets the value of the magnification factor

Table 5.4: The configurable parameters for three types of connectivity: intra–layer, inter–layers, and receptive fields.

decay parameter. Thus, an excitatory synapse from neuron i to neuron j has the creation probability of:

$$p = e^{\frac{-d[u_i,u_j]}{\sigma_e}}, \tag{5.13}$$

where $d[u_i, u_j]$ is the Euclidean distance between the units and σ_e is a constant which controls how fast the connectivity decays with the distance in the layer (formula adapted after Ström, 1997). An inhibitory connection is added with a probability of

$$p = 1 - e^{\frac{-\sigma_i}{d[u_i,u_j]}}. \tag{5.14}$$

After the probabilities of all possible output connections of the neuron i have been generated, these values are sorted in a descending order in an array. The array is then parsed starting from the highest values and an output synapse is created if the associated probability is less than a random number r. The value of this number is uniformly distributed within an interval, whose limits are set differently, depending on the sign of the synapse $r = base_{i,e} - std.dev_{i,e}$. Synapses are added to the network topology until the number of connections per neuron c (Equation 5.11) is reached.

Figure 5.8 shows for two target units at locations (4,12) and (13,6) in the network, the presynaptic units in the layer. The excitatory units (shown in blue) are distributed (probabilistically) within an area of dimension four surrounding the target unit. The long-range in-

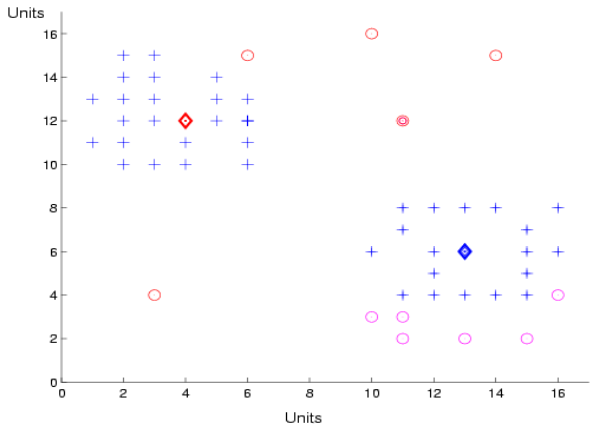


Figure 5.8: Sparse probabilistic connectivity pattern in SpikeNNS, which a short-range distribution of excitation and long-range distribution of inhibition. For two neurons, at locations (4,12) and (13,6) the input units are shown. The blue signs designate excitatory presynaptic units. The inhibitory units are shown in red (or magenta). The connectivity decay parameter is $\sigma = 4$.

hibitory synapses are received from units located at distances greater then four, up to the network margins. The percent of excitatory units in the network is of 80%. Parameters used in the generation of this pattern, were $\sigma_e = 4$, $\sigma_i = 3.5$. Random values were generated using $base_e = 1$ and $std.dev_e = 0.3$ for excitatory synapses, and $base_i = 0.7$ and $std.dev_i = 0.4$ for inhibitory synapses.

Topographical projections. In SpikeNNS, topographical projections can be created only between hidden layers (see parameters in Table 5.4). Each neuron in the source layer s projects in the target layer t in a sub-array of units of size L_p. Note that current implementation of the function works only for dimensions of the target layer that are equals or larger than the source layer dimension. If bi parameter is set on (i..e, $bi = 1$), than reciprocal projections between layers s, t are created.

Receptive fields. Receptive fields can be created from an input layer i to any hidden layer t (see Table 5.4 for parameters). In such a connectivity scheme, a hidden unit of coordinates (x, y) receives stimulation from a square area of size R_f that is centered in the input layer (approximately) at $(\frac{x}{\sqrt{m}}, \frac{y}{\sqrt{m}})$. The value of the magnification factor m gives the ratio of hidden units per one input unit. That is, because the hidden layer has a higher dimension then that of the input layer and this allows the formation of several receptive fields for each location in the input array.

Implementation of the functions described above is independent of the type of the neural model specified (i.e., rate-coding or spiking neuron). Hence, they can be used to create network topologies, when any activation or output functions from SNNS are employed. Note that each of the above functions deletes any previously existing connectivity between the layers given as parameters. Therefore, they cannot be used in conjunction with the SNNS graphical interface for setting intra- or inter-layer connectivity. All parameters for the initialization of the neural model the setting of the connectivity patterns and for learning can specified in the SNNS configuration file. The SNNS control panel allows on-line modification of at most five parameters. When a particular function is loaded, such as ACT_Spike, LEARN_Spike or INIT_Spike, all its parameters are initialized from the configuration file. Only the first five values are displayed and can be updated through the control panel.

5.3.3 Learning parameters

As noted above, the parameters for learning are set in the configuration file and initialized during the call of the function INIT_Spike. Training of a pulsed neural network in SpikeNNS is performed using Algorithm 6 (Section 5.2.2), which represents our most efficient version of an event-driven strategy. The general learning framework was described in Section 5.2.3. Three timing parameters have to be set for the LEARN_Spike function: an absolute value of the simulation start time t_{start}, an arbitrary value of the integration time moment t_{int}, and the time out limit t_{out}, until which the network activity is computed during a training step. For a description of these parameters significance see Section 4.3.4. Other parameters which have to be set for the training of a pulsed neural network concerns: the look-ahead value for the event-driven algorithm T_{window}, the flag a_f, which tells if the network activity is reset after each pattern propagation, and the learning frequency L_f (see description of parameters in Section 5.2.3).

During training and testing of a network, a number of statistics with respect to the neurons firing behavior can be collected and saved in a results file. This information concerns mainly the spike timings for each neuron in the hidden layers and the discharge rates during each input pattern application. The extension of the SNNS simulator to support modeling of spiking neural networks is only at its beginning and more work will be required in order to provide an integrated new version of SNNS.