NTNU – Trondheim
Norwegian University of
Science and Technology

Development and assessment of a novel model for artificial neural networks.

Per Roald Leikanger

Master of Science in Engineering Cybernetics
Submission date: May 2012
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Development and Assessment of a Novel Model for Neural Simulation

Per R. Leikanger

May 8, 2012

Abstract

When simulating a spiking neuron, numerical integration of synaptic input is often utilized to compute the neuron's depolarization. This report shows that the Numerical Integration Model(NIM) for spiking neuron simulations have a cumulative error that diverges unless the expectancy value for the local truncation error is zero. An alternative neuron simulation scheme, κM, was developed and is presented in this text. Experimental and theoretical results shows that the κM error varies within a bounded domain.

Experiments have been conducted on sample–and–hold implementations of the two models. A κM_{100} simulation, a κM simulation with 100 iterations per forcing function period, was compared with NIM simulations with finer temporal resolutions. It is shown that before 15 periods of a sinusoidal depolarizing input current has been simulated, the κM_{100} simulation produced a smaller error than a $NIM_{10.000}$ simulation. Since the NIM simulation has a number of time steps that is two orders of magnitude larger than the κM simulation, this represents a significant efficiency improvement.

Contents

List of Figures

Chapter 1

Introduction

While the digital computer processes information by algorithms, networks of neurons can be said to process information by pattern recognition [30]. Thus, the two computational systems utilize different computational schemes, with different capabilities and limitations. Each can, however, emulate the computational scheme of the other to accomplish certain tasks. By classification of input and producing output based on previously learned patterns, biological neural systems are capable of performing algorithmic tasks. Digital systems are likewise able to emulate neural abilities by simulating networks of neurons. This is referred to as Artificial Neural Networks(ANNs) and is an example of bionics, technology inspired by nature.

Neurons propagate information by discrete output transmissions. An action potential is initialized after the depolarization of the neuron, defined as a leaky integral of input, goes beyond the *firing threshold* [1]. When the action potential reaches a synapse, a synaptic transmission causes the postsynaptic neuron to be depolarized or hyperpolarized, depending on the synapse. The size of a transmission does not vary with the magnitude of the neuron's input, but can to a certain extent be defined by the strength of the synaptic connection alone[14]. It is

debated whether neurons propagate information mainly as the action potential frequency ("the firing frequency"), or if the exact timing of spikes also is important[4, 5, 9, 30]. The main branch of ANN technology models information propagation as a floating point number, defined by the neuron's immediate input[31]. These ANNs can be said to simulate the neuron in the frequency domain, where the floating point number represents the firing frequency of the neuron[22, 31].

Simulating the neuron in the frequency domain is a major simplification of the system, and all information about timing is lost. Such models can therefore not be used for exact simulations of the neuron or where the relative spike time of neurons is important[5, 19]. An element of particular importance is synaptic plasticity, seen as the foundations of learning and memory in neuroscience[2, 15, 19–22]. In frequency based ANNs, local learning rules are defined by the presynaptic and postsynaptic neuron's firing frequency, r'_j and r_i.

$$\Delta\omega_{ij} = Cr_i r'_j \quad , \qquad \begin{array}{l} r'_j \quad \text{is the presynaptic neuron' s firing frequency} \\ r_i \quad \text{is the receiving neuron's firing frequency} \\ \omega_{ij} \quad \text{it the synaptic weight between neuron } j \text{ and } i \\ C > 0 \end{array} \tag{1.1}$$

where $\omega_{i,j}$ represents the magnitude of the synaptic connection between neuron j and neuron i [31]. This is a mathematical interpretation of what is referred to as "Hebbian learning" after Donald A. Hebb who first proposed this mechanism. *Hebb's postulate* states:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased. [8]

Both Hebb's postulate and the mathematical interpretation presented in equation (1.1) describes a monotonic increase in synaptic

weight and is obviously unstable; any correlation between the two neurons' firing frequencies makes the connection between them stronger, thus increasing the correlation. Numerous attempts have been made to develop stable learning rules in frequency based ANNs, with still increasing complexity(See e.g. [10, 28, 29]).

In 1987, Gustafsson et al. proposed that the increase in synaptic weight after a transmission varies with the postsynaptic neurons's depolarization[6]. At about the same time, Levy and Steward found that synaptic transmission could cause *Long–Term Depression*(LTD), a decrease in synaptic weight, after a single transmission[24]. These two findings explain how it is possible with a graded synaptic plasticity ranging from negative to positive weight change after a transmission. The resulting learning rule has later been referred to as *Spike–Timing Dependent Plasticity*(STDP), as the postsynaptic depolarization at the time of transmission often correlates with the relative spike time of the pre– and postsynaptic neuron[33]. The biological background for STDP is discussed in sec. 2.1.3.

ANNs that operate in the frequency domain do not contain any information about timing, and STDP can not be used. Possible temporal elements in signal processing are also lost. A networks of nodes that emulate neurons in the time domain, by simulating the depolarization of the neuron, propagate information as spikes. Such ANNs are therefore referred to as Spiking Artificial Neural Networks (SANNs)[5].

Despite the listed advantages, SANN is not often used for practical applications. The main reason might be the computational complexity of these simulations and the chaotic nature of neural networks. Small errors in each node can have large effects on how the whole network behaves. It is shown in section 4.4.2 that the error in simulations that utilize numerical integration becomes large unless the computational time step is very small, or the simulated period is short. In an attempt to avoid a cumulative error, a simulation model that utilizes the algebraic equation for the neuron's depolarization was developed

and is presented in this report. The novel model is compared to the existing neuron simulation scheme, to assess whether the model is an improvement with respect to the total simulation error.

This report is written for an audience with some knowledge from cybernetics and systems theory, as well as those who study the field of computational neuroscience. Parts of chapter 4 requires some knowledge of object oriented programming. These parts can be considered as documentation for *auroSim*, implemented in C++ and distributed under GPL[23] , and may be ommited. A deep insight into the mechanisms of biological neural systems is not required, as the most important aspects for signal processing are introduced in chapter 2.

Chapter 3 gives an overview of mathematics and concepts used in the novel spiking neuron simulation model, κM. The system's value equation is found by solving the differential equation for the *Leaky Integrate–and–Fire*(LIF) neuron's depolarization. The novel simulation scheme can also be used for simulating other neuron models, by utilizing this model's value equation instead of the LIF model's.

In chapter 4, the design of software intended for a theoretical comparison of the two spiking neuron simulation schemes is presented. The design and implementation is done in such a way that as many elements as possible are common between the two implementations, in order to make principal differences prominent. The resulting software is referred to as *auroSim* in this text.

Chapter 5 presents two experiments for *auroSim*. One considers an idealized situation, and is intended as a test of the design and error analysis of the two simulation models. The second experiment considers a more complex input pattern, defined by a sinusoidal activation level. This can for example represent an applied stimuli through an electrode or *Local Field Potential Oscillations* of cortical neurons. The results from the experiment is used in an efficiency comparison of the two models.

The report concludes with a discussion of the differences in design,

implementation, error mechanisms and efficiency of the two models. Most chapters end with a discussion of the presented elements, and the final chapter can be seen as a conclusion of all the chapters' summaries. The interested reader can find some additional information in the appendix. These elements are excluded from the main text to increase readability.

Chapter 2

Background Theory

The biological brain can be thought of as the computational system of an animal. It receives information from sensors located at various locations in the body and sends output to its various manipulators. This includes e.g. muscles and the being's endocrine system.

A biological computational system is fundamentally different from digital technology. Instead of having a few, computationally powerful processing units, the biological brain has a huge amount of weak processing units, called neurons. The neuron processes information by doing a leaky integration of input, and sending output when the value goes beyond some threshold[30]. Large networks of such cells, with dynamic connections between them, comprise the biological brain and is seen as the basis of memory, thought and intelligence.

Biological neural systems can in some respects outperform the digital computer. Tasks that involve associative computations or learning are performed much better by a neural network than by algorithms. An example of this is the pattern recognition of a two week old baby that recognizes the mother's face, a task that only recently has been accomplished by digital computational systems. In the computer, this can be accomplished by an Artificial Neural Network, ANN.

Before the mechanisms of neural simulators and ANNs can be discussed, the original system has to be introduced. This chapter is reserved for this purpose, and starts by introducing the most important aspects of neural signal processing mechanisms. After the biological neuron has been introduced, a short review of the history of ANNs is presented. This section concludes with introducing Spiking Artificial Neural Networks, neural network simulators with nodes where the neuron's depolarization is considered.

2.1 Biological Neural Systems

In the late 1800s, Camillo Golgi developed a way of staining nervous tissue so that complex networks became apparent in nervous tissue. Santiago Ramòn y Cajal used Golgi's technique in such a way that individual neurons could be separated, and it was observed that nervous tissue was not a continuous web but a network of discrete cells. Ramòn y Cajal proposed what has later been known as the neuron doctrine; that the computational capabilities of the brain comes from a network of individual "brain cells" that process incoming transmissions and sends output when its input history has a certain pattern. For their contribution, Ramòn y Cajal and Golgi shared the 1906 Nobel's price in Physiology and Medicine [2, 11].

Modern neuroscience follows the neuron doctrine. Each node in a neural network is called a neuron, and the connection between neurons are called synapses. When the presynaptic neuron "fires" an action potential, the following synaptic transmission cause the postsynaptic neuron to become excited or inhibited. Transmissions in excitatory synapses increase the postsynaptic membrane potential, causing that neuron to approach firing. Inhibitory transmissions does the opposite, and inhibits the postsynaptic neuron with respect to firing. Firing of an action potential causes transmission in all the neuron's output

synapses, and a resetting of the depolarization to the reset potential v_r [1, 11, 13].

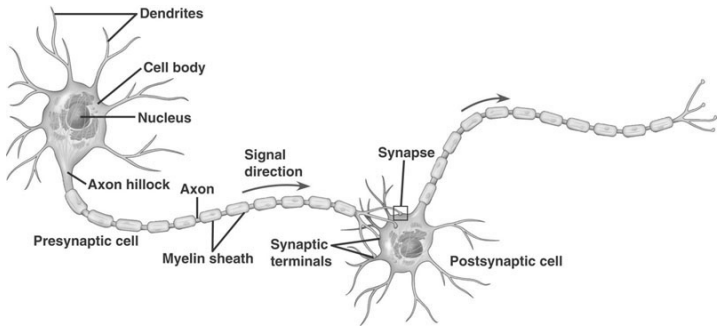


Figure 2.1: *Illustrative model of the neuron. The signal propagation goes from left to right in this figure; Synaptic integration at the dendrites, action potential through the axon and finally transmission through the neuron's output synapses. (Figure from* http://biomedicalengineering.yolasite.com/neurons.php*)*

In this section, the most important elements of neural signal processing are presented, enabling the reader to become more familiar with how neural networks process information.

2.1.1 The Neuron

In terms from graph theory, a biological neural network is a directed, cyclic graph. The nodes are called neurons and the edges between nodes are called synapses. In addition to the synapse, the neuron contains some elements that are fundamental for signal processing. The most important element is the neuronal membrane.

Each neuron is surrounded by a phospholipid bilayer cell membrane with a low permeability to ions, enabling a different concentration of ions over the membrane. All neuron membranes have ionic pumps dedicated to uphold an ionic concentration gradient over the

membrane. Different ionic pumps push the corresponding ions "upstream" in relation to the ionic concentration gradient, resulting in an electrochemical potential over the membrane. The resting membrane potential of a neuron generally lies at about $-65mV$ [2, 16].

When specialized ionic gates permeable to certain ions are opened, these ions can flow freely through the gate. Depending on which ions are let thought, the neuron membrane is either hyperpolarized(more negative membrane potential) or depolarized(more positive membrane potential). When the membrane potential becomes more positive than the firing threshold of the neuron, an action potential is initiated at the axon hillock, the base of the neuron's axon. [1, 14, 17].

2.1.2 The Axon and the Action Potential

Voltage–gated sodium and potassium channels are located along the membrane of the axon. If the membrane potential is more positive than the "firing threshold" of the neuron, these channels open, causing the membrane to have a transient positive increase in membrane potential. Through passive transmission of the electrical charge, due to diffusion of ions, the membrane potential at the next site of voltage gated channels becomes more positive than the firing threshold and the process is repeated. The size of the signal arriving at the synapse, at the distal end on the axon, is therefore independent of the total distance traveled [17].

The two most important voltage gated channels for the action potential is the sodium and the potassium channels. The Na^{2+} channel is most responsive and opens and closes faster than the K^+ channel. The highest concentration of Na^{2+} ions is on the outside of the neuron, resulting in an inflow of positively charged ions that depolarize the neuron. The potassium ion has the highest concentration inside the cell, and activation of the K^+ channel cause a flow of positively charged ions out of the cell, repolarizing the neuron. Because the K^+

channel is slightly less responsive than the Na^{2+} channel, and that both channels only stay open for a short while, the transient membrane potential of the action potential has the form shown in fig. 2.2 [17].

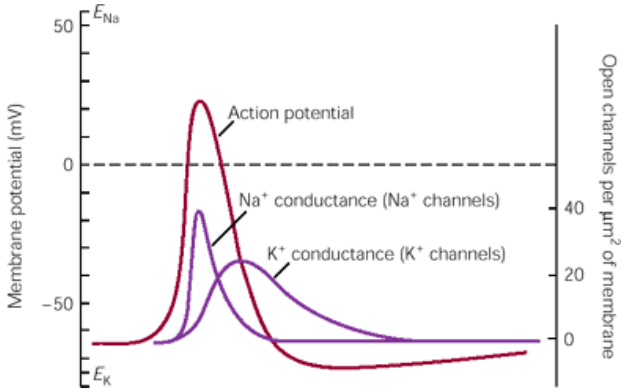


Figure 2.2: *The action potential. Activation of the Na^{2+} channel cause positively charged ions to flow into the neuron, depolarizing the neuron. The slower K^+ channel has the opposite effect. Both channels close after a short while, and the membrane potential returns to the resting value after a small undershoot [17].*

After a successful opening of the voltage–gated channels in the axon membrane, internal mechanisms close the ion channels after a little while. The channels stay closed long enough to enable the active sodium–potassium pump to reestablish some of the ionic concentration gradient over the membrane. During this time, it is impossible to elicit a new action potential. This time interval is called the absolute refraction time for the neuron, and is important to prevent the action potential form "travelling back" along the axon[2]. as well as limiting the maximal firing frequency of the neuron(see appendix A.2).

An important part of the active propagation of the action potential is that the signal is independent of the distance travelled. Because of this, the synapses located at various locations along the axon re-

ceives the same transmission–initiating signal. The importance of this becomes clear when the mechanisms of synaptic transmissions are introduced.

2.1.3 The Synapse

When the action potential reaches an axon terminal, voltage–gated Ca^{2+} channels in the active zone of the terminal opens and Ca^{2+} enters the cytosol of the axon terminal. The axon terminal contains bag–like organelles called synaptic vesicles filled with different neurotransmitters. Free intracellular Ca^{2+} cause these organelles to be pulled toward the neuron membrane. The synaptic vesicles fuse into the neuron membrane when close enough, causing its content to be released into the synaptic cleft on the outside of the membrane. The neurotransmitters diffuse out in the fluid of the synaptic cleft, and some come in contact with postsynaptic receptors. When the right neurotransmitter bind to a specific group of receptors, called ligand–gated channels, an ionic channel is opened in the postsynaptic membrane. Depending on the channel(and thus the ions that are let through), this can either depolarize(excite) or hyperpolarize(inhibit) the postsynaptic neuron[14].

The *N-methyl-D-aspartic acid*($NMDA$) receptors are of a particular importance for learning in biological neural systems[34]. As opposed to *non-NMDA receptors*, these channels enable Ca^{2+} ions to flow into the neuron. This is thought to take part in regulating the synthesis of new $AMPA$ receptors, and is considered important for synaptic plasticity as well as transmission[26]. Because $NMDA$ channels are blocked by a Mg^{2+} ion covering the opening, the membrane potential has to be sufficiently depolarized to remove this block and let ions through[12]. Due to the number of $NMDA$ receptors and variations around the Mg^{2+} blocks, this creates a graded magnitude of the Ca^{2+} inflow and thus a graded synaptic plasticity. Synaptic plas-

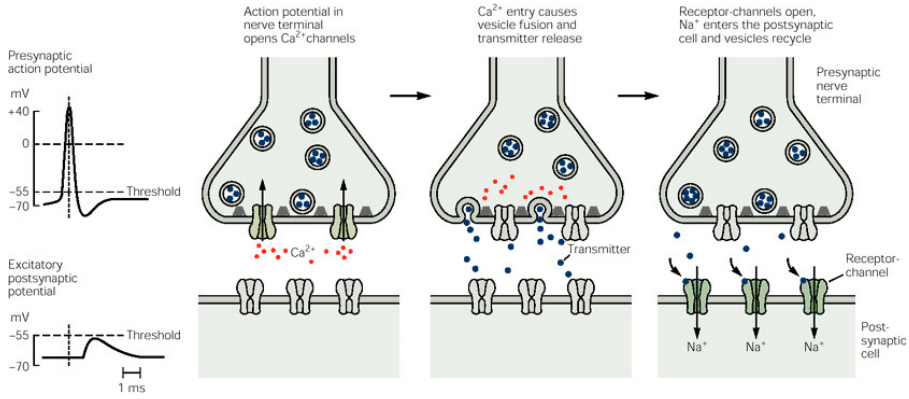


Figure 2.3: *Synaptic transmission in an excitatory synapse; An action potential arriving at the terminal of the presynaptic axon enables Ca^{2+} to enter the presynaptic cytosol, causing synaptic vesicles to fuse with the membrane. The containing neurotransmitters are released into the synaptic cleft. These diffuse passively across the synaptic cleft and bind to transmittor–specific receptors in the postsynaptic membrane [14].*

ticity can be modelled as a function of the postsynaptic membrane potential at the time of transmission[12, 20, 33]. Since this is closely linked with the relative time of firing for the pre– and postsynaptic neurons, this mechanism is referred to as Spike Timing Dependent Plasticity(STDP).

2.2 Artificial Neural Systems : A Review of ANN History

The pragmatic use of neural network simulations started with the "McCulloch–Pitts neuron" in 1943. Warren McCulloch, an early neuroscientist, and the young mathematician Walter Pitts initiated a formalized discussion about the mechanics of the neuron and the use of

neuron simulations in technology. This resulted in the first neuron emulator(artificial neuron). Artificial Neural Networks based on the McCulloch–Pitts neuron model has later been referred to as the first generation ANN[25]. Each node is modelled as a boolean device(with an on–off response), where the node sends output if the immediate level of input is large enough. The first generation ANN can therefore be said to be a network of simple threshold gates, and does not take into consideration the depolarization state of each node. One famous example of a first generation ANN is Rosenblatt's *Perceptron*[7].

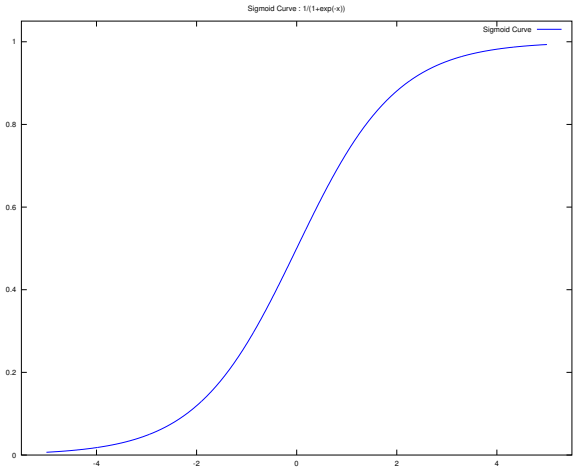


Figure 2.4: *Sigmoid curve* $\frac{1}{1+e^{-x}}$*, often used as the activation funtion in second generation ANNs.*

A better simulation of the neuron is done in the nodes of a second generation ANN[19, 25]. Each node computes the output level as a floating point number based on the immediate level of input to the node. From sec. 2.1, we have that the biological neuron sends discrete output pulses when its depolarization level goes above the fir-

ing threshold. A continuous propagation of a floating point number can therefore be said to represent the frequency of such transmissions as a function of the present input. The function used for computing the output is referred to as the *activation function* of the node, and is found to give the best results if it is given by the continuously differentiable sigmoid function[7].

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

As the biological neuron has a state, defined by the depolarization value of the neuron, the stateless computation of the McCulloch–Pitts neuron is a gross simplification of the original system. It is more correct to consider the neuron as being stateless in the frequency domain, and the stateless computation of a second generation ANN can therefore be said to be more correct than in a first generation ANN[22, 25]. Since the concept of mean frequency only makes sense for time intervals of a certain size, precise simulations with small computational time steps does not necessarily give accurate simulation results for an ANN of the second generation. For more precise simulations of the neuron or simulations where temporal mechanisms are important, one has to simulate the neuron in the time domain by considering its depolarization.

2.3 Spiking Artificial Neural Networks

A node in a $SANN$ simulates the neuron with respect to its depolarization. When the depolarization value crosses the firing threshold, a spike is initiated and the signal is propagated in the neural network. Many formal spiking neuron models exist, where the most common is the LIF neuron model[3].

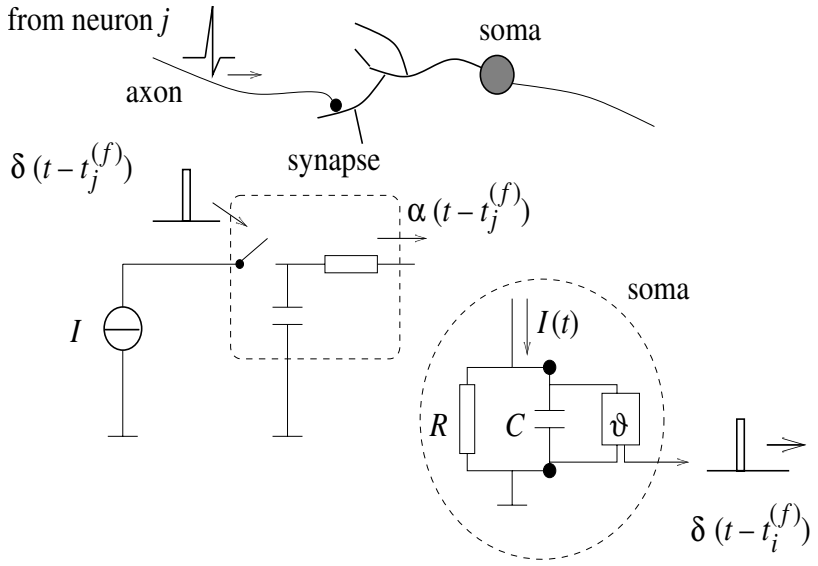


Figure 2.5: *A schematic diagram of the LIF neuron model. Each node can be modelled as the circuit inside the dashed circle on the right–hand side of the figure. Depolarizing input is represented by the input current $I(t)$, and when the potential over the capacitor is larger than the firing threshold at time $t_i^{(f)}$, a spike $\delta(t-t_i^{(f)})$ is generated. The left–hand side of the figure shows a model of synaptic transmissions as a low–pass filtering of the presynaptic action potentials $\delta(t_j^{(t)})$, generating a input current $\alpha(t-t_j^{(f)})$ to neuron i [5].*

The LIF model is a simple phenomenological model of the biological neuron, and is highly popular due to its simplicity. The leaky integration of the LIF neuron model can be modelled by the electrical circuit shown in fig. 2.5. When the membrane potential $v(t)$ crosses the neuron's firing threshold τ, a spike is initiated and transmissions through all the neuron's output synapses is the result. The neuron's membrane potential is then reset to the resting membrane potential $v_r < \tau$ [5].

$$\lim_{t \to t^{(f)}; t > t^{(f)}} v(t) = v_r \tag{2.2}$$

To the author's knowledge, the LIF neuron has previously only been simulated by numerical integration in the computer. The depolarization is integrated numerically by summing all synaptic input transmissions, and subtracting the leakage after each computational time step. If the efficiency of the synapse from neuron j to neuron i is modelled by ω_{ij}, the total input current during a time interval can be found by

$$I_i(t_n) = \sum_j \omega_{ij} \sum_f a(t_n - t_j^{(f)}) + \xi_i(t_n) \tag{2.3}$$

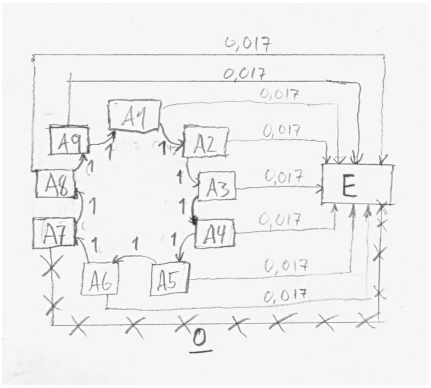
where $a(t - t_j^{(f)})$ is defined by the neuron's synaptic input currents and $\xi_i(t_n)$ comes from other sources of depolarizing input[3]. Examples of this can be a current inserted through a probe, or the sensed signal of a sensory neuron.

Leakage can be simulated by subtracting a fraction of the present membrane potential every time step. Because computational resources are limited, a finite temporal resolution(discrete time) is utilized. This involves discrete steps in simulation time, and the previously computed depolarization level has to be used to find the leakage $l(t_n)$. This introduces a delay, with the size of the computational time step, and

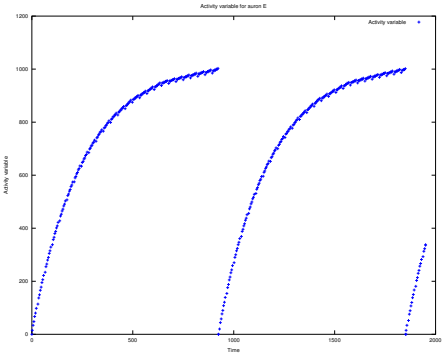
establishes an error source for the simulation.

$$l(t_n) = \alpha v(t_{n-1}) \tag{2.4}$$

Thus, the discretization of time introduces an error defined by the computational time step — if the size of the computational time step is increased, the simulation error becomes larger. More accurate simulations can therefore be accomplished by making the computational time steps smaller, but this simultaneously increase the computational load of the simulation.



(a) *Model of Spiking ANN Connections*



(b) *Depolarisation Time Course*

Figure 2.6: *(2.6a) A schematic model of the synaptic connections in an artificial neural circuit intended to illustrate neural integration. The synaptic connections in fig. 2.6a are represented as a factor of the firing threshold. A single transmission through a synapse with* $\omega_{ij} = 1$ *therefore cause the postsynaptic artificial neuron ("auron") to fire. Thus, the neural circuit* $[A1, A9]$ *is self sustaining, and causes synaptic transmissions through the synapse from auron* $[A*]$ *to auron* E*. (2.6b) The resulting depolarization curve for auron* E*. Every auron but* $A7$ *is connected to auron* E*, making the effect of leakage prominent. Note the small decrease in auron* E*'s depolarization, every ninth time step. (Figure 2.6b is generated by* $auroSim_N$*, the part of* auroSim *with numerical integration) [22]*

Chapter 3

Neural Modelling

Numerical integration involves an accumulation of error, since the error is integrated alongside the considered variable. A simulator based on numerical integration therefore involves an accumulation of local truncation errors, the error from each computational time step. In an attempt to avoid a diverging error, a neural simulator based on algebraic equations is developed and presented in this chapter.

To utilize continuous equations in a neural simulator, it is found in the preliminary project that depolarizing input has to be represented as a continuous flow [22]. After the algebraic solution to the LIF neuron's depolarization has been presented in sec. 3.1.1 and 3.1.2, the concept of synaptic transmission as a continuous flow is discussed in sec. 3.1.3.

3.1 The κ Formalism for Neural Activity

One system that behaves like a leaky integrator is a bucket with a set of small holes at the bottom. If the LIF neuron is visualized as a leaky bucket with input from a gutter, excitatory synaptic input can be represented by an agent pouring cups of water into that gutter. When the number of agents pouring water into the gutter becomes very large and the size of each transmission is small, this can again be visualized as rain.

The resulting water level in the leaky bucket can either be simulated by counting the number of raindrops(and computing the size of the leakage in every computational time step) or by estimating the corresponding flow through the gutter and utilizing the algebraic solution to the system's differential equation to find the water level. For simulations with a bounded temporal resolution(discrete time), more accurate simulations can be achieved by utilizing an algebraic equation than by numerical integration. This is tested in chapter 5. In this section, the mathematics and necessary concepts of a flow simulation is presented.

3.1.1 Algebraic Solution for the LIF Neuron's Depolarization

Subthreshold integration in the LIF neuron is defined by the general leaky integrator's differential equation[5]:

$$\begin{aligned} \dot{v}(t) &= \dot{v}_{in}(t) - \dot{v}_{out}(t) \\ &= I(t) - \alpha v(t) \end{aligned} \tag{3.1}$$

The inflow is represented by $\dot{v}_{in}(t) = I(t)$, and "leakage" is represented by $\dot{v}_{out}(t) = \alpha v(t)$. The algebraic solution to 3.1 is derived in appendix A.1. For time intervals where κ and α are constant, it is found that

the system's subthreshold depolarization is defined by

$$v(t_v) = \kappa - (\kappa - v_0)\, e^{-at_v} \quad , \; \kappa = \frac{I}{\alpha} \tag{3.2}$$

The variable v_0 represents the neuron's initial depolarization value and t_v represents the time from the start of the considered time interval($t_v = t - t_0$). Recall that equation 3.2 is only valid for time intervals where κ and α

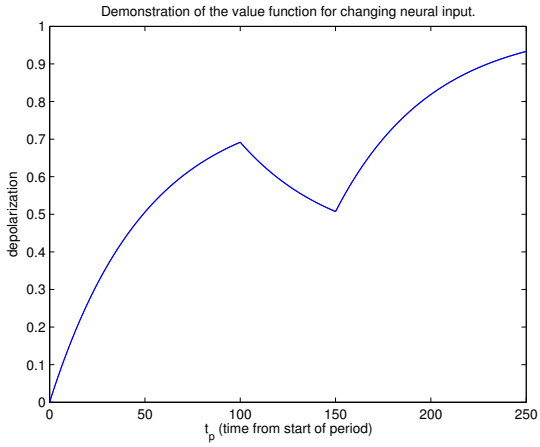


Figure 3.1: *The figure shows how the concept of time windows enables the use of* (3.2) *for simulating the neuron's depolarization. In the time interval* $t_p = [0, 100]$, $\kappa_0 = 0.7$ *is valid. At time* $t_p = 100$, κ *is changed to* $\kappa_1 = 0.5$, *before being set to* $\kappa_2 = 1$ *at time* $t_p = 150$. *[22]*

remain constant. To formalize such an interval for later discussions, the concept of time windows is introduced.

Definition 1. *A time window is a time interval where* κ *and* α *are constants, within one inter–spike period.*

When the neuron's input flow is changed or the neuron fires an action potential, a new time window is initialized. The initial value v_0

can be found by computing the last value of the previous time window, and t_0 is acquired by saving the time of initiation for the new time window.

3.1.2 The Action Potential Discontinuity

As introduced in sec. 2.1, the neuron fires an action potential when the depolarization value crosses the firing threshold. The firing time for a neuron can therefore be found by the equation $v(t_w^{(f)}) = \tau$, where $t_w^{(f)}$ is the firing time and τ is the firing threshold of the neuron.

$$\begin{aligned} v\left(t_w^{(f)}\right) &= \tau \\ \kappa - (\kappa - v_0)\, e^{-\alpha t_w^{(f)}} &= \tau \\ e^{-\alpha t_w^{(f)}} &= \frac{\kappa - \tau}{\kappa - v_0} \\ t_w^{(f)} &= -\alpha^{-1} \ln\left(\frac{\kappa - \tau}{\kappa - v_0}\right) \end{aligned} \tag{3.3}$$

If an absolute refraction time t_r is defined for the neuron, where the depolarization remains constant after firing, this value has to be part of the equation for the estimated firing time. Another way of viewing the resulting equation is as the remainder of current inter–spike interval, $p_r(\kappa, v_0)$.

$$p_r(\kappa, v_0) = -\alpha^{-1} \ln\left(\frac{\kappa - \tau}{\kappa - v_0}\right) + t_r \tag{3.4}$$

Since eq. (3.4) is derived from (3.2), the same constraints are valid. The estimate for the remainder of the current inter–spike interval is only valid until a new time window is initialized. If depolarizing inflow is defined to be constant during a computational time step, a firing

time estimate during the current time step can not change before the estimated time. The estimated firing time can therefore be utilized as the artificial neuron's firing time. If the double precision floating point format is utilized in the simulator, this gives a near–continuous temporal resolution for the neuron's firing times.

An inter–spike interval is finalized by the neuron firing an action potential, after which the neuron's depolarization is reset to the membrane resting potential before the process starts anew. The immediate estimate of the total inter–spike interval can be computed by eq. (3.4), from the neuron's reset potential v_r.

$$p_{isi}(\kappa) = p_r(\kappa, v_r) \tag{3.5}$$

This equation is important when we next consider synaptic flow of action level.

3.1.3 Synaptic Flow

Let all synaptic input be modelled as the flow κ_{ij}, where j represents the presynaptic neuron and i the receiving neuron. Other input that changes neuron i's depolarization is represented by $\xi_i(t)$. The final value for the neuron's depolarization, $\kappa_i = \frac{I_i}{\alpha}$, is defined by the sum of all the neuron's input flows. If $\mathscr{D}$ is the set of integers representing neuron i's presynaptic neurons, the total inflow during the n'th iteration can be written as

$$\begin{aligned} I_{i,t_n} &= \kappa_{i,t_n} \cdot \alpha \\ &= \left(\sum_j \kappa_{ij,t_n} + \xi_i(t_n) \right) \alpha \quad , \; j \in \mathscr{D} \end{aligned} \tag{3.6}$$

Synaptic input, κ_{ij}, is the most important element for neural signal processing[14], and is the main focus of this section. The function

$\xi_i(t)$, representing other input, can have different forms for different depolarizing sources. This element therefore has to be modelled separately for each such source.

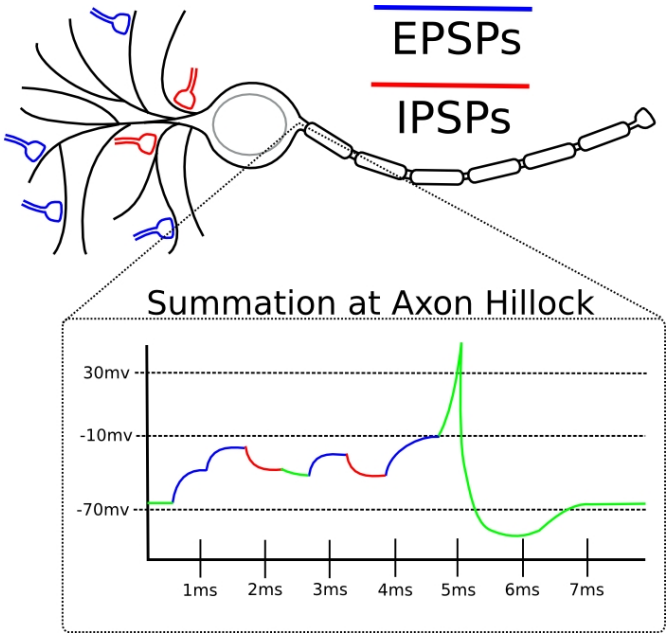


Figure 3.2: *A simulation of neural integration of synaptic input. Excitatory Postsynaptic Potentials(EPSP) increase the membrane potential of the postsynaptic neuron and excite the neuron toward firing. Inhibitory Postsynaptic Potentials(IPSP) hyperpolarize the postsynaptic neuron, thus inhibiting the neuron with respect to firing. When the membrane potential at the axon hillock crosses the firing threshold, set to $-10mV$ in this simulation, an action potential is fired. (The figure is from the website,* http://techlab.bu.edu/resources/software_view/epsp_ipsp/*, of the educational "*EPSP IPSP*" software, intended for illustration of EPSP and IPSP after synaptic transmissions).*

Let the synaptic weight ω_{ij} be defined as the postsynaptic change in depolarization after one synaptic transmission. Synapse j's contribution to the total change in depolarization after a time interval Δt

can be written as the number of transmissions in that interval scaled by the synaptic weight ω_{ij}.

$$\Delta v_i(\Delta t_n) = N_{j,\Delta t} \cdot \omega_{ij,t_{n-1}} \qquad , \; j \in \mathscr{D} \tag{3.7}$$

where N_{j,t_n} represents the number of transmissions in the synapse from neuron j to neuron i in the time interval Δt_n. The variable $\omega_{ij,t_{n-1}}$ represents the synaptic weight, updated at time t_{n-1}.

In κM, a continuous variable representing the present estimate of the inter–spike interval can be used instead of the integer number of transmissions. This enables a higher resolution for the propagated signal and thus a smaller simulation error. For a time interval where the presynaptic activation level κ_j is constant, synaptic flow of activation level can be written as

$$\kappa_{ij,t_n} = \frac{\omega_{ij,t_n}}{p_{isi}(\kappa_{j,t_n})}\Delta t \qquad , \; j \in \mathscr{D} \tag{3.8}$$

For a simulation with constant computational time steps $\Delta t = C_t$, this constant can further be incorporated into the variable that represents synaptic weight ω_{ij}. We arrive at the equation for synaptic flow of activation level for constant time steps:

$$\kappa_{ij} = \frac{\omega_{ij}}{p_{isi}(\kappa_j)} \qquad , \; j \in \mathscr{D} \tag{3.9}$$

When synaptic plasticity is introduced, it is important to remember that synaptic weight is scaled by the constant C_t. For consistency, it is important to scale synaptic plasticity by the same factor.

3.2 Implications of κ–Mathematics

Algebraic analysis of a node's activation level is possible when neural input is represented as a continuous flow. The propagation of information is represented by the distribution of a changed κ. This makes

it possible to utilize a less confusing jargon when talking about neural activation level. Algebraic transfer functions can also be set up, making it possible to do computations on the filter properties of a neural network.

Combined with the concept of synaptic flow and time windows, the κ formalism enables a novel neural simulation scheme, the κ simulation model(κM). By letting the activation level, κ, be propagated as a mechanistic function of the presynaptic neuron's firing frequency, neural network dynamics can be simulated, and eq. 3.2 can be used to find the neuron's depolarization. Thus, the κM simulation model has elements from second as well as third generation ANNs.

The concept of time windows from definition 1 makes it possible to utilize equation 3.2 to simulate the neuron's depolarization. Every time the neuron's activation level is altered, a new time window is initialized by updating the initial depolarization value v_0 and saving the time of initiation, t_0. The depolarization value can be found for any time t_v in a time window, by the equation

$$v(t_v) = \kappa - (\kappa - v_0)\, e^{-at_v} \quad , \; t_v = t - t_0 \tag{3.10}$$

The next firing time can be estimated by eq. 3.4. This makes it possible to have spike times with an intra–iteration time accuracy, and a near–continuous resolution for possible spike times can be accomplished.

Chapter 4

Design/Implementation and Theoretical Comparison

To assess whether κM can be used to simulate a spiking neuron, and to compare the resulting design/implementation with one that utilize numerical integration, both models were designed and implemented by the author. The Numerical Integration Model(NIM) and κM differ in how they compute the neuron's depolarization, how information is propagated, and how spike times are computed. The design of the software intended for a theoretical comparison of the two models is presented in this chapter. This software, referred to as *auroSim*, is later used in experiments that consider the comparative efficiency of the two simulation models.

4.1 General Design of the Simulation Software

When designing a simulator of neural networks, one has to consider simulation time explicitly. Networks of neurons are highly concurrent, but have to be simulated by a sequential composition in the digital computer. Thus, asynchronism for the nodes has to be emulated in the simulation. One way to achieve this is to separate the considered simulation time interval into discrete time slices("time steps"), and let time in the simulation be expressed by the integer time–step number. Before the design of time emulation in *auroSim* is presented, the concept of concurrent is defined in a way that is valid for discrete time.

Definition 2. *Two tasks occur simultaneously if they can not be separated by their time of occurrence.*

Two events during the same time step are therefore defined to happen simultaneously, unless additional information about timing is provided. One approach for emulating concurrency is to let discrete time be defined as a discretization of the real world's clock(RWC). It is important that the whole list of tasks is completed before time is incremented, so that no tasks are lost or delayed to subsequent time steps. This creates a strong dependence between the maximal workload and the minimal computational time step in a simulation. Such a constraint is wasteful and clearly undesirable.

An alternative approach is to utilize a scheme based on serial execution. If all tasks to be executed simultaneously are located in one of two lists(list A), new tasks induced by these actions can be inserted into the other list(list B). When all tasks in one list are completed, the variable that represents time is incremented, and the alternative list becomes the active list. Since causality defines that the effect

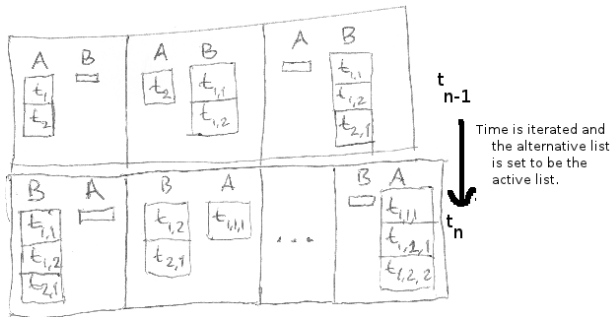


Figure 4.1: *Time simulation based on the sequential computation in the digital computer. Iteration t_{n-1} have list A as the active list. Two new tasks, $t_{1,1}$ and $t_{1,2}$ is generated by task t_1 and inserted into the alternative list(list B). Task t_2 generates task $t_{2,1}$. When all tasks in the active list A is completed, time is iterated and list B is set as the active list. The next computational time step with B as the active list is illustrated in the lower part of the figure. Note that no tasks are inserted into the currently active list.*

happens after its cause, elements can not be inserted into the active list during its execution. This serial approach enables concurrency to be simulated without dependence of RWC, and one does not have to consider the maximal work load of the simulation. The simulation software implemented in this work utilize a modification of this latter time scheme, as presented in section 4.1.2.

4.1.1 Simulator Design

The classes of *auroSim* are classified into one of two groups, classes that represent mechanism dependent on time and classes that are not directly involved in the simulation. Objects in the simulation have causality, and are instantiated from classes derived from the abstract class *timeInterface*. These classes inherit the pure virtual functions of *timeInterface*, and are abstract unless the functions are overloaded in the derived classes. This assures that all objects directly involved

in the simulation have defined its own *doTask()* and *doCalculation()* functions. The reader is referred to [32, chap. 12] for more about abstract classes in C++, and appendix D.2 for UML class diagrams of *auroSim*.

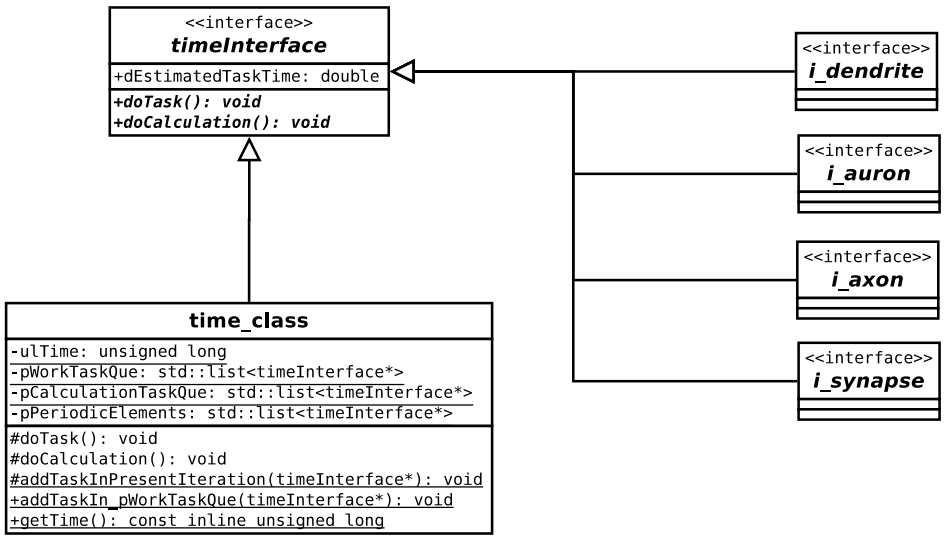


Figure 4.2: *UML class diagram for* auroSim. *All classes directly involved in the simulation are derived from* class timeInterface. *The classes listed on the right hand side of the figure are abstract classes meant to be inherited to neuron subelements of the two simulation models, NIM and κM. For a derived class of* timeInterface *to be able to make objects, the pure virtual functions* doTask() *and* doCalculation() *have to be defined for that class. This ensures that all objects involved in the simulation have a defined behaviour in time.*

All common aspects of the two simulation models, NIM and κM, are located in the abstract neuron subelements classes *i_dendrite*, *i_auron*, *i_axon* and *i_synapse*. These are derived to the model–specific neuron subelements for the κM and NIM simulator. Thus, only the functionality that differ between the two models are implemented separately.

The main loop of the simulation is located in the function *void**

taskSchedulerFunction(void)*. While *bContinueExecution* is set, the first element of *pWorkTaskQueue* is popped and its *doTask()* member function is executed. New elements are inserted at the end of *pWorkTaskQueue*. In this way, the task scheduler function is responsible for driving causality forward in the simulation. A graceful termination of the simulation is possible by setting *bContinueExecution = false*.

```
1  void* taskSchedulerFunction(void* )
2  {
3      ...
4
5      // Simulation's main-loop:
6      while( bContinueExecution )
7      {
8          // Pop first element before execution: Save pointer to the
9          // pointer variable pConsideredElementForThisIteration
10         static timeInterface* pConsideredElementForThisIteration;
11         pConsideredElementForThisIteration = time_class::pWorkTaskQue.front();
12
13         // Then pop element from pWorkTaskQue:
14         time_class::pWorkTaskQue.pop_front();
15
16         // Perform task:
17         pConsideredElementForThisIteration->doTask();
18     }
19     return 0;
20 }
```

In order to simulate time by the serial execution performed by *taskSchedulerFunction(void*)*, *time_class* has been designed as a time separation object. As can be seen in fig. 4.2, this is where *pWorkTaskQueue* is located as a *static* member. Class *time_class* also contains the variable that represent t_n, *static unsigned long ulTime*, and a *doTask()* function that is responsible for incrementation of this variable. Since this class is fundamental for time simulation in *auroSim*, and enables time to be emulated by a single linked list, a whole section is reserved for introducing this class.

4.1.2 Time

Class *time_class* contains the elements *pWorkTaskQueue*, *pCalculationTaskQueue* and *ulTime* as static member variables. Before the main loop of the simulation starts, *pWorkTaskQueue* is initialized by

inserting a single *time_class* object into the linked list. This is done in the function *initializeWorkTaskQueue()*, marked as a friend function of *time_class*. The *friend* keyword is a way of allowing other elements to access the *private* parts of a class declaration[32, Appendix C.11]. The static flag *bPreviouslyInitialized* is used to prevent reinitialization of *pWorkTaskQueue*.

```
1  void initializeWorkTaskQue(){
2  {
3      // Flag to prevent reinitialization
4      static bool bPreviouslyInitialized = false;
5      if(bPreviouslyInitialized)
6          return;
7
8      // Insert pointer to object of time_class, allocated in the free store
9      time_class::pWorkTaskQue     .push_back( new time_class() );
10
11     // Set flag to prevent reinitialization of pWorkTaskQue
12     bPreviouslyInitialized = true;
13 }
```

Because the *time_class* object is allocated in the free store, that object will exist for as long as the implementation runs or is explicitly deallocated. A pointer to this element is legal to insert into *std::list<timeInterface*> pWorkTaskQueue* since *time_class* is derived from *class timeInterface*. The *time_class* object inserted into *pWorkTaskQueue*, referred to as **timeSeparationObj** in the remainder of this text, is responsible for administration of time in *auroSim*. When *timeSeparationObj.doTask()* is called, *ulTime* is incremented after a *self*–pointer is pushed to the back of *pWorkTaskQueue*[22]. In this way, *timeSeparationObj* acts as a time separation object, where the execution of its *doTask()* function is the only way a new time step can be initialized in *auroSim*.

When an element's task is performed from *taskSchedulerFunction(void*)*, the pointer to that element is removed from *pWorkTaskQueue*. Some elements create other tasks, inserting them at the end of *pWorkTaskQueue*. Since *timeSeparationObj* defines the separation of two computational time steps, this element lies after all other tasks in the current time iteration. New tasks are therefore inserted by their order of creation in the subsequent time step (after *timeSeparationObj* in

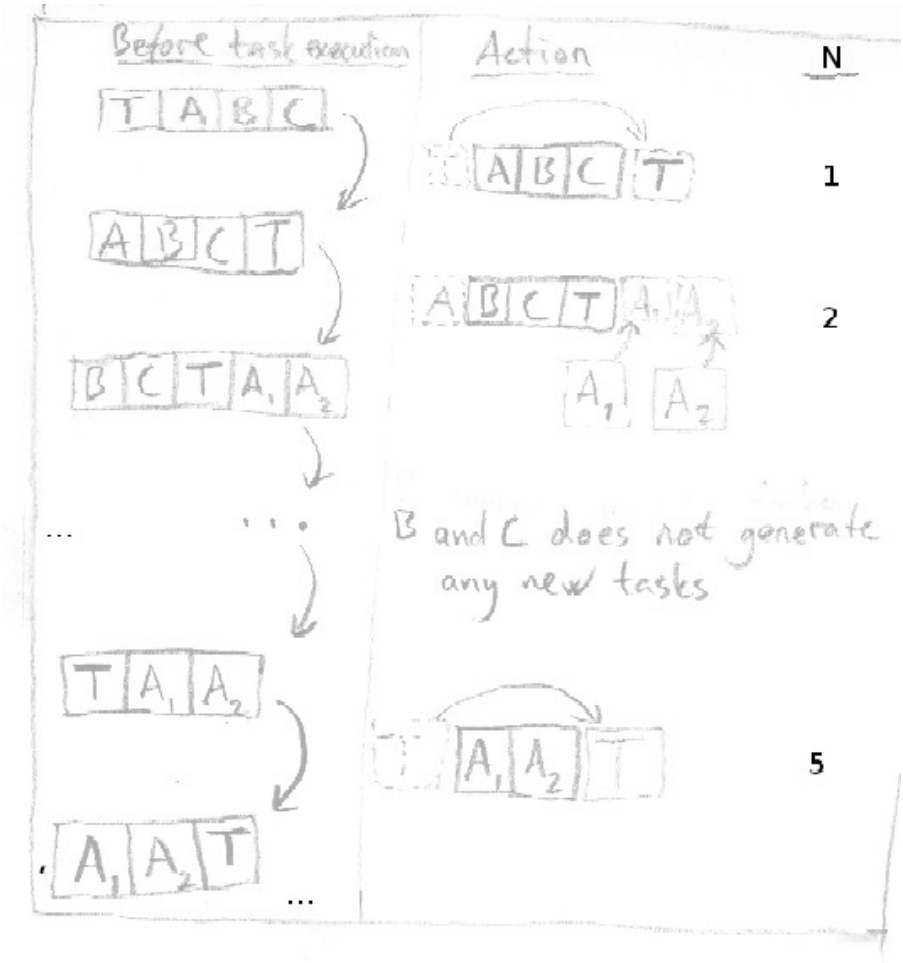


Figure 4.3: *The first sketch of how concurrency is simulated in* auroSim. *taskSchedulerFunction() pops the first element of pWorkTaskQueue and executes its* doTask() *member function. 1) Element T, representing* timeSeparationObj, *iterates* ulTime *and inserts a* self *pointer at the back of* pWorkTaskQueue. *2) Element A generates two new tasks, A_1 and A_2, before the pointer is removed from* pWorkTaskQueue *by* taskSchedulerFunction(). *Element B and C do not generate any new tasks. Computation nr.* 5 *moves* timeSeparationObj *to the back of the list, and the situation is similar to* pWorkTaskQueue *after action nr. 1.*

pWorkTaskQueue). This enables a single linked list to behave like the two alternating lists in fig. 4.1. Time simulation with a single linked list, and the use of *timeSeparationObj* is illustrated in fig. 4.3.

4.2 The Artificial Neuron

The artificial neuron in *auroSim* is designed as a simplification of the biological neuron as shown in fig. 2.1. Each node contains the most important elements of the neuron with regard to signal propagation, located in four subelements that represent [synapse, dendrite, soma, axon]. Each subelement of the artificial neuron has a pointer to the previous and the next element in the signal pathway, enabling a direct simulation of the intracellular communication of the neuron.

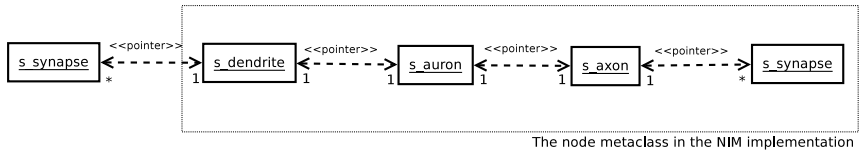


Figure 4.4: *A diagram of the subelements of the artificial neuron. The signal is propagated from the left to the right in the figure. Transmissions in a synapse calls the postsynaptic dendrite's* newInputSignal(). *When it is time for the neuron to fire(checked by* newInputSignal() *in the NIM version of the dendrite), the pointer to the node's auron element is inserted into* pWorkTaskQueue. *Auron's* doTask() *function push its axon pointer to the back of* pWorkTaskQueue, *and the axon's delay is simulated in the same manner.*

Construction of Node Elements

The object design of a node enables the implementer to specify the spatio–temporal resolution in the simulation. To achieve a higher resolution, a smaller computational time step can be defined. The

axonic delay before any particular synapse is defined by the number of serially linked axon elements, scaled by the size of the computational time step.

Since the subelements of the artificial neuron are designed to be separate entities linked by pointers, special effort has to be made to make each node act as a single object. In *auroSim*, this is achieved by considering the whole node as a "metaobject", where all elements are allowed to access the next and previous subelement's protected parts. The *friend* keyword allows another class or function to access the *private* or *protected* elements of the class. In this way, all subelements of a node metaclass are given the same privileges as if it was defined to be a single node class. This design opens many opportunities for the uses of *auroSim*, but is also makes the construction of a node non–trivial.

To construct the node metaclass object illustrated in fig. 4.4, it is most convenient to start with a subelement with only one previous and one subsequent element. The only element that satisfies this constraint is the [auron] subelement. The construction of this element can be represented by the notional constructor *auron::auron()*.

```
1 auron::auron() : timeInterface("auron"){
2     ...
3     pOutputAxon = new axon(this);
4     pInputDendrite = new dendrite(this);
5     ...
6 }
```

The classes [auron, axon, dendrite and synapse] do not exist in the implementation, but are used in this section to illustrate how the constructor of the model specific *s_{element}* and *K_{element}* are designed. Because the implementation always use dereferenced pointers, the *free store* is used for the node subelements. The *new* $< T >$ operator allocates memory for an object of type $< T >$ in the free store, and give the same results as *malloc(size(T))* for memory allocation in C [32]. Utilizing the dynamic memory enables a more precise control of the scope of each element existence, as an element in the free store

lasts for the remainder of the run or until explicitly deallocated [32, Appendix C.9].

Destruction of Node Elements

To avoid memory leaks in C, elements constructed in the free store have to be explicitly deallocated. In *auroSim*, this is done by the subelement's *destructor*.

Like for the construction, the destruction of a whole node starts at the [*auron*] subelement and spreads to the node's more distal parts. For the [*dendrite*] and [*axon*] element, a *while* loop is used to remove all synaptic connections.

```
1 /*** Deallocation is common for both models' dendrite, and therefore located in
      i_dendrite ***/
2 i_dendrite::~i_dendrite()
3 {
4     // Delete all dereferenced pInputSynapse objects. The synapses are
          responsible for removing its pointer from the presynaptic and
          postsynaptic node.
5     while( !pInputSynapses.empty() ){
6         delete (*pInputSynapses.begin() );
7     }
8 }
```

The function *std::list::empty()* returns *false* as long as the list contains elements, and *true* if it is empty. The function std::list::begin() returns a pointer to the first element of the list. The free–store memory used by X is deallocated by the function *delete(X)*. This also calls the destructor of X.

If an axon sends a signal to a deallocated synapse, the action is undefined and errors might occur. To avoid undefined behaviour, the destructor of a class is responsible for removing all pointers to the destructed object. This can be seen in the destructor of *s_synapse*:

```
1 /*** Destructor for s_synapse ***/
2 s_synapse::~s_synapse()
3 {
4     // Remove all [this]-pointers from prenode's pOutSynapses-list:
5     for( std::list<s_synapse*>::iterator iter = (pPreNodeAxon->pOutSynapses).
          begin(); iter != (pPreNodeAxon->pOutSynapses).end() ; iter++ ){
6         if( *iter == this ){
```

```
7               //list::erase() calls the elements destructor, but this does not
                    concern us as the element is a pointer. If the element was the
                    object itself, this would create an infinite recursive
                    destructor loop.
8               (pPreNodeAxon->pOutSynapses).erase( iter );
9           }
10      }
11
12      // Remove all [this]-pointers from postnode's pInputSynapses-list:
13      for( std::list<s_synapse*>::iterator iter = pPostNodeDendrite->
            pInputSynapses.begin(); iter != pPostNodeDendrite->pInputSynapses.end()
            ; iter++ ){
14          if( *iter == this ){
15              //Erase the postsynaptic node's pointer to this synapse:
16              (pPostNodeDendrite->pInputSynapses).erase( iter );
17          }
18      }
19      ...
20  }
```

The [*synapse*] destructor iterates over all pre– and postsynaptic elements' synapse pointers, and removes all pointers to itself. This shows why the [*dendrite*] element's destructor can safely delete its synapses without consideration of postsynaptic pointers to the synapse element. The function *erase(X)* also calls the destructor for element X, but since the argument is a pointer in the listed code, the pointer's destructor is called instead of the synapse's destructor. In this way, an infinite recursive *synapse::∼synapse()* destructive loop is avoided.

4.3 Class Hierarchy – Differentiation by Inheritance

All classes that are part of the simulation are derived from class *timeInterface* [22]. As seen in fig. 4.5, the pure virtual functions *doTask()* and *doCalculation()* stay undefined in *i_auron*. This is also valid for the other subelement classes of the node metaclass, causing the *i_*{*element*} classes to be abstract ({*element*} ∈ [*dendrite*, *auron*, *axon*, *synapse*]).

> A class with one or more pure virtual functions is an abstract class, and no objects of that abstract class can be

created [32].

Figure 4.5 shows the class diagram for the *auron* subelement of a node, where it can be seen that all pure virtual functions are overloaded in *s_auron* and *K_auron*. These classes can therefore be instantiated and objects can be made. The UML class diagram of the other subelements are presented in appendix D.2, showing a similar class hierarchy composition for the other node elements. Because all differences between the two models are implemented separately, the similarities and differences between the two models were emphasized to the author.

4.3.1 NIM – Design and Implementation

A spiking neuron is often simulated by a Numerical Integration Method (NIM), where the depolarization is found by numerical integration. All depolarizing and hyperpolarizing input in the course of a time step is added to the node's depolarization value. Leakage is simulated by subtracting a fraction of the difference between the current depolarization value and the defined resting potential. For simplicity, the resting membrane potential is defined to be zero in *auroSim*. The leakage constant is written as $\alpha = 1 - l_f$, where l_f is the leakage fraction. In this way, the computation of leakage can be implemented as a single multiplication.

$$\begin{aligned} v(t_n) &= v(t_{n-1}) - l_f \cdot v(t_{n-1}) \\ &= (1 - l_f) \cdot v(t_{n-1}) \\ &= \alpha \cdot v(t_{n-1}) \end{aligned} \tag{4.1}$$

Because of the order of magnitude for synaptic input connections in the biological neuron, it is highly likely that a neuron receives synaptic

Figure 4.5: *UML class diagram of the auron subelement of a node. The* i_auron *element in fig. 4.4 is inherited to the model specific classes* s_auron *and* K_auron*. The auron classes are further derived to the sensor_auron classes for the two models, introduced in section B.2.*

input every time step. Leakage is therefore computed every time step in *auroSim*. For sparse neural networks or for simulations with very small time steps, it could be more efficient to implement leakage as $v(t_n) = \alpha^x \cdot v(t_{n-x})$, since the probability of not getting input every time step is larger.

The Nodes' Input

In *auroSim*, the [*dendrite*] receives all input to the artificial neuron. As introduced in sec. 2.1.2, the size of the transmission at any particular synapse is defined by the synaptic weight of that synapse. Depending on whether the synapse is an *excitatory* or *inhibitory* synapse, the postsynaptic membrane potential is either increased or decreased. In *auroSim*, this is implemented by letting the synapse send $[(1 - 2\,\text{bInhibitorySynapse}) \cdot \omega_{ij}]$ as an argument to the postsynaptic dendrite's *newInputSignal(double)* function.

```
1  inline void s_synapse::doTask()
2  {
3      // If the synapse is inhibitory, send inhibitory signal(subtract):
4      //  (bInhibitorySynapse is a boolean variable that defines whether the
           synapse is inhibitory or not)
5      pPostNodeDendrite->newInputSignal( (1-2*bInhibitoryEffect)*(FIRING_THRESHOLD
           * dSynapticWeight) );
6
7      // Write to log:
8      synTransmission_logFile <<"\t" <<time_class::getTime() <<"\t"
9                      <<(1-2*bInhibitoryEffect) * dSynapticWeight
10                     <<" ;    \t#Synpaptic weight\n" ;
11 }
```

The postsynaptic dendrite's *newInputSignal(double)* function adds the input to the node's depolarization. If this variable goes beyond the firing threshold, an action potential is initialized by pushing the node's first axon pointer to *pWorkTaskQueue*.

Action potential in *NIM*

Spatio–temporal delay in the axon is simulated by a linked list of [*axon*] objects, each pushing the next element on to *pWorkTaskQueue*.

For a greater temporal resolution, smaller computational time steps and a larger number of serially linked axon elements can be utilized. When one of the axon elements contains a pointer to an output synapse, that synapse's pointer is pushed to *pWorkTaskQueue*, causing synaptic transmission to happen in the following time step.

4.3.2 κM – Design and Implementation

As seen in fig. 4.5, the design of *K_auron* is more complex than for *s_auron*. This is partially because the node has to keep an overview of the floating point time instance for initiation of new time windows. A near–continuous resolution for the initiation of time windows, combined with the ability to compute the exact firing time by equation 3.4, enables the use of intra–iteration firing time accuracy.

Since synaptic flow is utilized instead of discrete synaptic input transmissions, the activation variable of a κM node is defined to represent the activation level κ from equation 3.2. Every time a new time window is initialized, the initial depolarization $v_0 = v(t_0)$ is updated by computing the new value by eq. 3.10. By also saving the time of initiation, t_0, this equation can be used to update the neuron's depolarization the next time a new time window is initialized. This enables κM to be used to simulate the neuron's depolarization by the algebraic value equation.

The Node's Input

In section 3.1.3, discrete synaptic flow is defined as the number of transmissions, $N_{j,\Delta t}$, scaled by the synaptic weight for that synapse.

$$\Delta v_{ij}(\Delta t_n) = N_{j,\Delta t} \cdot \omega_{ij,t_{n-1}} \qquad , \; j \in \mathcal{D}$$

An appropriate description of $\Delta v_{ij}(\Delta t_n)$ is the *synaptic flow of activation level*, since the flow has a direct influence on the postsynaptic

node's activation value κ_i. By the use of synaptic flow κ_{ij}, defined by the presynaptic neuron's activation level, a data format of higher precision than integers can be used. The postsynaptic activation level can be written as

$$\kappa_{i,t_n} = \sum_j \kappa_{ij,t_n} \qquad , j \in \mathcal{D}$$

where $\mathcal{D}$ is the set of integers representing neurons with a synaptic connection to neuron i. For a κM implementation, it could be advantageous to consider edge transmissions κ^*_{ij} as the *change* in synaptic flow.

$$\begin{aligned} \kappa^*_{ij,t_n} &= \frac{\mathrm{d}}{\mathrm{d}t}\kappa_{ij}(t_n) \quad , j \in \mathcal{D} \\ &= \kappa_{ij,t_n} - \kappa_{ij,t_{n-1}} \end{aligned} \tag{4.2}$$

When a subset $\mathcal{M}$ of the presynaptic neurons have an altered synaptic flow, this method gives a slightly more efficient simulation — only the edge transmissions from $\mathcal{M}$ have to be added to the postsynaptic node's activation level. This can be written as

$$\kappa_{i,t_n} = \kappa_{i.t_{n-1}} + \sum_l \kappa^*_{il,t_n} \qquad , l \in \mathcal{M} \subseteq \mathcal{D} \tag{4.3}$$

Because edge transmission as the derivative demands numerical integration, the accumulation of error has to be considered. A specialized *timeInterface* derived class, whose *doTask()* recalculate the node's activation level is devised for this purpose. An object of this class is included as a member variable of the *K_auron* class, and gives a periodic recalculation of the node's activation level. The recalculation is designed to be dynamic, in such a way that the period to the next recalculation is longer if the deviation from the recalculated activation level level is small. Documentation for *recalcKappaClass* can be found in appendix A.3.

Action Potential in κM

As discussed in section 3.1.2, the use of the algebraic solution when simulating the neuron enables the node's spike times to have a near–continuous time resolution. In *auroSim*, this is implemented by letting *time_class::doTask()* insert an *auron*'s pointer when it is estimated to fire during the next time step. Since this is done before time is incremented, the element will execute its task during the correct computational time step. By sorting the *K_auron* tasks by the *dEstimatedTaskTime* member variable, neuron firing is scheduled by the estimated task time instead of the tasks' order of creation. This could be of importance when multiple processing units are utilized for simulation of a large network of neurons.

To simulate spatio–temporal delay in the axon, each output synapse is scheduled after its predefined transmission delay. For example, if the axonic delay before a synapse is defined to be 2.15 and the node fires at time 141.2, the synapse's task can be scheduled for execution at time 143.35 by writing this time to the synapse's *dEstimatedTaskTime*. Due to the mechanisms described in sec. 4.1.2, the synapse will execute its task at that time without the need for simulation of axonic propagation delay. This gives a more constant work load for the simulation, something that is very advantageous if spiking neuron simulations are to be used for real–time applications.

4.4 A Theoretical Comparison of the two Models

4.4.1 On Computational Complexity

A κM simulation involves more complex operations than in a NIM simulation. To assess whether it takes longer time to simulate, a quick experiment was set up. A set of runs of *auroSim* have been executed to compare the run time of the experiment that produced the simulated solution in experiment 2 (see section 5.1.2). All simulations were conducted with the same parameters, using either the κM or the NIM simulation model. The run time of the two variants of *auroSim* was found by the command

time ./auroSim.out -r1000000 -n1.5

This executes a simulation with 1.5 forcing function periods, each with 10^6 time steps. The mean output of the *time* shell command, for κM and NIM, is presented in table 4.1.

	$NIM_{1.000.000}$		$\kappa M_{1.000.000}$	
	Mean	Std. dev.	Mean	Std. dev.
run	0.434s	0.104	0.800s	0.088
user	0.336s	0.337	0.703s	0.079
sys	0.007s	0.003	0.009s	0.007

Table 4.1: *Mean run time for ten runs of the NIM and the κM version of auroSim. The standard deviation for all items is also listed. The interested reader is referred to appendix C.3 for the run time of all runs in the experiment.*

The κM simulations required almost the double amount of 'wall clock time' in these particular runs of auroSim. A comprehensive study of the run time of the two simulation models has not been conducted,

since the relative run times of the two models is hardware–dependent and would only give an example for this specific architecture. The results still indicate that the κM implementation requires more computational resources than the NIM implementation. This is most probably due to the computational complexity of κM.

Because the principal goal of a simulation is to produce accurate results, and the error can be decreased by making the computational time steps smaller, it is possible to measure efficiency by the simulation method's error[27]. The run times in table 4.1 put large requirements on κM in order for this method to be more effective than NIM. A comparative efficiency analysis, based on simulation accuracy, is presented in chapter 5.

4.4.2 Time and Error for the Two Models

When simulating time variant variables in discrete–time environments, truncation errors arise from the discretization of time. Mechanisms than make the variable time variant are computed based on the previously updated value instead of continuously updating the value. For a NIM simulation, this means that all depolarizing input and the effect of leakage during a time step does not influence the total size of that time step's leakage. This effect is larger for simulations with longer computational time steps. As mentioned in section 2.3, a simulation with a smaller error can therefore easily be designed by increasing the temporal resolution of the simulation. This is not a good solution, as it also greatly increases the computational load of the simulation.

Because the Numerical Integration Model(NIM) is fundamentally different from a simulation model that considers depolarizing flow(κM), the two models' error mechanisms are analyzed separately. All analysis done in this text are of the un–improved models, implemented with a simple sample–and–hold numerical technique. Optimization by estimating the intermediate values in each time step can be utilized for

both models, but this is outside the scope of this work.

Numerical Integration Method(NIM)

The considered variable in a NIM simulation is the depolarization value of the neuron. An inter–spike interval is completed when this value goes to suprathreshold levels, causing the initiation of the next spike. The neuron's depolarization is reset to $v_r < \tau$ after a spike, meaning that the considered variable goes through a net rising phase in the course of an inter–spike interval.

A rising phase means that earlier values are smaller than the current value. Equation 4.1 shows that leakage is proportional to the depolarization value, and that the previous value is utilized for computing the current leakage. The simulated leakage in NIM thus generally produces a positive depolarization error, i.e. it causes the depolarization value to be larger than it should be. In the course of an inter–spike interval, all local truncation errors caused by this effect are integrated to what will be referred to as the inter–spike truncation error. When utilizing the sample–and–hold integrated technique, this error is predictable and always causes the neuron to fire to early. An early firing gives an earlier start of the next inter–spike interval, meaning that the neuron's depolarization is integrated over an interval that is too long. In most cases, this further increases the positive depolarization error and is the background of the cumulative property of the NIM error.

An opposite error comes as a direct consequence of having discrete time. The *action potential* is defined to happen when the depolarization crosses the firing threshold from below. To preserve causality in a network of artificial neurons, the *action potential* has to be delayed to the time step after the threshold crossing. This introduces a small delay before firing, causing a delayed transmission and a delayed initiation of the subsequent inter–spike interval. As previously described, this gives an initial depolarization error for that inter–spike interval.

The error from having discrete possible firing times has the opposite effect of the inter–spike truncation error.

The net inter–spike simulation error is defined by the relative size of these mechanisms. The error from having an erroneous leakage varies from having a size of $e_l = 0$, if the neuron uses an eternity to reach the firing threshold, to the size of the correct leakage if the depolarization goes all the way from v_r to τ in one iteration. The error caused by having discrete possible firing times varies from $e_d = 0$, if the threshold crossing happens at the very end of the time step, to having a magnitude defined by the size of one full computational time step if the threshold crossing happens immediately after the initiation of that time step. The derivative of the accumulated truncation error(the change in global truncation error) is therefore hard to predict and supress. Since the global truncation error in NIM is defined by the integral of all previous depolarization errors, any systematic local error causes the global truncation error to diverge for $t \to \infty$.

Algebraic Simulation Model(κM)

In κM, the considered variable is the *synaptic flow* of activation level, visualized as a stream in the gutter analogy in sec. 3.1. This flow varies as a continuous function within a bounded domain, and does not have a net rising phase during each inter–spike interval. The flow of activation level is invariant of time, and a delayed computation of the neuron's activation level only delays its response. Thus, the κM error is bounded and varies as a function of the derivative of the neuron's input flow.

If the κM simulator is implemented with intra–iteration time accuracy (see section 3.1.2), the next inter–spike interval can be initiated at the computed time instance. If all tasks are executed according to estimated spike times, a task planned slightly before another will be initiated before that task. This effect is only limited by the data

format used, and a double precision floating point variable is utilized in *auroSim*. The IEEE standard defines the smallest exponent of this data format to be -308, giving an accuracy where two numbers can be separated by steps down to 10^{-308} [18]. This makes it possible to have almost infinitesimal sizes for the delay meant to assure causality, and the next inter–spike interval can be initiated immediately. In this way, also the second discussed error mechanism of NIM is avoided in κM. Because the κM error only comes from the delayed update of a bounded variable, the error varies within a bounded domain. This will be referred to as the stability property of the κM error.

Chapter 5

Efficiency; Experimental Comparison

The primary design criteria for a simulator is to produce accurate simulation results. As introduced in section 4.4.2, the simulation error of a neural simulator can be decreased by increasing the temporal resolution of the simulation. This also greatly increases the computational load of the simulation, as more computations have to be conducted for the same simulated time domain. A relative efficiency comparison can therefore be performed by comparing the accuracy of two simulations for a given temporal resolution, or the resolution needed to accomplish the same accuracy.

Since the run time of the κM simulation presented in sec. 4.4.1 is almost the double of that of the NIM simulation, large requirements are lain upon κM's accuracy. If the hypothesized accuracy improvement is large enough, κM can still be as efficient, or even more efficient, than the NIM simulation model. The purpose of this chapter is to assess the comparative efficiency of the two models, by considering the absolute simulation error for simulations done by κM and NIM.

5.1 Design of Experiments to Assess Efficiency

To compare the accuracy of the two simulation models, low–resolution simulations of κM and NIM can be compared to a simulation with much higher temporal resolution. In this work, the low–resolution simulations have less than 1000 time steps per forcing function period, while the high–resolution NIM simulation has 1.000.000 time steps per period. The high–resolution simulation results will be referred to as the simulated solution in the remainder of this text.

The simulated solution has a number of time steps that is more than three orders of magnitude larger than for the low–resolution simulations. The simulated solution can therefore be considered to be the correct time course, for a temporal resolution up to that of the low–resolution simulations. To assess the accuracy of the two simulation models, one can therefore define the simulated solution to be the correct answer and find the errors for each of the two low–resolution simulations.

The time course for the neuron's depolarization in the three simulations are compared in *Octave*, an open source numerical computing environment similar to *Matlab*. The considered variables are written to a log file during the execution of *auroSim*, resulting in an executable *Octave* script when a run of *auroSim* is finished. All plots with the caption "Generated by *auroSim*" are results of executing such log files in *Octave*. The reader is referred to appendix B.1 for more on *auroSim*'s logging facility.

To make the experiments as comparable and reproducible as possible, the behaviour of a single node is simulated for the two simulation models. This node is implemented as a sensory node that receives depolarizing input defined by an externally applied signal $\xi_i(t_n)$. For the sake of reproducibility, algebraic functions are utilized for all ex-

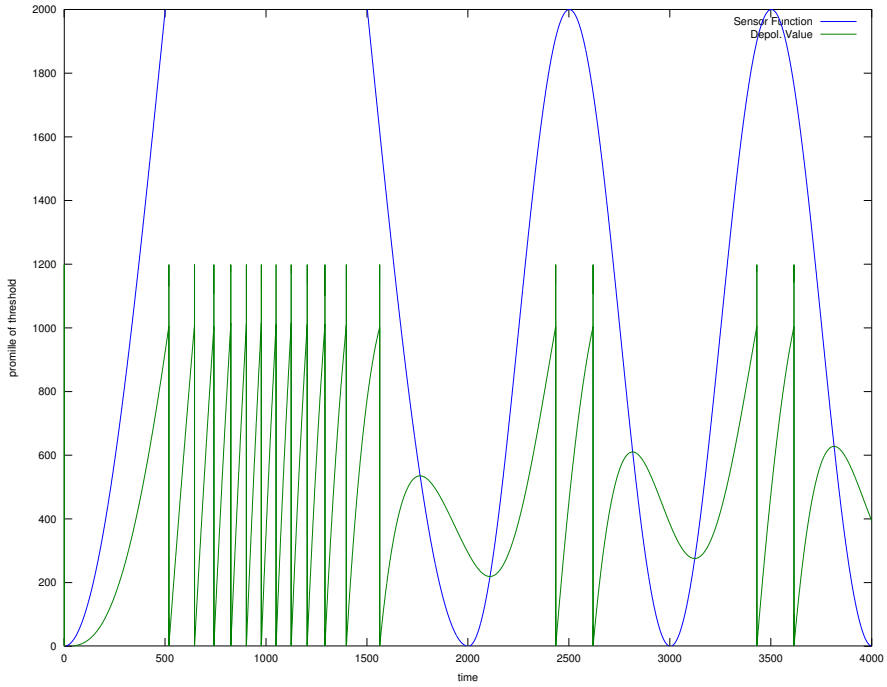


Figure 5.1: *Plot of a NIM node's sensory function. The sensory function is set to be* $f_s(t_n) = 2\tau\left(1 - cos(\pi \cdot \frac{t_n}{1000}\right)$ *for* $t_n \in [0, 1500]$. *After* $t_n = 1500$, *the sensory function halves the amplitude and doubles the frequency. Firing is represented by a vertical line for the depolarization from* $y = 0$ *to* $y = 1200$. *(Generated by* auroSim*) [22]*

periments in this work. For details on the design and implementation of the sensory node, see appendix B.2.

5.1.1 Experiment 1: Idealized Situation

First consider an idealized situation, with a constant depolarizing inflow. This can be implemented as a sensory neuron with a forcing function $\xi(t_n) = 1.1\tau$. This simple input flow simplifies analysis and shows whether the theory presented in chapter 3 can be used to simulate the neuron.

This experiment can be used to assess whether the concept of *time windows* and *intra–iteration time resolution* works as designed. The concept of *time windows* can be examined, since the activation level κ is "changed" to the same value every time step. Each time κ is changed, a new *time window* is initialized and a new estimate for the next firing time is computed. This also enables an analysis of whether proactive firing time scheduling can be used to simulate the neuron's firing: If the spike is delayed as a result of having a time grid of possible spike times, the simulation error will have a step from before to after the spike. The concept of intra–iteration time accuracy therefore works as intended if the error after a spike is a linear continuation of the error curve before the spike. To make the effect observable in plots of the neuron's depolarization, a temporal resolution of only 100 time steps is chosen for experiment 1.

Because of the simple sensory function, the exact solution can be computed for the neuron's spike times. Experiment 1 can therefore be used to assess the accuracy of a simulation, up to a very high precision. This enables an analysis of the simulated solution's error, and a discussion of when it can be considered to be the correct solution for accuracy comparisons.

5.1.2 Experiment 2: More Realistic Input Flow

Section 4.4.2 concludes that the κM error is a result of the delay between an altered depolarizing flow and the initiation of a new *time window*. This implies that the error is constant for a constant forcing function. When designing an experiment for assessing the efficiency of the two simulation models, the form of the input should preferably affect both simulation models equally. The best way to achieve this is to consider a forcing function where neither the value nor the derivative of any order is constant.

Let the forcing function be defined by a trigonometric function that gives an activation level corresponding to κ being above the firing threshold for the whole simulation. When $\kappa < \tau$, the simulated depolarization has the possibility to level out at a subthreshold value, suppressing the simulation error. This is avoided to make the error from the two simulation models prominent. The forcing function in experiment 2 is defined to be

$$f(t) = (2.1 + sin\left(2\pi \cdot \frac{t_n}{l}\right)) \cdot \tau \tag{5.1}$$

where l defines the temporal resolution of the simulation. The neuron was simulated over one and a half period of (5.1), to enable a comparison of the error for two time instances where the forcing function is in the same phase. This was done to expose any cumulation of error for the two simulation models.

./auroSim.out -n1.5 -r[temporal resolution]

It is important to emphasize that the experiment is conducted with the first chosen forcing function. No attempts have been made to optimize the results for any of the models. This can be done, and be the basis of a more thorough analysis of the two simulation models' error mechanisms.

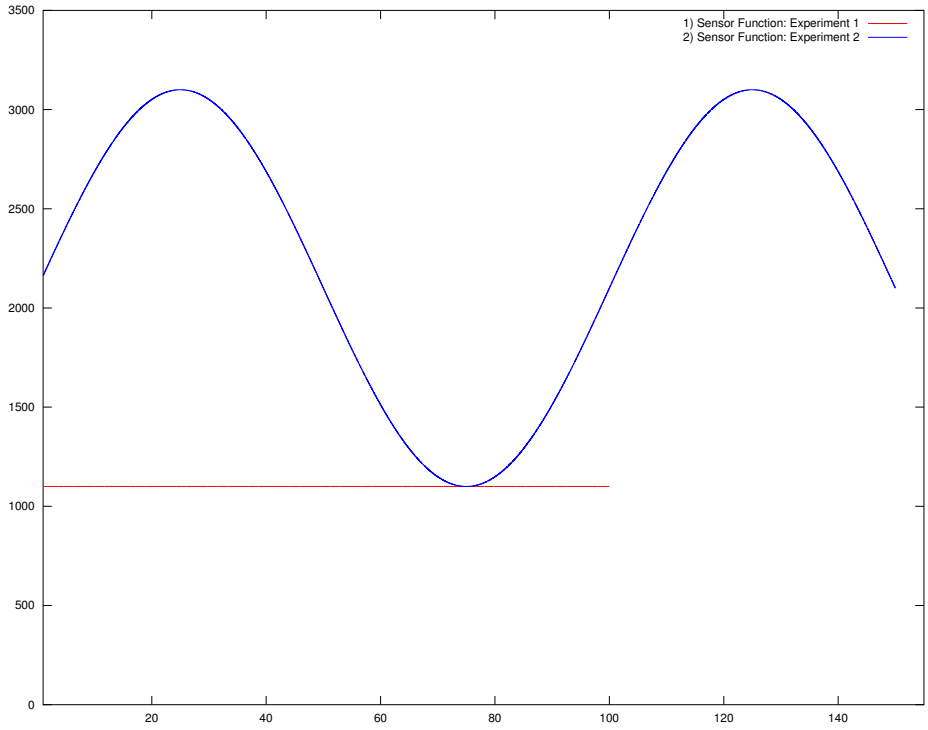


Figure 5.2: *Sensory functions for the two experiments. 1) First experiment — constant input, corresponding to inserting a constant current through a probe 2) Second experiment — dynamic input, corresponding to one and a half period of eq.* (5.1)*. (Generated by* auroSim*)*

5.2 Results

5.2.1 Static Input Flow

The primary motivation behind experiment 1 is to find whether κM can be utilized to simulate the neuron. The fundamental concept of *time windows* is put to the test, since the activation level is changed(to the same value) every computational time step. Because initiation of a new *time window* involves recalculation of the node's firing time estimate, this experiment can also be used to test whether proactive firing time scheduling works as designed. A plot of the results is presented in fig. 5.3.

The algebraic solution for the neuron's spike times was found by adding (3.5) recursively to the previous firing time. The results are presented in table 5.1, alongside the simulation results from the κM_{100} simulation, with a temporal resolution $l = 100$, and the simulated solution. The $NIM_{1.000.000}$ simulation's absolute error has a monotonic increase of up to one time step for every spike, while the κM_{100} simulation appears to give the correct spike times for all spikes in the simulation

Spike #	Analytic solution	κN sim.	Simulated solution
1	23.978953..	23.978953..	23.9789
2	47.957905..	47.957905..	47.9578
3	71.936858..	71.936858..	71.9367
4	95.915811..	95.915811..	95.9156

Table 5.1: *Spike times for the artificial neuron. The analytic solution is computed by adding* (3.5) *recursively to the previous spike time. The κN simulation has a temporal resolution of $l = 100$, while the simulated solution is the result of a $NIM_{1.000.000}$ simulation with $l = 1.000.000$.*

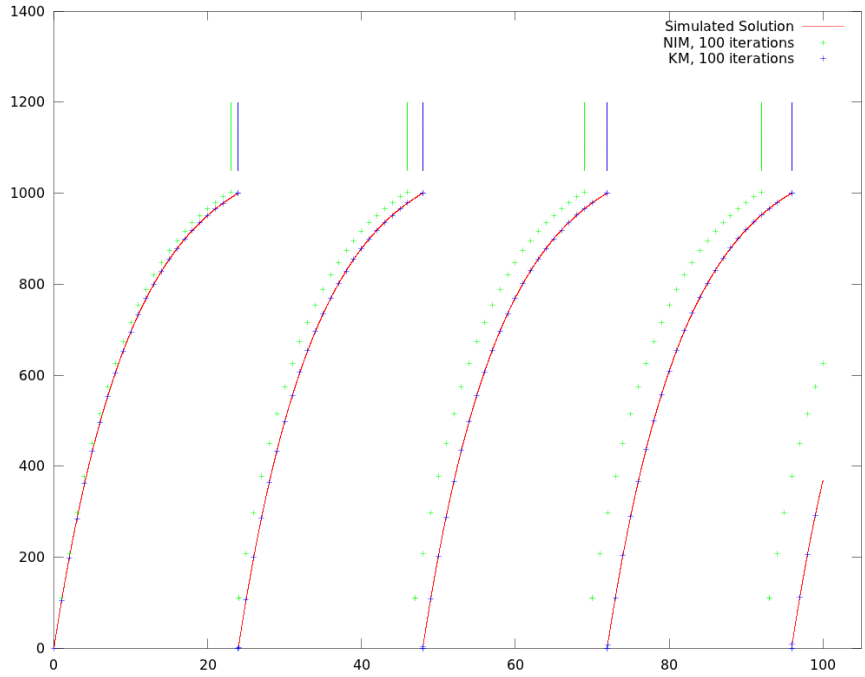


Figure 5.3: *The transient time course of the artificial neuron's depolarization, simulated with NIM and κM. The computational time step is set to $\Delta t = 1\%$, giving 100 time iterations for the two simulations. The red curve shows the simulated solution of experiment 1. (Generated by* auroSim*)*

5.2.2 Dynamic Activation level

Experiment 2 considers a dynamic input current, defined as one and a half period of (5.1). The simulation results are presented as points in fig. 5.4 whenever a new value is available. Note that the NIM simulation is conducted with the temporal resolution $l = 1.000$, while the κM simulation only has 100 time steps per forcing function period.

Since the depolarization value is written to log every time it is updated, the number of points from each simulation indicate the temporal resolution of that simulation. Spikes are represented by a vertical line from $x = 1050$ to $x = 1200$ when the neuron fires. The spikes in the figure indicates that the simulation error is larger in the second period of the forcing function than in the first period. To enable further analysis of this effect, the spike time errors have been isolated and is presented in fig. 5.5.

The error in spike times for the $NIM_{1.000}$ simulation shows the hypothesized cumulative property of the NIM error. To examine the extent of the two models' error properties, experiment 2 was simulated over a time interval that is ten times as long. A plot of the resulting spike time errors is presented in fig. 5.6.

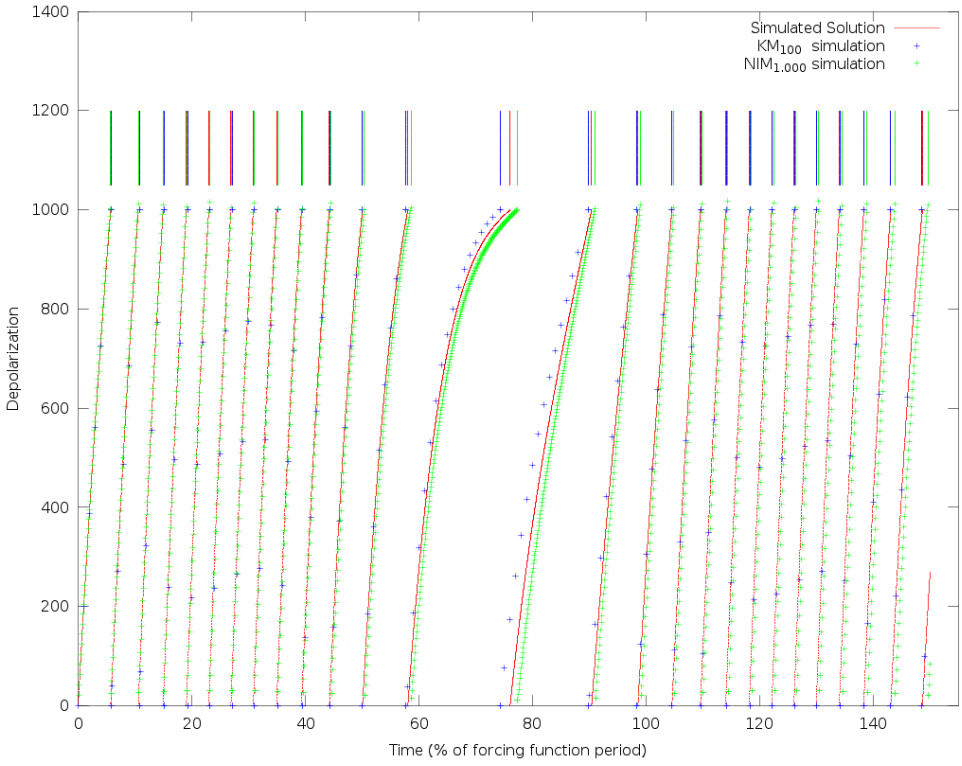


Figure 5.4: *The neuron's depolarization curve in a $NIM_{1.000}$ simulation and a κM_{100} simulation. The two simulations have a number of time steps that differ with one order of magnitude. The red curve shows the simulated solution of experiment 2. (Generated by* auroSim*)*

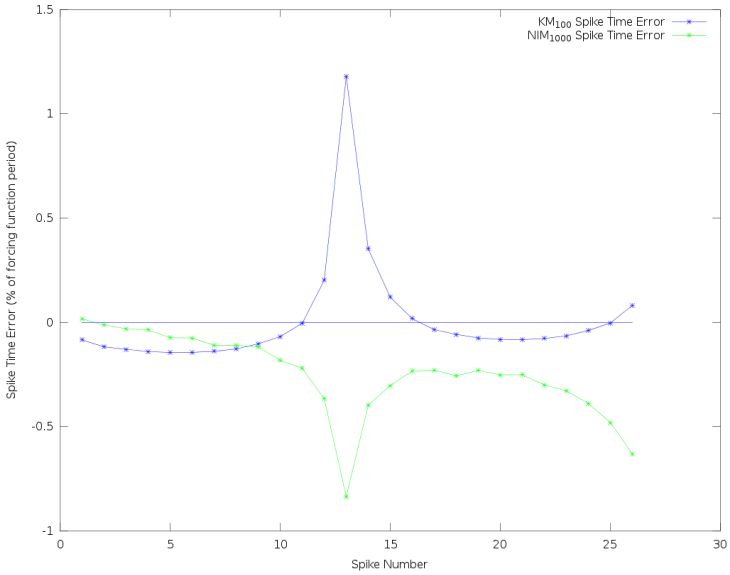


Figure 5.5: *The spike time error for all* 26 *spikes in the* κM_{100} *and the* $NIM_{1.000}$ *simulations. From fig. 5.4. it can be seen that the second period of the forcing function starts at spike number* 15*. An indication of the cumulation of error can therefore be found by comparing the spike time error for spike number* 5 *and spike number* 20 *for the two models. (Generated from log files generated by* auroSim*)*

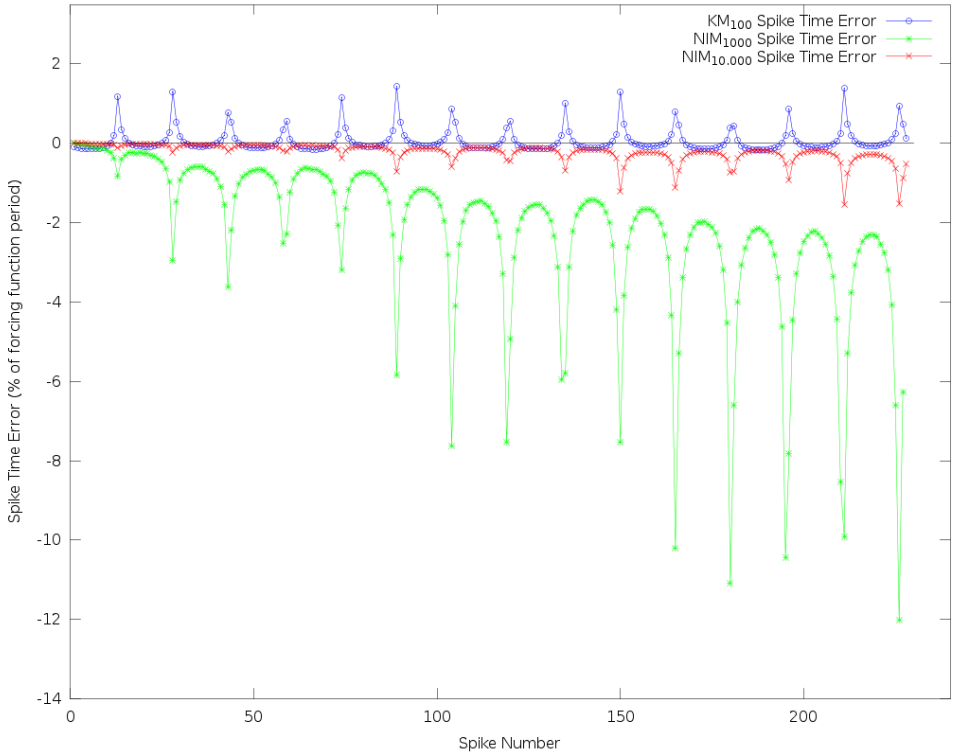


Figure 5.6: *The error in spike times for the* κM_{100}*,* $NIM_{1.000}$ *and* $NIM_{10.000}$ *simulations, simulated over a time interval that is ten times as long as in experiment 2. Due to the number of spikes, the simulated solution was found by a* NIM_{1E8} *simulation to make sure the solution's error is acceptable. The* $NIM_{10.000}$ *and the* κM_{100} *simulations gave the correct* 228 *spikes, while the* $NIM_{1.000}$ *simulation produced one spike less. A* NIM_{100} *simulation resulted in only* 224 *spikes, where the largest error was* -33.6*. (Generated from log files generated by* auroSim*)*

5.3 Discussion of Experimental Results

The primary motivation for the first experiment is to assess whether the theory discussed in chapter 3 makes it possible to implement a spiking neuron simulator based on synaptic flow. The concept of time windows, as defined in sec. 3.1.1, enables the use of the algebraic solution for simulation of the neuron's depolarization. In the implementation used in this work, a new time window is initiated every computational time step, making it irrelevant whether the activation level is constant or dynamic. This makes the results from experiment 1 pertinent for error analysis.

The simple form of the neuron's forcing function in the first experiment enables a precise error analysis for the its spike times. It is possible to compute the neuron's firing times algebraically, enabling an analysis of the simulated solution's error. The simulated solution has a cumulative error that increases with up to one computational time step for every spike, given this level of input. In a simulation with only 26 spikes, this gives a maximum spike time error for the simulated solution, $f_{e,max} = \frac{26}{1000000} = 2.6 \cdot 10^{-5}$. Thus, in experiment 2, the theoretical maximum spike time error for the simulated solution is much smaller than the computational time step in both low–resolution simulations.

$$\Delta t_{NIM,1.000} = \frac{1}{1000} = 10^{-3}$$

This shows that the simulated solution can be considered to be the correct solution, up to an accuracy defined by the low–resolution simulation with finest granularity, $NIM_{1.000}$. In the second part of experiment 2, where the experiment is simulated over a time interval that gives 228 spikes for the neuron, a $NIM_{100.000.000}$ simulation is used to define the simulated solution.

Experiment 2 considers a sinusoidal input flow corresponding to an activation level that varies between 1.1τ and 3.1τ. Since the forcing

function has the property that no aspect of the signal is constant in the time domain, the results from experiment 2 is more valid for an efficiency analysis than experiment 1. The experiment shows that the κM_{100} simulation generally is more accurate than the $NIM_{1.000}$ simulation. The comparative efficiency improvement is larger when the same experiment is simulated over a time interval that is ten times as long. The absolute error becomes larger in the $NIM_{10.000}$ simulation than in the κM_{100} simulation before the simulation is over. This implies a considerable efficiency improvement, since the NIM simulation utilizes a number of time steps that is two orders of magnitude larger than the κM simulation.

Chapter 6

Discussion and Conclusion

6.1 Summary

The mechanisms of biological neuron networks, the computational system of biological beings, is not fully understood. On a low level, neuroscientists have found that networks of neurons propagate information by discrete action potentials. An action potential causes a transmission through all the neuron's output synapses, leading to the increase or decrease in the postsynaptic neuron's value. This value, referred to as *the depolarization* of the neuron, is the result of a leaky integration of synaptic input transmissions.

Digital simulations have discrete time, and a neuron's depolarization is often simulated by numerical integration. This is done by adding synaptic input and subtracting an estimate of the neuron's leakage. In this work, the previous time step's value is utilized when computing leakage for the NIM model (*sample–and–hold integration*).

This study shows that the error from each computational time step varies like a stochastic variable, and that the total error is defined as the integral of all local errors. This results in a diverging simulation

error, unless the local truncation error has an expectancy value of zero. In an attempt to avoid this, a novel simulation scheme has been developed that does not involve numerical integration. Using the concept of *time windows*, time intervals where the neuron's depolarizing inflow is held constant, a neural simulator was developed that utilize the algebraic value equation in these intervals. Software intended to make differences in design of the two simulation schemes have been designed and implemented, *auroSim*. The artificial neuron has the functional lay–out of the biological neuron, with four distinct subelement types, [*i_dendrite*, *i_auron*, *i_axon*, *i_synapse*]. The abstract *i_*{*element*} types are inherited to *s_*{*element*} and *K_*{*element*}, model specific classes. All common aspects between the two simulation models can thus be placed in the ancestor *i_*{*element*} class, making principal differences in design of the two simulation schemes prominent.

It is shown experimentally that although the κM simulation scheme is computationally more complex, the simulation is more effective. Because the κM simulation scheme produces less errors, longer computational time steps can be used to achieve the same accuracy. This makes it possible to utilize fewer computational time steps to achieve the same degree of simulator accuracy, enabling a more effective simulation. It is also shown that the absolute error of the algebraic simulation scheme is bounded, something that could be of importance in complex ANN simulations.

6.2 Discussion

One question that presents itself is the importance of a gradually increasing cumulative error. The most immediate errors are the ones that alter the length of an inter–spike interval. These are represented as the derivative of the spike–time error curves in fig. 5.5; when an inter–spike interval has an erroneous length, the spike–time error is

changed by this amount. Fig. 5.5 shows that in the first period of the forcing function, the κM_{100} spike–time error change with about the same rate as the $NIM_{1.000}$ error. After spike nr. 20, the derivative of the spike–time error is larger in the $NIM_{1.000}$ simulation than in the κM_{100} simulation. This illustrates a significant efficiency improvement, as the NIM simulation has a temporal resolution that involves ten times as many time steps as the κM simulation.

Fig. 5.6 shows the spike time errors for the same experiment, simulated over a longer time interval. One can observe the cumulative property of the NIM error as a gradual increase in the absolute spike–time error. To compare the κM_{100} simulation's spike–time error with the $NIM_{10.000}$ simulation's error, the difference in absolute error is presented in fig. C.2. This figure shows that in the second half of the experiment, the κM_{100} error is generally less than the $NIM_{10.000}$ simulation's error. This implies an even greater efficiency improvement, as the NIM simulation has a number of time steps that is two orders of magnitude larger than the κM simulation's. In all conducted experiments, this effect becomes larger for longer simulations.

Reproducibility has been an important element in the conducted experiments in this work. The most important elements of the simulation software are well documented, and the forcing functions in the experiments are represented by algebraic functions. It is possible that the use of algebraic forcing functions limits the validity of the results, since the input to a node in a neural network is far from being a smooth algebraic function. Experiment 2 considers a sinusoidal forcing function, where neither the value nor the derivative is constant for any time interval. This can be used as a basis in a Fourier series to produce any periodic signal. This forcing function can therefore be seen as a component in any signal, and is considered to be an appropriate algebraic function for efficiency measurements. A stochastic Wiener process could also be used, but this would make the experiments harder to validate for others. To simplify further analysis and for a thorough

study of the implementation, *auroSim* has been published under *GPL*. The source code can be found under branch *master* in the git repository located at *https://github.com/leikanger/masterProject* [23].

One element that could be worth examining, is the ability of the κM simulation model to simulate the neuron by other formal neuron models. The LIF neuron model is often used because it is simple, and does not involve complex operations. Other neuron models are reported to produce more accurate simulation results [5]. The κM simulation scheme is thought to be applicable for any neuron model where the depolarization is described by an ordinary differential equation. As long as the value equation is defined as a function of a single variable, *time windows* can be defined, and the κM simulation model can be utilized. The use of κM for systems defined by partial differential equations or sets of ordinary differential equations, is also an area that could be worth examining.

When edge transmission is implemented as the derivative of synaptic flow, transmissions are only needed when there is an altered activation level for the presynaptic neuron. When a double precision floating point data type is used, with the smallest increase defined to be 10^{-308}, it is highly unlikely that the activation level of a node remains constant over any time interval. The concept of edge transmission as the derivative does not decrease the efficiency of a simulation, but it does not improve it either. It does increase the complexity of the design/implementation, and is recommended to be removed for further uses of *auroSim*.

6.3 Conclusion

This work introduces an entirely new way of considering a neuron's activation level. The novel formalism considers what the neuron's depolarization would approach, κ, if no firing interrupts it. The κ–

formalism enables the use of algebra to find the neuron's depolarization, as well as the immediate firing frequency of the neuron. Combined with the concept of *time windows*, time intervals where the depolarizing inflow is held constant, spiking neuron simulations can be conducted without the use of numerical integration.

The traditional Numerical Integration Model(NIM) and κM is compared theoretically and experimentally in this report. The analysis of the NIM model shows that the local truncation error has stochastic elements, and that the global truncation error diverges unless the local errors have a expectancy value $\hat{e} = 0$. The κM error is a result of a delayed update from a variable that varies within a bounded domain, producing a bounded error.

The two simulation models were implemented in a common framework, and accuracy comparisons was conducted. These comparisons are relevant since the differences between the models were isolated and potential faults in the common framework affects both models equally. It is shown that in the course of 15 periods of a sinusoidal forcing function, the κM_{100} simulation, a κM simulation with 100 time steps per forcing function period, generally produces more accurate results than a $NIM_{10.000}$ simulation. This is a significant efficiency improvement, as the $NIM_{10.000}$ simulation has a number of time steps that is two orders of magnitude larger than for the κM_{100} simulation. All results imply that this effect becomes larger for longer simulations, making the κM simulation model a significant improvement of today's spiking neuron simulation model.

Appendix A

Mathematical Derivations

A.1 Algebraic Solution to the LIF Neuron's Depolarisation

The subthreshold behaviour of the LIF neuron model can be modelled as a general leaky integrator.

$$\begin{aligned} \dot{v}(t) &= \dot{v}_{in}(t) - \dot{v}_{out}(t) \\ &= I - \alpha v(t) \end{aligned} \tag{A.1}$$

where I represents the input flow and α represents the leakage constant. Laplace transform gives

$$\begin{aligned} sV(s) - v_0 &= \frac{I}{s} - \alpha V(s) \qquad , \qquad v_0 = v(t_0) \\ (s+\alpha)V(s) &= \frac{I}{s} + v_0 \\ V(s) &= \frac{1}{s+\alpha}\left(\frac{I}{s} + v_0\right) \end{aligned}$$

and

$$\begin{aligned} v(t) &= \mathscr{L}^{-1}\Big\{V(s)\Big\} \\ &= \frac{I}{\alpha} - \frac{I}{\alpha}e^{-\alpha t_w} + v_0 e^{-\alpha t_w} \qquad , \; t_w = t - t_0 \end{aligned} \tag{A.2}$$

The value equation for the leaky integrator with initial value v_0 is only valid for time intervals where I and α remain constant. Such an interval is referred to as a *time window*, defined in sec. 3.1.1. The variable that represents time in the equation is measured from the start of the current time window, $t_w = t - t_0$.

A.2 Refraction time and simulator time scale

The inter–spike interval for a neuron consists of two phases. The absolute refraction period and the depolarizing phase (see sec. 3.1.2). Equation (3.3) models the depolarizing phase of the neuron. The equation for the whole inter–spike interval is defined as

$$p_{isi}(\kappa) = p_d(\kappa) + t_r \tag{A.3}$$

where t_r represents the absolute refraction period of the neuron. For the firing frequency of the neuron, $f(\kappa) = p_{isi}^{-1}(\kappa)$, the asymptote is defined by

$$\lim_{\kappa->\infty} f(\kappa) = \lim_{\kappa->\inf} \left(\frac{-\alpha}{\ln\left(\frac{\kappa-\tau}{\kappa}\right) - \alpha t_r} \right) \qquad = \frac{1}{t_r} \tag{A.4}$$

This shows that the absolute refraction period limits the output frequency of the neuron. This is illustrated in fig. A.1.

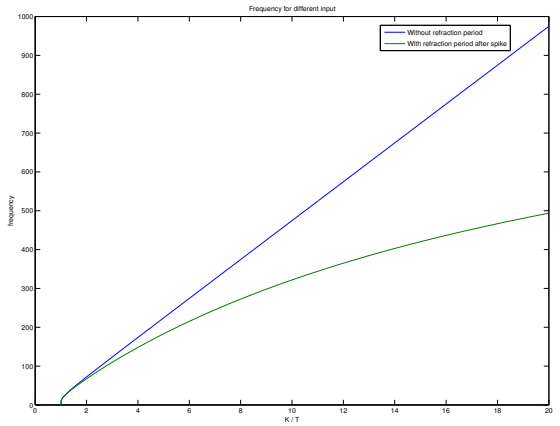


Figure A.1: *Firing frequency of a neuron, with and without absolute refraction period.*

For biological neurons, the maximum firing frequency is about 1000 Hz [2].

$$\lim_{\kappa->\infty} f(\kappa) \approx 1000\,\mathrm{Hz} \tag{A.5}$$

If this is defined as the maximal firing frequency of the artificial neuron, the corresponding absolute refraction period t_r can be found by equation A.4.

$$t_r = \frac{1}{1000\mathrm{Hz}} = 1\,\mathrm{m}s \tag{A.6}$$

If the absolute refraction period is defined to be $1ms$, it is convenient to define the size of a time step to have the same size. In this case, the absolute refraction period can be simulated in NIM by simply blocking input for one time step after the simulated action potential. This consideration is not necessary for κM.

A.3 Activation level recalculation

The concept of edge transmissions as the derivative potentially gives an increase in the efficiency of the simulation, as only the necessary additions have to be executed. The value is found as the sum of all such edge transmissions, and the effect of an altered activation level is computed after the time step. As the activation level is found as the sum of all preceding edge transmissions, small numerical errors is also integrated and could give a large deviation from the correct activation level. Because of this, an adaptive mechanism for recalculation of the activation variable κ is devised.

The size of the error is hard to estimate, as it can vary with the hardware architecture, the system load and the number of input transmissions to the node in question. Because of this, the number of time steps between each recalculation in a node is designed to be adaptive. When the activation variable has a small deviation from the actual activation level, the interval to the next recalculation can be set higher than if the deviation is large.

It is important to limit both the minimal and maximal period between recalculation of κ. This is achieved by the altered sigmoid function (A.7).

$$p_e(E) = (c_1 + c_2) - \frac{c_2}{1 + e^{-(c_4 \cdot E - c_3)}} \tag{A.7}$$

From equation (A.7), it can be observed that the altered sigmoid function has a maximal value of $c_1 + c_2$. In fig. A.2, $c_1 = 100$ and $c_2 = 250$ gives the maximal interval of 350 time steps between recalculation. Because of a small value for the κ errors while experimenting with this aspect, the minimal period between recalculations was set to $c_1 = 100$ iterations. This can easily be adjusted if κ errors become an issue.

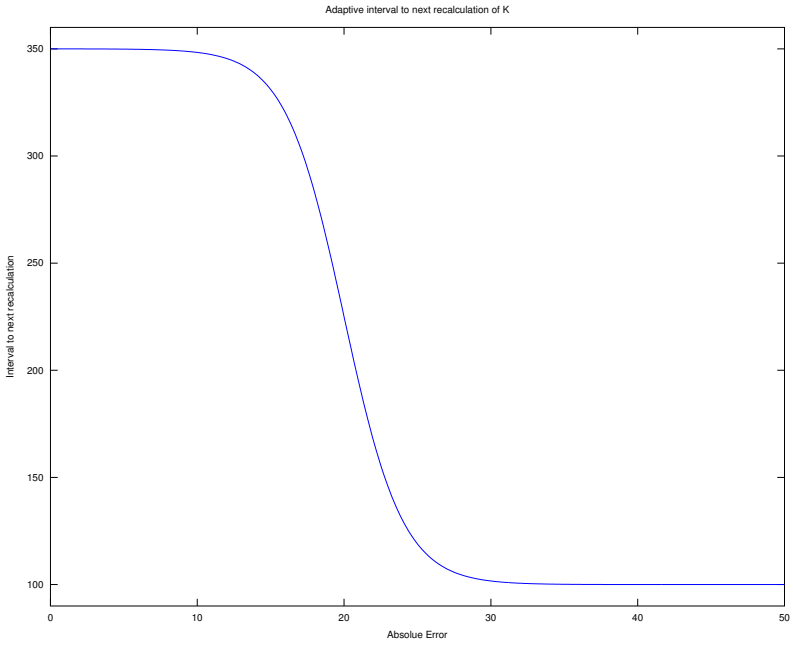


Figure A.2: *Plot of the altered sigmoid function* (A.7) *with* $c_1 = 100$, $c_2 = 250$, $c_3 = 10$ *and* $c_4 = 0.5$. *The minimal interval is given by* c_1 *and the maximum period by* $c_1 + c_2$.

Appendix B

Implementation Details

B.1 Log, for Comparison

For a comparison between the two models, the considered variables are logged during the execution of *auroSim*. This is done by file streams for each of the compared variables, registered as *private* members of the *i_auron* class. The log with most importance for later sections is the one concerned with the node's depolarization, and will be the presented example of this section.

The public member funtion *writeDepolToLog()* takes care of writing the node's depolarization to the *private* log stream. Because the two models represents depolarization differently, this function is pure virtual in *class i_auron* and overloaded in the derived *s_auron* and *K_auron*.

```
1  inline virtual void s_auron::writeDepolToLog() const
2  {
3      // Handle resolution for the depol-logfile:
4      static unsigned long uIterationsSinceLastWrite = 0;
5
6      // Unless it is time for writing to log, return.
7      if((++uIterationsSinceLastWrite > uNumberOfIterationsBetweenWriteToLog)){
8          depol_logFile   <<time_class::getTime() <<"\t"
9                          <<dAktivitetsVariabel <<"; \t #Depolarization\n" ;
10         // Reset counter
11         uIterationsSinceLastWrite = 0;
```

```
12      }else{
13          return;
14      }
15  }
```

The presented source code shows the *writeDepolToLog()* funtion for the *s_auron* class. The log file is implemented with a maximal resolution limit for the log, so that the file log is written every *uNumberOfIterationsBetweenWriteToLog*'th time step. This is done to make the execution of the log files simpler to handle for the computer, and is designed to limit the number of log entries to *LOG_RESOLUTION*, defined for the precompiler. The log is written as a Octave(similar to Matlab) executable scrips, and the values are written in the syntax of a matrix. The first column represents time and the second hold the depolarization value for that time. In this way, the values can be plotted directly by a plot command in octave.

The destructor of an *i_auron* object finalize the log so than it is executable in octave. It closes the parenthesis of the matrix, before it inserts commands to plot the result and save the resulting figure. All figures with the caption *"(Generated by* auroSim*)"* comes from the execution of such log files or modified versions.

```
1  i_auron::~i_auron(){
2      ...
3      depol_logFile    <<"];\n\n"
4              <<"plot(data([1:end],1), data([1:end],2), \"@;Depolarization;\");\n"
5              <<"title \"Depolarization for auron " <<sNavn <<"\"\n"
6              <<"xlabel Time\n" <<"ylabel \"Activity variable\"\n"
7              <<"akser=([0 data(end,1) 0 1400 ]);\n"
8              <<"print(\'./eps/eps_auron" <<sNavn <<"-depol.eps\', \'-deps\');\n"
9              <<"sleep(" <<OCTAVE_SLEEP_ETTER_PLOTTA <<"); "
10             ;
11     depol_logFile.flush();
12     depol_logFile.close();
13     ...
14 }
```

To be certain that all logs are finalized correctly, an automatic destruction of all *i_auron* objects is conducted before the program terminates. This is done in the static member function *i_auron::callDestructorForAllAurons()*, registered at glib's *atexit(void (*)(void))* function. When the program terminates normally, either

by returning from main or with an *exit(int)* function, *i_auron::callDestructorForAllAurons()* is called, calling the destructor for all constructed auron objects.

B.2 The Sensory Neuron

The sensory neuron is a simple way of setting up replicable experiments. A sensory neuron can be implemented by eq. (3.6), where $\xi_i(t_n)$ represents the sensory input at time t_n. As long as the sensory neuron does not receive other input and $\xi_i(t_n)$ is defined by an algebraic function, it is possible to attain the algebraic solution to the neuron's depolarization.

In *auronSim*, a sensory auron is instantiated from a class derived from one of the two model–specific auron classes. The sensory auron contains two important elements; A function pointer to the sensory function and the *static* list *pAllSensoryAurons*. To introduce these elements, the constructor of the NIM sensory neuron is presented.

```
1 s_sensory_auron::s_sensory_auron( std::string sName_Arg , const double& (*
      pFunk_arg)(void) ) : s_auron(sName_Arg)
2 {
3     // Assign the sensory function to the object's function pointer:
4     pSensoryFunction = pFunk_arg;
5     // Add a [this]-pointer to the static s_sensory_auron::pAllSensoryAurons:
6     pAllSensoryAurons.push_back(this);
7 }
```

The constructor takes a function pointer as an argument, assigning it to the member pointer function of type *const double& (*pSensoryFunction)(void)*. It also inserts the node's address as an element in *pAllSensoryAurons*. Before *time_class::doTask()* iterates time, the return value from a call to the dereferenced function *(*pSensoryFunction)()* is sent to the node's *s_dendrite::newInputSignal(double)* for all elements in the list *pAllSensoryAurons*.

This design makes it possible to execute different experiments relatively effortlessly, and it is simpler to carry out a proper analysis of the

accuracy of κM and NIM. The sensory neuron class is useful when experiments on the accuracy of the two simulation models are designed in chapter 5. Experiments can also be conducted by the reader, by declaring sensory functions with the presented format and sending the address to this function to the constructor of *sensoryAuron*s.

Appendix C

Other Results

When the activation level is below threshold, the simulated depolarization have the opportunity to level out on some value. This conceals the simulation error, and have therefore been excluded from the main text. An experiment where κ goes below the firing threshold was conducted and is included in this appendix for the sake of completeness(fig. C.1).

Fig. C.2 shows the difference in absolute error for the second part of experiment 2, between the $NIM_{10.000}$ simulation and the κM_{100} simulation. A positive value means that the κM_{100} have the largest error for the corresponding spike, while a negative value means that the $NIM_{10.000}$ simulation produce the largest error. The cumulative property of the NIM error and the stability property of the κM error can be observed by an increasingly negative value for the difference in absolute error.

C.1 An experiment where $\kappa \in [0.5\tau, 2.5\tau]$

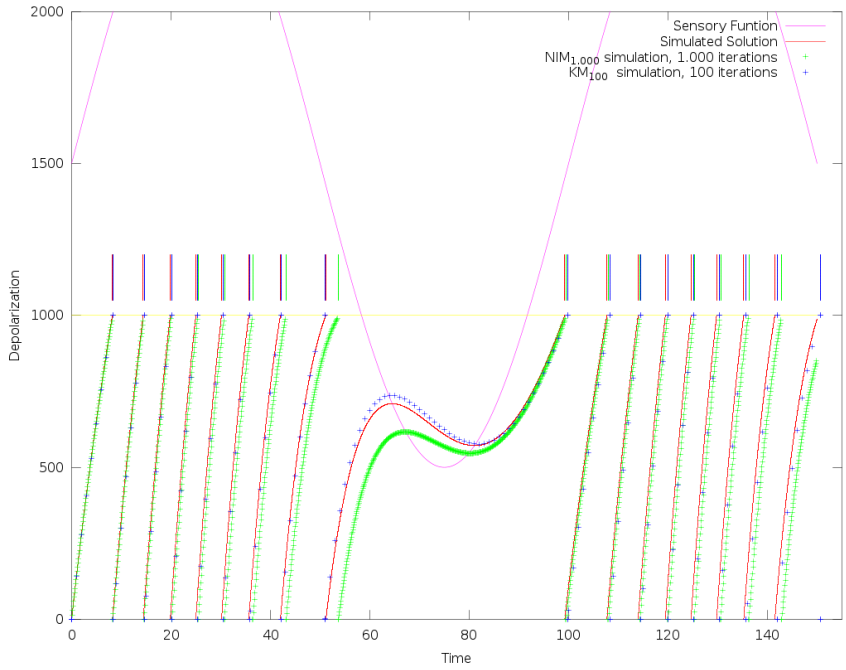


Figure C.1: *When κ is below the firing threshold, the depolarization value has more time to reach the final value κ. When comparing with fig. 5.4, one can observe that the error is smaller in this experiment than in experiment 2. The κM simulation still produces more accurate results than a NIM simulation with ten times the number of time steps. (Generated by auroSim)*

C.2 Difference in absolute error in experiment 2b, $NIM_{10.000}$ and κM_{100}

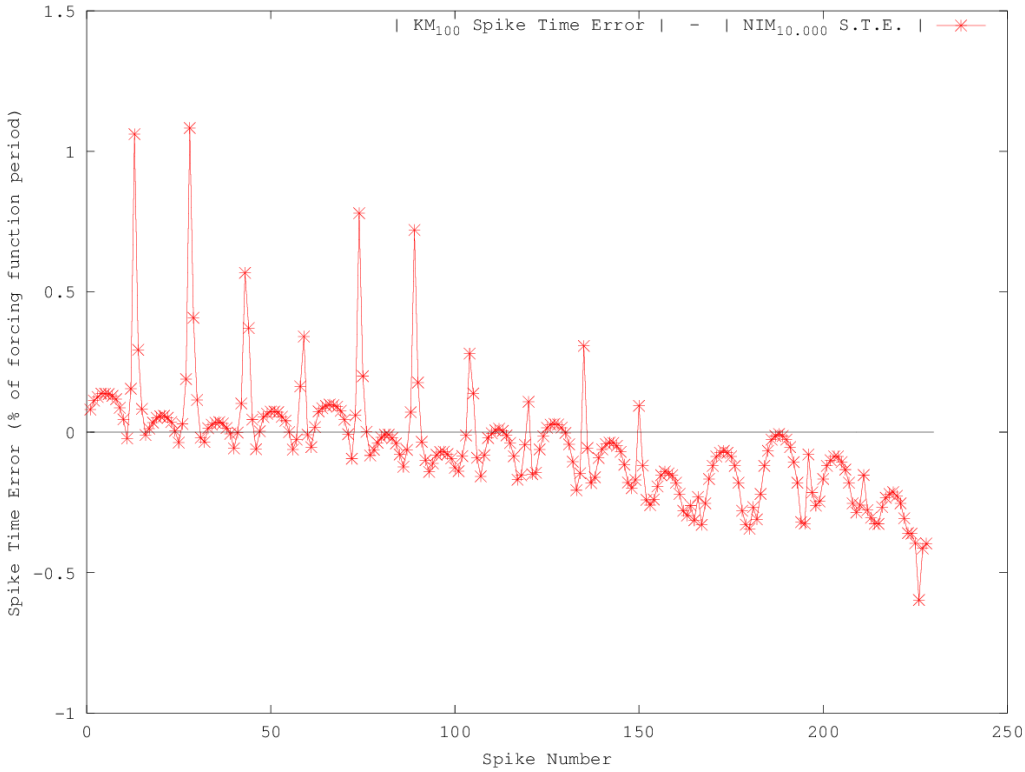


Figure C.2: *The difference in absolute spike time error of a $NIM_{10.000}$ simulation and a κM_{100} simulation for experiment 2b, $f(x) = |E_{\kappa M}| - |E_{NIM}|$. After only* 150 *spikes, the κM_{100} simulation have a smaller error than the $NIM_{10.000}$ simulation, something that can be observed by noting that $f(x)$ becomes negative $f(x) < 0 \Leftrightarrow |E_{\kappa M}| < |E_{NIM}|$.*

C.3 Result from 'time' command, section 4.4.1

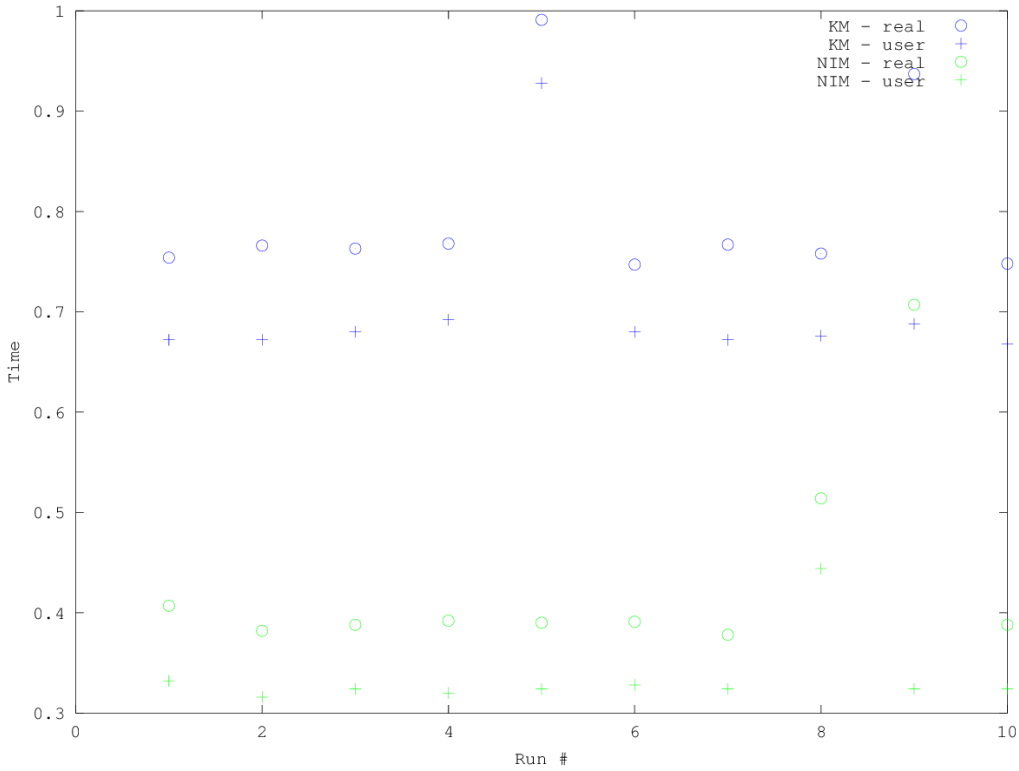


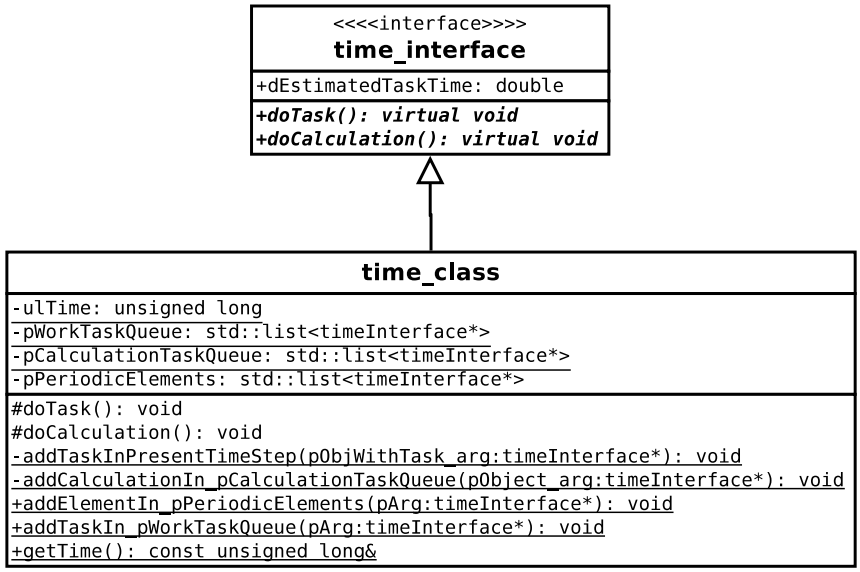
Figure C.3: *Results of 'time' command for ten runs of auroSim$_{KM}$ and auroSim$_{NIM}$, for simulations with the same temporal resolution as the simulated solution in experiment 2. Every second run was with auroSim$_{NIM}$ and the other was a auroSim$_{KM}$ run, in order to minimize the possibility that a global system load would effect the two models differently.*

Appendix D

UML Class Diagrams

To make this report a better documentation of *auroSim*, the UML class diagrams of the most important classes have been included in this appendix. In addition to the different subelement classes, the UML diagram of *time_class* is presented in this appendix. All elements are derived from class $time\_interface$, making all elements inherit the pure virtual functions $doTask()$ and $doCalculation()$. Unless these are overloaded in the derived class, that class is also abstract and no instances of it can be made from it.

D.1 Time Class



time_class

Figure D.1: *The main response of class* time_class *is all aspects of simulation time. pWorkTaskQueue have all objects with tasks, including an object of time class, whose task's main responsibility is to iterate time. Most elements of* time_class *is declared* static*, and have a class–wide scope.*

D.2 Node Subelement Classes

The artificial neuron has a design like the functional lay out of a biological neuron, as illustrated in fig. 2.1. This gives the design presented in fig. 4.4. The UML class diagram of the different subelement classes is presented in this appendix.

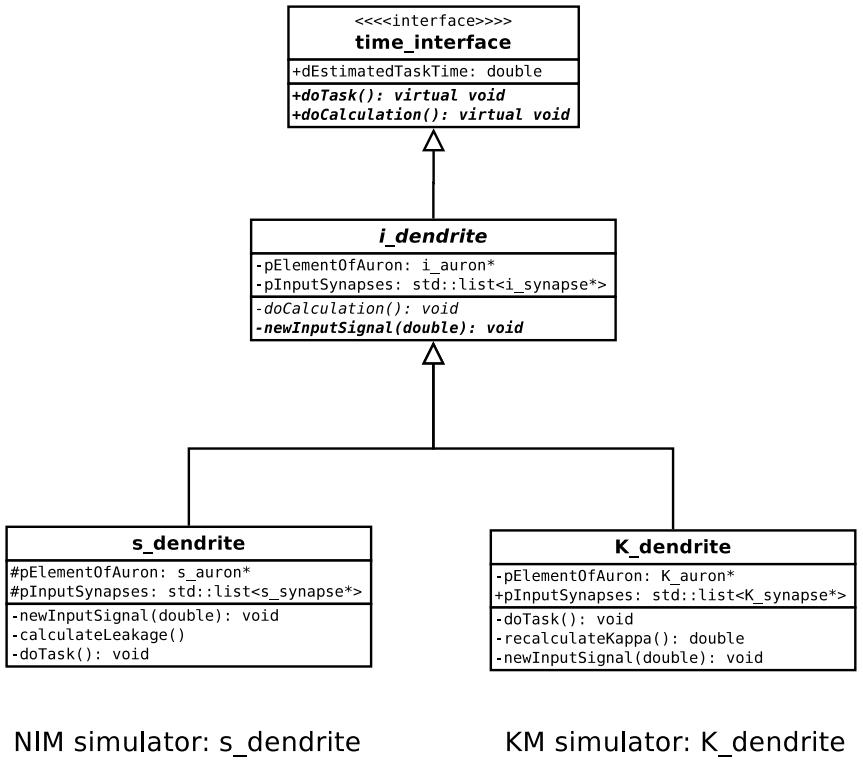


Figure D.2: *UML class diagram for the dendrite subelement.*

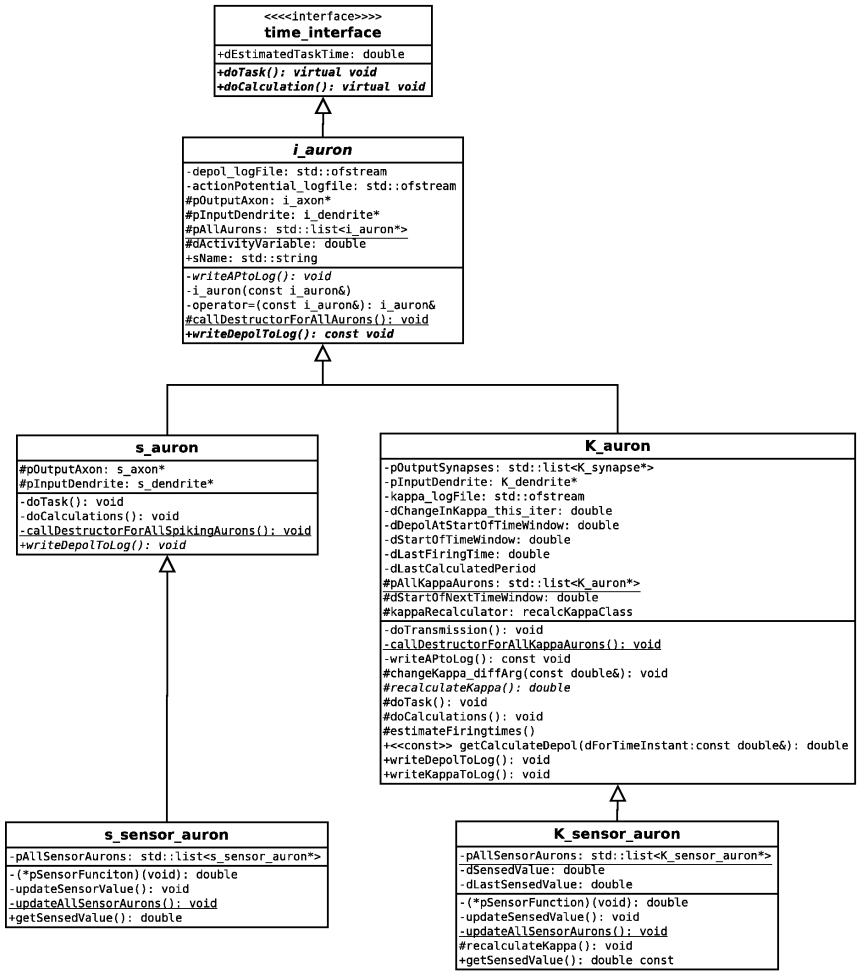


Figure D.3: *UML class diagram for the auron subelement. The sensory auron, used in the experiments, is a specialization of the auron.*

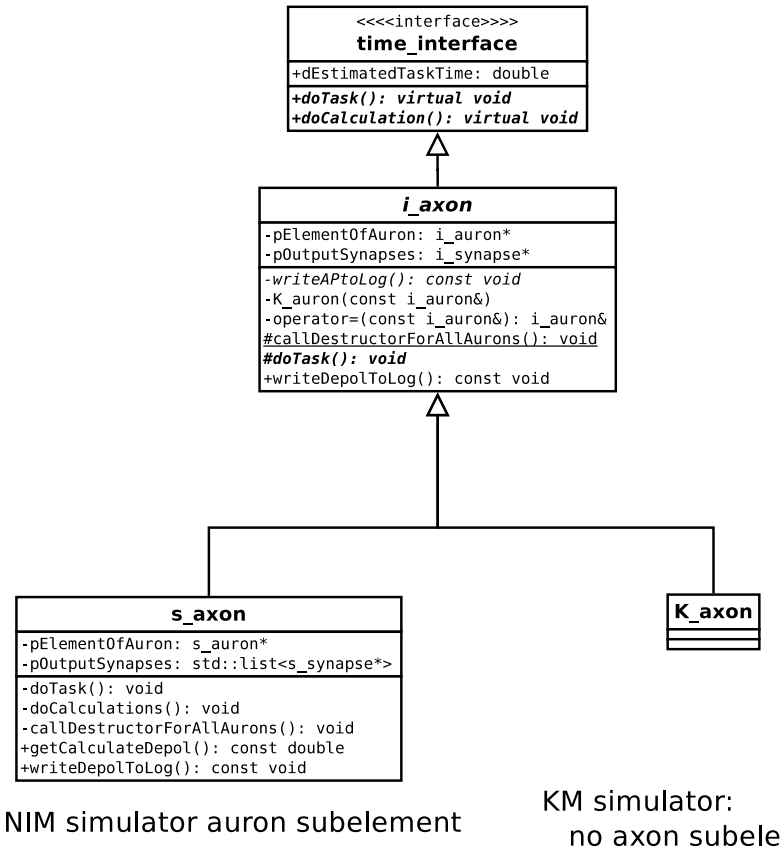


Figure D.4: *UML class diagram for axon subelement. It is found that κM's ability to schedule tasks can be used to simulate spatio–temporal effects like axonic transmission. The axon is therefore not nessecary in the κM implementation.*

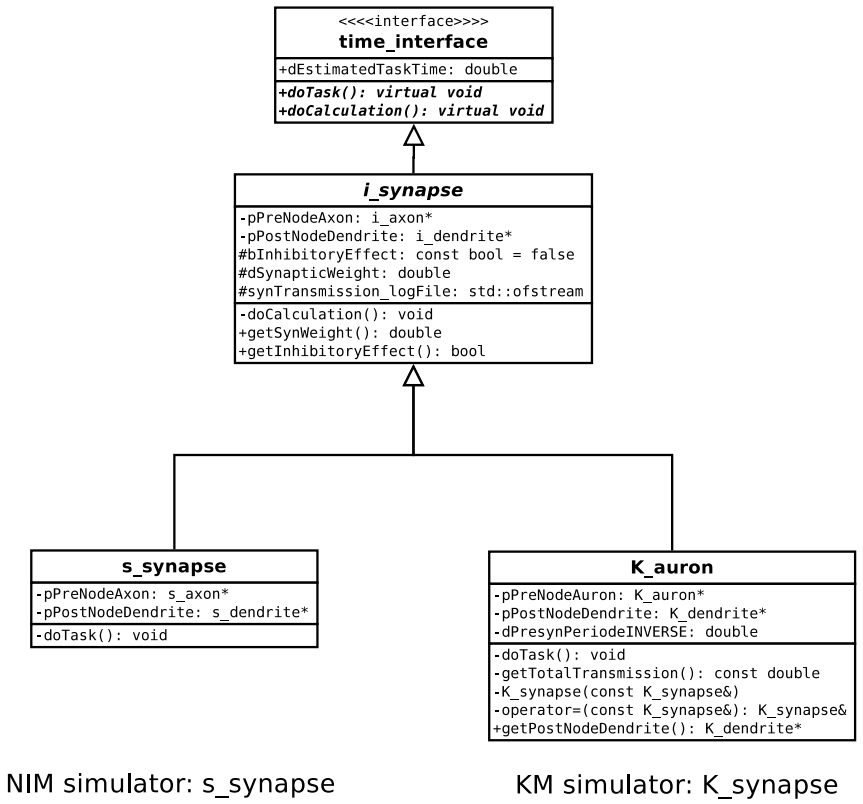


Figure D.5: *UML class diagram for synapse subelement. Both K_synapse and s_synapse is derived from the abstract class i_synapse.*

Bibliography

[1] George J. Augustine, David Fitzpatrick, and Dale Purves. *Neuroscience*. Sinauer Associates, 4th edition, 2004.

[2] Mark F. Bear, Barry Connors, and Michael Paradiso. *Neuroscience: Exploring the Brain (Third Edition)*. Lippincott Williams & Wilkins, third edition, 2006.

[3] Razvan V. Florian. Biologically inspired neural networks for the control of embodied agents. Technical report, Center for Cognitive and Neural Studies (Coneural), 2003.

[4] Ester P. Gardner and John H. Martin. Coding of sensory information. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 411–429. Elsevier, New York, 2000.

[5] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 1 edition, August 2002.

[6] B Gustafsson, H Wigstrom, WC Abraham, and YY Huang. Long-term potentiation in the hippocampus using depolarizing current pulses as the conditioning stimulus to single volley synaptic potentials. *J. Neurosci.*, 7(3):774–780, 1987.

[7] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[8] Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*, chapter 4. Wiley, new edition, 1949.

[9] A. J. Hudspeth. Hearing. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 590–613. Elsevier, New York, 2000.

[10] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31 –44, mar 1996.

[11] Eric R. Kandel. The brain and behavior. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 5–18. Elsevier, New York, 2000.

[12] Eric R. Kandel. Cellular mechanisms of learning and the biological basis of individuality. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 1247–1279. Elsevier, New York, 2000.

[13] Eric R. Kandel. Nerve cells and behavior. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 19–35. Elsevier, New York, 2000.

[14] Eric R. Kandel and Steven A. Siegelbaum. Overview of synaptic transmission. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 175–186. Elsevier, New York, 2000.

[15] Eric R. Kandel and Steven A. Siegelbaum. Synaptic integration. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 207–228. Elsevier, New York, 2000.

[16] John Koester and Steven A. Siegelbaum. Membrane potential. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 125–149. Elsevier, New York, 2000.

[17] John Koester and Steven A. Siegelbaum. Propagated signalling: The action potential. In Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell, editors, *Principles of Neural Science, 4th edition*, pages 150–169. Elsevier, New York, 2000.

[18] Erwin Kreyszig. *Advanced Engineering Mathematics 8th Edition*, chapter 17 - "Numerical Methods in General". John Wiley & Sons, Incorporated, 1999.

[19] Per R. Leikanger. Artificial neural network models. Term project (nevr3004), Kavli institute/Center of the Biology of Memory, NTNU, May 2010.

[20] Per R. Leikanger. The role of stdp in memory formation. Term project (nevr3003), Kavli institute/Center of the Biology of Memory, NTNU, March 2010.

[21] Per R. Leikanger. Synaptic plasticity. Term project (nevr3001), Kavli institute/Center of the Biology of Memory, NTNU, October 2010.

[22] Per R. Leikanger. Development and assessment of a novel model for artificial neural networks. Term project, Department of Engineering Cybernetics, NTNU, july 2011.

[23] Per R. Leikanger. *https://github.com/leikanger/masterProject*. Commit id: '4f771ad1665b499...' in branch *master*, 2012.

[24] W.B. Levy and O. Steward. Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus. *Neuroscience*, 8(4):791 – 797, 1983.

[25] Wolfgang Maass and Technische Universitaet Graz. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10:1659–1671, 1997.

[26] Robert C. Malenka. Synaptic plasticity in AMPA receptor trafficing. *Annals of the New York Academy of Sciences*, 2003.

[27] Abigail Morrison, Sirko Straube, Hans Ekkehard Plesser, and Markus Diesmann. Exact subthreshold integration with continuous spike times in discrete-time neural network simulations. *Neural Comput.*, 19:47–79, January 2007.

[28] Peter E. Gordon Paul Easton. Stabilization of hebbian neural nets by inhibitory learning. *Biological Cybernetics*, 51(1):1–9, 1984.

[29] Helmut Riedel and Detlev Schild. The dynamics of hebbian synapses can be stabilized by a nonlinear decay term. *Neural Networks*, 5(3):459 – 463, 1992.

[30] Edmund T. Rolls and Alessandro Treves. *Neural Networks and Brain Function*. Oxford University Press, USA, 1 edition, January 1998.

[31] Sandhya Samarasinghe. *Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*. AUERBACH, September 2006.

[32] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 3 edition, February 2000.

[33] Dean V. Buonomano Uma R. Karmarkar, Mark T. Najarian. Mechanisms and significance of spike-timing dependent plasticity. *Biological Cybernetics*, 87(5):373–382, 2002.

[34] Robert Waltereit and Michael Weller. Signaling from camp/pka to mapk and synaptic plasticity. *Molecular Neurobiology*, 27:99–106, 2003. 10.1385/MN:27:1:99.