Simulation of neuronal networks using NEST

Simulering av nervcellnätverk med NEST

Johan Hake

Norges Landbrukshøgskole
Institutt for tekniske fag

Ås, augusti 2003

Preface

This thesis is a partial fulfillment of the Cand. Scient. degree in Physics at the Agricultural University of Norway. It has been written at the Department of Agricultural Engineering, with professor Gaute Einevoll as main supervisor and associate professor Hans Ekkehard Plesser as second supervisor.

I would like to thank professor Einevoll, who has not only been my main supervisor, encouraging me and giving me good and thoughtfull feedbacks, but the person who first introduced me to the thrilling and interesting subject of neurophysics. Without him, I would probably not have written this thesis. I would also like to thank associate professor Hans Ekkehard Plesser. His vast knowledge from shortcuts in emacs and other basic but necessary things, to deeper understanding of analytical and numerical methods, has been a great help for me during my work. I also want to thank both for coping with my impatience and need for doing it in my way and I hope I can work together with them in the future.

When I learned using the excellent operating system of Linux, the programing language of C++ and finally the publishing tool of LaTeX, I have also had great help from my fellow students. Thanks!

I also want to thank my wonderful wife Hanna and our comming child, who she bears. Hanna has *really* coped with me during my *not-so-good-days*, especially in the writing process. The child who shall see the light of the day this autumn, has also been a major inspiration for me.

I want to finish with citing Gaute, who with a grin on his face, often reminded me during the not-so-easy times in the process.

The more it hurts the more you learn!

Ås, August 13, 2003

Johan Hake

Summary

This thesis deals with the implementation of efficient and reliable models of spiking neurons and conductance-based synapses, present in the *dorsal lateral geniculate nucleus* (LGN) in a cat, in the framework of the NEST neuronal network simulator.

The *leaky-integrate-and-fire-or-burst* (LIFB) model is used to model the special bursting activity present in neurons in the LGN, and conductance-based models for three types of synaptic receptors, AMPA, $GABA_A$ and $GABA_B$ are presented. Two types of integration methods, the general purpose Runge-Kutta (RK) method, and the exact integration method are used to integrate these models. The RK methods are robust and extensively used methods that provide a formal framework for error analysis. The second (RK2) and fourth (RK4) order of these methods are used. The method of exact integration is a precise and efficient method that gives a structure for fast registration of presynaptic spikes and lumping of conductances, when used to integrate the synaptic conductances.

A thorough error analysis is presented, revealing five discontinuities in the LIFB model, each introducing an error when integrated, and one technical error occurring when a presynaptic spike is registered. A generic procedure for handling discontinuity errors are introduced and implemented to handle two out of five of these errors. The technical error is handled by using the precise spike time when a presynaptic spike is registered.

In this thesis an effective procedure for handling conductance-based synapses in NEST is presented. It does not use the general but time consuming event handling system in NEST to communicate the conductances between a synapse and the receiving neuron, but rather places the synapse inside the receiving neuron, so direct communication between these are possible.

The extensions to NEST implemented as part of this thesis will allow for more realistic simulations of the LGN circuit than previously possible.

Sammandrag

Den här hovedoppgaven behandlar effektiva och tillförlitliga implementeringar av modeller av fyrande neuroner och konduktansbaserade synapser, vilka är närvarande i *dorsal lateral geniculate nucleus* (LGN) i en katt, i den neurala nätverks simulatorn NEST.

Leaky-integrate-and-fire-or-burst (LIFB) modellen används för att simulera den speciella *bursting* aktiviteten som finnes i neuroner i LGN. Konduktansbaserade modeller för tre olika typer synaspsreseptorer, AMPA, $GABA_A$ och $GABA_B$ presenteras. Två olika integrerings metoder, den generella Runge-Kutta (RK) metoden och exakt integrering används för att integrera dessa modeller. RK metoderna är robusta och används av många och de skänker oss ett ramverk för numerisk fel annalys. RK metoder av andra orden (RK2) och fjärde orden (RK4) används. Exakt integrering är en precis och effektiv metod som ger oss en struktur för effektiv registrering av presynaptisk fyring. När denna metoden används för att integrera den synaptiska konduktansen blir det också möjligt att hålla flera konduktanser i samma variable, så kallad *lumping conductances*.

En nogrann felanalys avslöjar fem diskontinuiteter i LIFB modellen, som var och en introducerar ett numerisk fel när den blir integrerad, och ett tekniskt fel som uppstår när en presynaptisk fyring blir registrerad. En generell procedur för hantering av diskontinuitetsfel blir utväcklad och implementerad för att hantera två av dessa. Det tekniska felet blir behandlat med att använda precis fyringshantering, när en presynaptisk aktionspotential skall registreras.

I denna hovedoppgaven presenteras ett effektivt sätt att hantera konduktansbaserade synapser i NEST. Det generella men tidskrävande händelseshanteringssystemet i NEST används inte för att kommunisera konduktansen mellan en synapse och det mottagande neuronet. Enskilda synapser blir hällre plasserade inne i det mottagande neuronet, så att direkt kommunikation mellan dessa blir möjlig.

Utvidgningen av NEST som är gjord som en del av denna hovedoppgaven gör det möjligt att företa mer realistiska simuleringar av den neurala kretsen i LGN, än vad tidigare varit möjligt.

Contents

Preface iii

Summary v

Sammandrag vii

1 Introduction 1

2 Early visual pathway 3
2.1 Introduction . . . 3
2.2 Retinal circuit . . . 4
2.3 Thalamus and LGN . . . 5
2.3.1 Tonic and burst firing . . . 6
2.3.2 Synaptic connections in the LGN . . . 8

3 Models 11
3.1 Introduction . . . 11
3.2 Model of relay cell, LIFB model . . . 11
3.3 Models of synapses . . . 15
3.3.1 Conductance in ionotropic synapses . . . 15
3.3.2 Conductance in metabotropic synapses . . . 19

4 NEST 23
4.1 Introduction . . . 23
4.2 Large networks and fixed time steps . . . 23
4.3 Nodes, events and synapses . . . 24
4.4 Interface . . . 24
4.5 Extendability . . . 24

5 Numerical implementations 25
5.1 Introduction . . . 25
5.2 LIFB model . . . 26
5.3 Synaptic input . . . 28
5.4 h variable . . . 34

5.5 Error analysis 35
5.5.1 Registration of a presynaptic spike 37
5.5.2 Onset and offset of an external current 39
5.5.3 Threshold passings 40
5.5.4 Incoming spikes and synaptic currents 45

6 Result of testing **49**
6.1 Test setup 49
6.2 Results 52

7 Discussion **59**

A Integration methods **63**
A.1 Runge-Kutta metods 63
A.2 Exact integration 68

B Interpolation parameters **71**

C Selected C++ code **73**
C.1 Introduction 73
C.2 LIFB Neuron with RK4TS method 73
C.3 Abstract synapse class 85
C.4 $GABA_A$ synapse 90
C.5 $GABA_B$ synapse 94
C.6 Spike event buffer 99

D Selected SLI code **103**
D.1 Introduction 103
D.2 The fifth test 103

References **107**

Chapter 1

Introduction

The understanding of information processing abilities of biological neural networks is a huge task, which during the last decade has been catalyzed by the fast development of computer power. With use of modern clustering technologies, networks of hundreds of thousands neurons can be simulated (Diesmann et al. 1999). To accelerate such large simulations, simple models of neurons are used. A type of model that is commonly used is the *integrate-and-fire* models. These models avoid the biophysical description of an action potential and only simulate the subthreshold dynamics of the neuron. Long before the mechanisms for action potentials were understood, Lapicque (1907) presented a basic model of an integrate-and-fire neuron. Nearly hundred years later these type of models is still used to simulate neural activity.

Our group at the Agricultural University of Norway at Ås, is investigating the information processing abilities of a circuit in the *early visual pathway* (EVP) of a cat, namely the dorsal lateral geniculate nucleus (LGN). One goal for the group is to develop a network model of the LGN circuit, based on spiking neurons, which we hope will tell us both qualitative and quantitative features of the real biological network. A step toward this goal is to implement an integrate-and-fire model of one of the most important cells in LGN, the *relay cell*. A few years ago Smith et al. (2000) suggested such a model, the *leaky-integrate-and-fire-or-burst* (LIFB) model. This model reproduces the significant feature of bursting activity of the relay cell. We plan to use this model, not only for the relay cell, but as a general working horse for other neurons with bursting activity in the circuit too. Three different synaptic models, that are present in the circuit of LGN, shall also be implemented. These are all conductance based models and are built upon models developed by Destexhe et al. (1998). All the models are implemented in the NEural Simulation Technology (NEST) initiative, which is a simulation program for spiking-neuron networks, written in C++ by Diesmann and Gewaltig (2003). NEST takes advantage of advanced, multiprocessor threading technology to simulate large

networks and it provides a very robust framework for network simulations with spiking neurons.

The overall goal of this thesis is to implement efficient and reliable models of spiking neurons and conductance based synapses, present in the LGN in a cat, in the framwork of the NEST neuronal network simulator. These models are escential for the development of network models for the LGN circuit.

The issue of efficient and reliable simulations of neural networks has been studied lately by different authors (Hansel et al. 1998, Shelley and Tao 2001, Destexhe et al. 1994 and Rotter and Diesmann 1999). Reliability in the simulation result is crucial when large network simulations are done. If a network is not simulated with sufficient precision, numerical errors are introduced and may undermine any result claimed by the simulation. Also when simulations with tens and hundreds of thousands of neurons are done, it is not enough to have a fast framework for neural communication, that NEST provides us. The implementation of the individual models have to be optimized, to achieve efficient integrations. We are going to combine two different integration methods, the general purpose method of Runge-Kutta (RK) and the fast and reliable integration method of exact integration, with different methods for dealing with numerical errors. By this we can benefit from the special features in our models, without losing reliability and speed when these are simulated.

The thesis is structured as follows. The second chapter contains a brief presentation of the EVP and the LGN circuit we are investigating, and the third chapter presents the models we are using. These are the fundamental building blocks in the network model we want to establish. Chapter 4 is a brief presentation of NEST, the simulation program we are using, and chapter 5 contains a presentation of how the models from chapter 3 is implemented in the different integration methods. This chapter also contains a thorough error analysis of the implemented models. The implementations and the error analysis are then tested in chapter 6. Finally, I discuss what was accomplished and suggest the next steps.

Chapter 2

Early visual pathway

2.1 Introduction

In this chapter we give a physiological introduction of the signal processing circuit we are going to develop models for, namely the feed forward circuit of the *dorsal lateral geniculate nucleus* (LGN), in a cat.

Where nothing else is mentioned this chapter is based on Sherman and Guillery (2001), Sterling (1998), Sherman and Koch (1998) and Dayan and Abott (2001).

The early visual pathway (EVP) in cats[1] carries information from the narrow band of electromagnetic radiation that we call light, through the circuit of the retina and the visual thalamus to the primary visual cortex. From here the information is carried higher up in the brain hierarchy for further interpretation. The EVP does not carry information about every photon striking the retina into the *primary visual cortex* but rather a processed and compressed neural image. The first would consume far too much energy and also would carry a lot of redundant and non-useful information.

There are three parts in the EVP, the *retinal circuit*, the *visual thalamus* or the (LGN) and the primary visual cortex. The retinal circuit receives physical information from the light striking the retina and transforms it into a neural image. The LGN relays this image to the primary visual cortex where it is further processed. A striking property of the EVP is that visual information from neighboring location in visual space are conveyed, through the entire pathway, by neighboring neurons. This means that a map of the visual information striking the retina is created in the primary visual cortex. This map is called the *retinotopic map*, and it places a strong constraint on the signal-processing in the EVP.

This thesis focuses on the feedforward circuit of the LGN, and is not concerned with recurrent signals or signals from other parts of the *central nervous*

[1] This thesis focuses primarily on the EVP of the cat. The general picture of the EVP is the same for most mammals, but that of the cat is the most studied.

system (CNS). Therefore only the parts of the EVP that give us the most important features about the information that reaches the LGN are emphasized.

2.2 Retinal circuit

The EVP starts with the retinal circuit. This is a complex circuit that receives an image of the physical world from the optical system of cornea, pupil and lens, and transfers it into a neural image that is carried by the optical nerve to the LGN. The transformation is done in three stages: *i)* transduction of the physical image by *photoreceptors*, i.e. *cones* and *rods*; *ii)* transmission of these signals by excitatory chemical synapses to *bipolar neurons*; and *iii)* further transmission by excitatory chemical synapses to *ganglion cells*. Axons from the latter form the the optical nerve. At each synaptic stage there are lateral processing neurons called, respectively, *horizontal* and *amacrine* cells. These six different types of neurons give the retinal circuit a vast operating range, from starlight and twilight to daylight signal-processing. The circuit also compresses the optical image, from just being a number of photons hitting the retina, to a more complex code that contain different spatial and temporal structures of the light that strikes the retina. The spatial and temporal structures that are created in the retinal circuit are commonly called the *receptive fields*.

The receptive field of retinal ganglion cells is the small roughly round area in the visual field where a cell transducts information from. It is created by converging inputs from many photoreceptors and bipolar cells and the special properties are created by different synaptic treatments of the signals. The receptive field could either be of ON or OFF type, with the difference that ON types are sensitive to high intensity light in the center and low intensity in the surround of the field, and OFF types are sensitive to low intensity light in the center of the field and high intensity in the surround. In this way the visual signal that is transduced through the rest of the EVP, does not carry information about every photon striking the retina, but rather a compressed image with information about bright and dark areas in the visual field.

Neurons carrying the visual signal through the EVP are divided into two coarse classes, depending on the behavior of the receptive field. One type, X cells, sums up contributions from the field linearly on a long timescale, while the other type, the Y cells, sum up contributions in a fast and transient manner. Another difference is that the spatial resolution of the receptive field of the X cells is higher than the receptive field of the Y cells. This is illustrated in figure 2.1. Here we see the relative size of the different type of ganglion cells, with their corresponding receptive fields, and their response to a light spot.

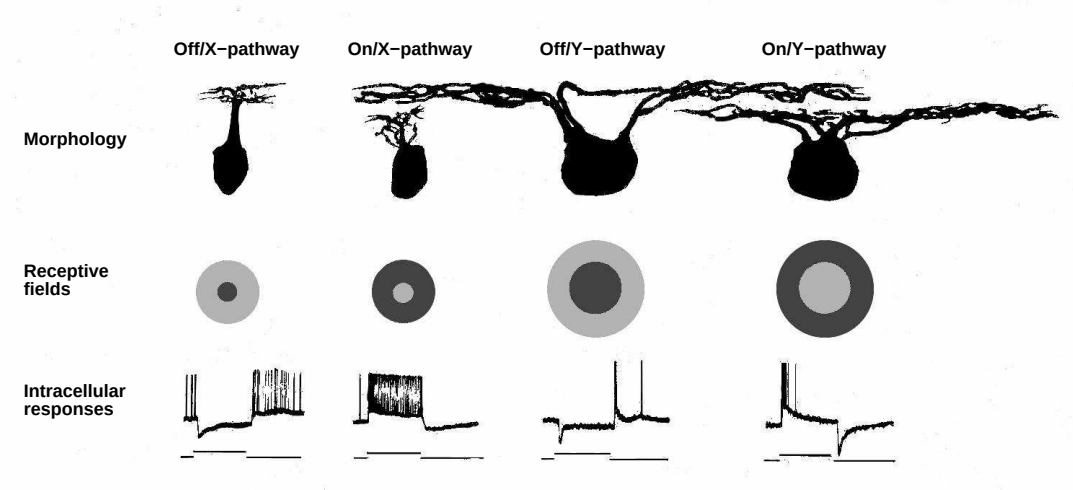

Figure 2.1: *The figure shows the form and function of cat retinal ganglion cells. Ganglion cells in the X-pathway have a narrow dendritic field, and the ones in the Y-pathway a broad one. The cells are stimulated with light in the center of their receptive fields. The response of the ON cells is firing and the OFF cells is suppression. The ganglioncells in the X-pathway give a transient and sustained response and the ganglion cells in the Y-pathway give mainly a transient response. Modified from Sterling (1998).*

We see that the receptive field of the ganglion cells in the X-pathway is much smaller than in the Y-pathway, and they therefore collect input from the retina with higher spatial resolution.

2.3 Thalamus and LGN

After the different temporal and spatial information from the visual image has been coded by the retina, the signal is carried by the ganglion cells to the part of the thalamus called LGN. Here activity from other parts of the CNS have the opportunity to modulate the relay of the signal to the primary visual cortex.

The neural components of the circuit of the LGN can be divided into three components: the external afferent input to the nucleus, the *relay cells* that project to cortex, and the *interneurons*. Figure 2.2 schematically illustrates the circuit of LGN. The external inputs can be divided further into two classes: driving and modulatory inputs (Sherman and Guillery 2001). The driving input is strong and capable of driving the relay cell. The signals coming from the retina are driving input, and it is this information the LGN relays to cortex. The modulatory inputs consist of local feedback from the *thalamic reticular nucleus (TRN)* and from the interneurons in the LGN, feedback from visual cortex and signals from the brainstem. The TRN is a layer of neurons that is

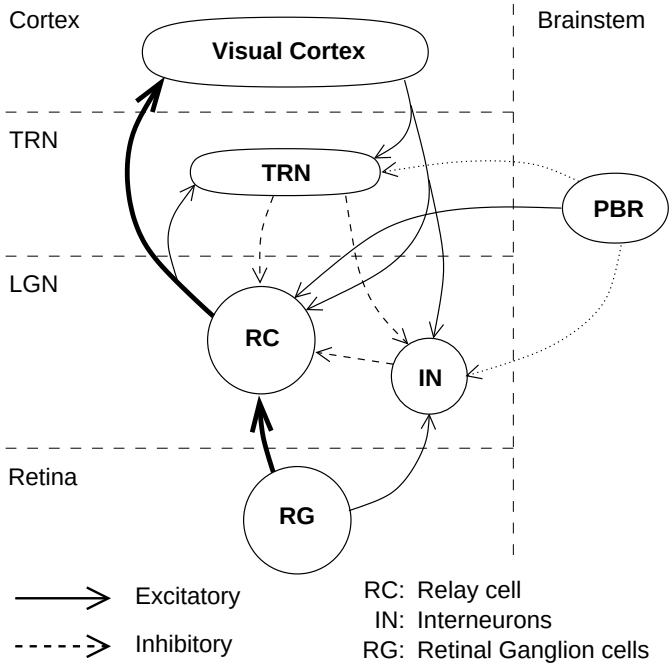

Figure 2.2: *Schematic figure of the LGN circuit. The relay cells receive driving signals from the retina and pass it to cortex, thick line. This signal is modulated by inhibitory signals from interneurons and TRN cells, and by excitatory signals from the visual cortex. The PBR cells, in the brainstem, innervate the whole LGN with both excitatory and inhibitory modulatory signals. The figure is redrawn and simplified from Sherman and Koch (1998).*

wrapped around the LGN. It is activated both by ascending input from LGN and by descending input from V1. The TRN makes inhibitory projection on LGN relay cells and interneurons. The brainstem is the lower extension of the brain where it connects to the spinal cord and it controls basic functions such as breathing, digestion, heart rate, blood pressure and being awake and alert. The brainstem sends both inhibitory and excitatory signals to the LGN.

2.3.1 Tonic and burst firing

Thalamic relay cells have two different response modes: *i) tonic* and *ii) burst* firing mode. The first corresponds to a linear response to the input, the latter to a non-linear transient response. The response mode of the relay cell depends on the membrane potential. If the relay cell has a relatively high potential, being relatively *depolarized*, the response mode is tonic. If the cell has a lower potential, being relatively *hyperpolarized* for a while, the response mode is burst. Figure 2.3 shows the response of a cat relay cell, when a constant current was injected. Depending on the initial potential of the cell the response mode is different.

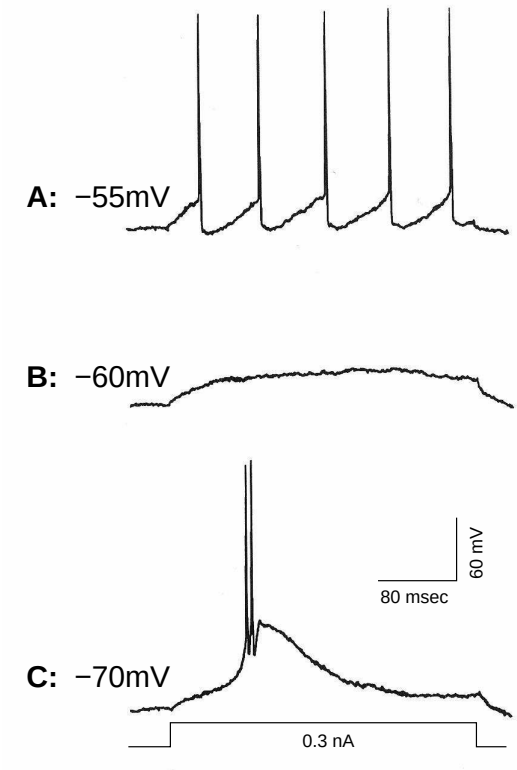

Figure 2.3: *Result from a constant current injection in a relay cell from a cat, with different initial membrane potentials. A: The injection manages to drive the relay cell with a tonic response. B: The injection does not manage to drive the relay cell and there is no response. C: The T-current is de-inactivated and the injection manages to depolarize the relay cell enough to cross the threshold for the T-current and the relay cell responds with a burst. From Sherman and Koch (1998)*

The spikes fired in tonic mode are ordinary sodium and potassium spikes with high threshold value, around -35 mV, A in figure 2.3. The spikes fired in the burst mode come from a transient calcium current called the T-current or just I_T. The threshold for activation of this current is around -65 mV, and it is therefore called the *low threshold calcium current*. When the current is activated the cell fires in burst mode. This lasts for about 20 msec, then the T-current inactivates and the cell either goes into tonic firing or it stops to fire. When the cell has been hyperpolarized, below -65 mV, for about 100-200 ms then the T-current is de-inactivated. It is then ready to fire a burst, if the potential crosses the low threshold again and activates the current.

It is important to mention that there are several different ion currents in the relay cell that contribute to the signal-processing ability of the relay cell, but the low threshold calcium current is one of the most salient. For more details see Hugenard and McCormick (1992) and McCormick and Hugenard

(1992).

The two different firing modes represent the driving stimuli in different ways. The tonic mode follows the stimuli in a linear way and it represents it more faithfully. If the stimulus is weak the tonic firing is weak, and if it is strong the response is also strong. The representation of the stimulus in burst mode is non-linear. The relay cell fires the same burst regardless of whether the stimuli is weak or strong. Some authors have suggested that a burst records the onset of a visual stimulus to the cortex. The modulatory inputs are important because they have the ability to alter the mode the relay cell is firing in.

2.3.2 Synaptic connections in the LGN

The synaptic connections between two cells decide what information a spike from a *presynaptic* cell delivers to a *postsynaptic* cell. The dynamics of a synaptic connection is therefore essential for the signal-processing ability of a certain circuit. If the synapse is strong, a spike from the presynaptic neuron excerts a strong influence on the dynamics of the postsynaptic neuron.

The range of different synapses in the LGN is vast. From fast, linear and strong *excitatory* synapses (depolarizing the membrane potential of the receiving neuron), to slow non-linear weak *inhibitory* synapses (hyperpolarizing the membrane potential of the receiving neuron). There are still some questions about the synaptic connections in the LGN but the feedforward circuit from the ganglion cell is fairly well mapped out.

As mentioned above, the LGN receives signals from the retina via the retinal ganglion cells, through two different pathways, X and Y. These signals are delivered to the relay cells and the interneurons in the LGN. The interneurons then deliver inhibitory signals to the relay cells. This overall picture is the same for the two different pathways. The X pathway also includes a special connection between the ganglion cell and the relay cell through an interneuron terminal called a *triad*, schematically showed in figure 2.4. The terminal is an appendage of the dendrites of the interneuron. It has been argued that this is electrotonically isolated from the other parts of the dendrites to the interneuron (Sherman and Guillery 2001). This indicates that spikes arriving from the ganglion cell through a triad, do not influence the spiking activity of the interneuron; but see Heggelund (2001).

The dynamics of a synapse are determined by the transmitter substance and the receptors used by respectively the pre- and postsynaptic neurons, to deliver the signal through the synapse. In the feedforward circuit of LGN there are two types of transmitter substances delivering the message of a presynap-

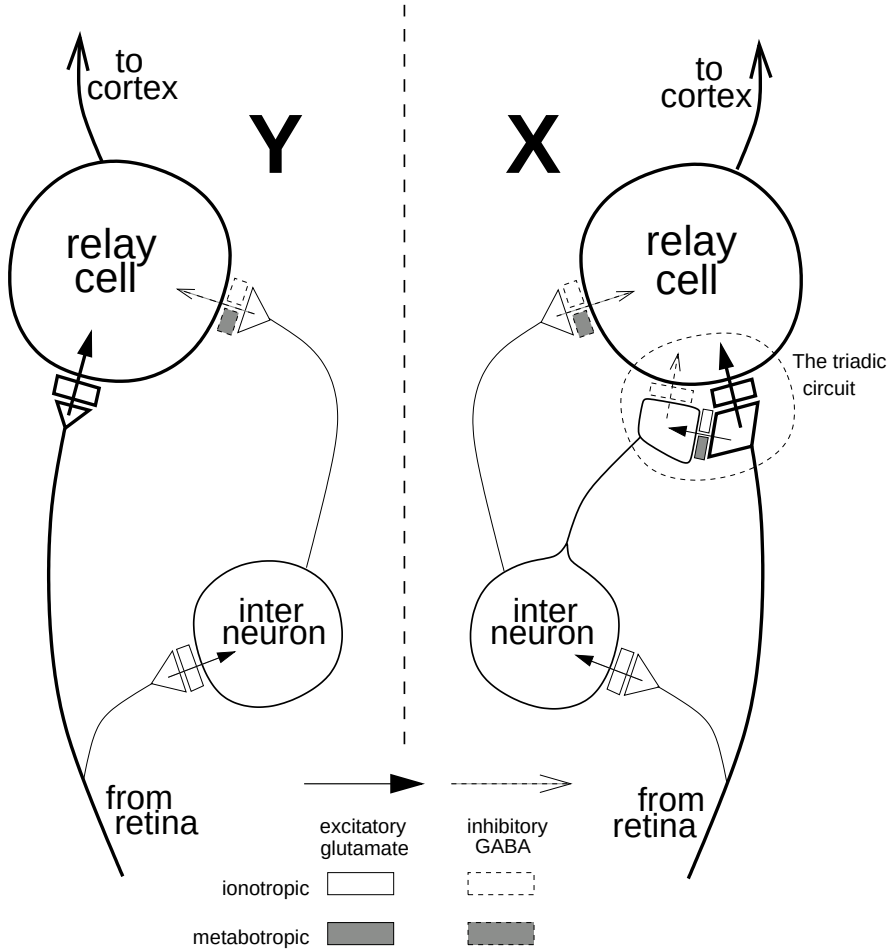

Figure 2.4: *Schematic view of the two different feedforward circuits in the LGN. The driving input follows the bold line. The arrows indicates the direction of the signal, and the boxes the receptor type on the post synaptic side. The special triadic circuit in the X-pathway are showed. For details about the receptors see table 2.1. Modified from Sherman and Guillery (2001).*

tic spike, *glutamate* for the excitatory synapses and *GABA* for the inhibitory synapses. The strength of a synapse is determined by the number of transmitter molecules a spike releases from the presynaptic terminal, by the type and number of receptors at the receiving terminal, and by the geometry of the transporting synapse. The receptors are divided into two types: *i) ionotropic* and *ii) metabotropic*. A ionotropic receptor is situated together with the ion channel the receptor is gating. While transmitter substance is bound to the receptor, the corresponding channel is open. A metabotropic receptor is not situated together with the ion channel, but it controls it by secondary messenger systems. When transmitter substance is bound to the receptor it releases a messenger proteins inside the cell, which then have to diffuse and attach to a second receptor which then opens an ion channel. While the ionotropic

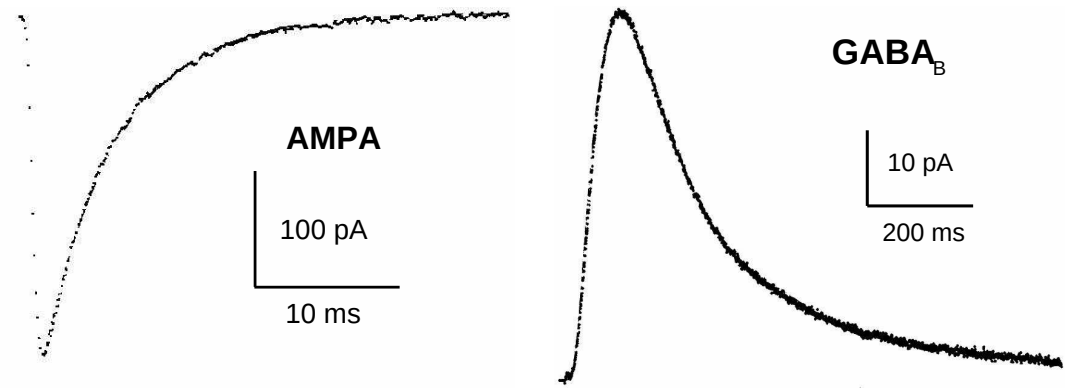

Figure 2.5: *Two synaptic currents showing the difference between a fast and strong ionotropic receptor, AMPA, and a slow and weak, metabotropic receptor, $GABA_B$. Current out of the membrane is defined to be positive, and AMPA is an excitatory synapse, letting positive ions into the cell, therefore the negative sign on the AMPA current. Not the scale of both time and current. Modified from Destexhe et al. (1998).*

synapse is fast and linear the metabotropic is slow and non-linear: it may need strong bursting input to open ion channels, but once the ionchannels are open they stay open for a long time. Figure 2.5 shows two different synaptic currents, one with a ionotropic receptor, AMPA, and one with a metabotropic receptor, $GABA_B$. The AMPA receptor gives a stronger current and is much faster, than the $GABA_B$ receptor.

Figure 2.4 shows the different synapses involved in the feedforward circuit in the LGN together with the type of receptors. The two different receptors receiving glutamate are the ionotropic AMPA receptor, see figure 2.5 and the metabotropic mGluR5 receptor[2]. There are two different receptors receiving GABA in the LGN circuit, the ionotropic $GABA_A$ receptor and the metabotropic $GABA_B$ receptor, see table 2.1.

There is no exact understanding of how the mGluR5 receptors control the release of GABA transmitters from the terminal in the interneuron dendrite in the triadic circuit. It seems plausible that it acts by depolarizing the terminal. This in turn could lead to more GABA transmitter release, but there are no direct evidence of voltage change in the terminal.

[2]m for metabotropic, Glu for glutamate, R for receptor and 5 for the fifth metabotropic receptor type. Together with mGluR1 they form the first group, of three, so far discovered in CNS. They are both excitatory metabotropic receptors. For more detail see Coutinho and Knöpfel (2002).

	X-pathway	**Y-pathway**
RG → RC	AMPA	
RG → IN	AMPA	
IN → RC	$GABA_A$, $GABA_B$	
The triadic circuit		-
GC→ IN	AMPA, mGluR5	
IN → RC	$GABA_A$	

Table 2.1: *Table shows the different receptors on the post synaptic side in the feedforward circuit in LGN. RG: retinal ganglion cell. RC: relay cell, IN: interneuron. Ionotropic receptors: AMPA, $GABA_A$. Metabotropic receptors: mGluR5, $GABA_B$.*

Chapter 3

Models

3.1 Introduction

As described in chapter 2, the important signal processing units in the feedforward circuit in the LGN are the relay cell and the synaptic connections to this cell from the ganglion cell and the interneuron. To simulate signal transfer in this circuit, we must have reliable models of these signal processing units. Different models exist, for different timescales and for different needs of accuracy and usability. If we, for example focus on the mean firing rate of a neuron and want to model this property, firing rate models are useful. On the other hand if we look at more detailed features in a single neuron or small populations of neurons, other more detailed models of neurons and of synaptic dynamics should be used. In this thesis we are dealing with models of the relay cell and for synaptic transmission in the feedforward circuit of LGN, that shall be used in large scale spiking network simulations. Therefore we only consider simplified spiking models of the relay cell and the synaptic transmissions.

In this chapter we present the models we are going to use for the relay cell, and the synaptic transmissions between neurons in the feed forward circuit of LGN. The next chapter are going to present the numerical implementation of these models.

3.2 Model of relay cell, LIFB model

The model used for the relay cell in this thesis is the model presented in Smith et al. (2000), the *leaky-integrate-and-fire-or-burst (LIFB)* model[1]. This is a *leaky-integrate-and-fire (LIF)* model with the T-current included in a phe-

[1]In Smith et al. (2000) the name of this model is Integrate-and-fire-or-Bursts, but to emphasize an important feature of the model we have added a Leaky in front of the name.

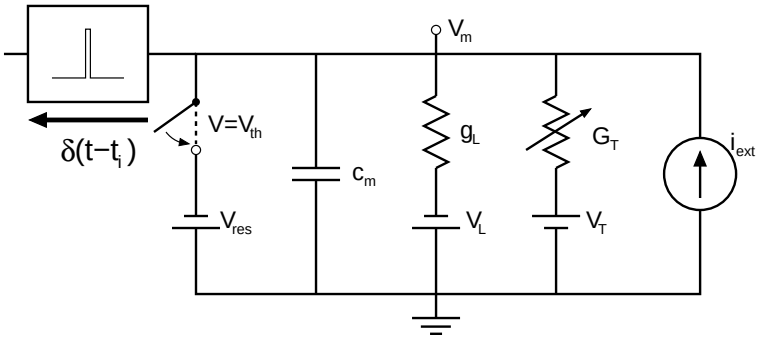

Figure 3.1: *The electrical equivalent circuit of the LIFB model. When the membrane potential V_m, the potential across the capacitor, reaches V_{th}, the switch is turned on and the circuit is short-circuited, resetting V_m to V_{res}. It also sends a delta-spike, $\delta(t-t_n)$, at this time to all its target neurons. $g_{_L}$ is the leak conductance pulling the membrane potential toward the reversal potential, $V_{_L}$, of the leak current. $G_{_T}$ is the variable conductance of the T-current given by, $g_{_T} m_\infty h$ pulling the membrane potential toward the reversal potential, $V_{_T}$, of the T-current. i_{ext} is a variable current source, corresponding to either synaptic currents or external input currents.*

nomenological way, leading to bursting activity. The LIFB model has been used to model other bursting neurons in the LGN too, for example the TRN cell (Smith and Sherman 2002).

A LIF model is a *single compartment* model that assumes that signals are carried by so called δ-spikes between the different neurons. A δ-spike does not carry any information about the width or hight of the action potential. A δ-spike is sent from a LIF neuron to a receiving neuron when the membrane potential crosses a threshold value, V_{th}. After a spike has been sent the potential is reset to a reset value, V_{res}. Performing this reset together with the sending of a δ-spike instead of simulating the whole dynamics of a single spike, simplifies the simulation and accelerate the computation. As many other models of neural dynamics this model is based on the assumption that the membrane of the neuron acts as a capacitor and a resistance between the inside and outside of the neuron, where the intracellular and the extracellular fluids close to the membrane act as the electrode plates of the capacitor. It also assume that the neuron is *electrotonically compact*, i.e. the potential is the same in the whole neuron. The LIFB model is given by

$$\begin{aligned} c_m \frac{dV_m}{dt} &= -i_{_L} - i_{_T} - i_{ext} \\ \text{if } V(t_n^-) = V_{th} &\Rightarrow \delta(t-t_n) \text{ and } V(t_n^+) = V_{res} \ . \end{aligned} \tag{3.1}$$

This model is based on the equation for the potential across two electrode plates in a capacitor, with currents going across the two plates, see the equivalent circuit in figure 3.1. The capacitance of the membrane c_m and the different

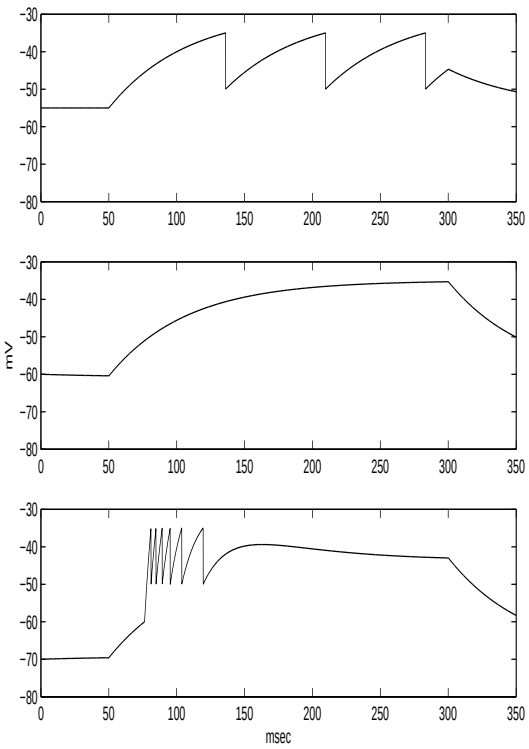

Figure 3.2: *The figure shows how the LIFB model reproduces salient features of real relay cell from the cat, see figure 2.3. The LIFB model responds, to an injected current* $i_{ext} = -0.9\mu A/cm^2$*, given different initial conditions. The current where injected between 50 and 300 ms. The potentials are held respectively on: -55mV, -60mV, and -70mV by injected currents:* $-0.35\mu A/cm^2$, $-0.15\mu A/cm^2$ *and* $0.15\mu A/cm^2$*. Depending on the initial potential the LIFB model responds differently to the injected current. The first responds in tonic mode, the second does not fire at all and the third responds with a burst. The figure show only the sub-threshold dynamics of the LIFB model. During a spike the membrane potential normally reaches 0 mV or more, but the LIFB model do not simulate the dynamics of a spike, but rather reset, the potential when it reaches 35 mV to 50 mV.*

currents are given in specific units[2], allowing equation 3.1 to be used for relay cells of different sizes. The current is defined to be positive if it goes out of the cell. A positive current into the membrane should depolarize the membrane potential. If for example i_{ext} is negative, a positive current into the cell, gives a positive contribution to the derivative of the potential, thus depolarizing the cell.

The leak current, $i_{_L}$, is given by

$$i_{_L} = g_{_L}(V - V_{_L}) \ . \tag{3.2}$$

In the absence of any input only the leak current is active and the membrane

[2]By specific units we refer to specific capacitance, conductance, etc, i.e. capacitance per unit membrane surface area

potential approaches V_L. V_L is therefore the resting potential of the neuron. The T-current, i_T, is given by

$$i_T = g_T m_\infty h \times (V - V_T) \ . \tag{3.3}$$

The dynamics of this current are modeled by a variable conductance: $g_T m_\infty h$. m_∞ is the activation variable given by

$$m_\infty = \begin{cases} 1 & : \quad V \geqslant V_h \\ 0 & : \quad V < V_h \end{cases} \tag{3.4}$$

When the potential is above V_h the current is activated, i.e. m_∞ is equal to one. The inactivation and the de-inactivation status is given by the h-variable, and its dynamics is modeled by

$$\frac{dh}{dt} = \begin{cases} -\frac{h}{\tau_h^-} & : \quad V \geqslant V_h \\ \frac{(1-h)}{\tau_h^+} & : \quad V < V_h \end{cases} \tag{3.5}$$

When the potential is above V_h, h is falling toward 0, becoming inactivated. When the potential is below V_h, h is rising toward 1, becoming de-inactivated. Typical values for the time constants for these two activities are $\tau_h^- = 20$ ms for the inactivation and $\tau_h^+ = 100$ ms for the de-inactivation. When the T-current becomes activated, the conductance of this current becomes non-zero. The reversal potential of the current is large, $V_T = 120$ mV, leading to a strong depolarizing current. i_{ext} are external currents coming into the LIFB neuron, and could be synaptic currents, see below, or forced AC or DC currents. The parameters of the LIFB model are taken from Smith et al. (2000) and are shown in table 3.1.

Parameter	Value	Unit	Parameter	Value	Unit
V_{th}	-35	mV	C	2	μF/cm^2
V_{res}	-50	mV	g_L	0.035	mS/cm^2
V_h	-60	mV	g_T	0.07	mS/cm^2
V_L	-65	mV	τ_h^+	100	ms
V_T	120	mV	τ_h^-	20	ms

Table 3.1: *The table shows the standard parameters used in the LIFB model, from Smith et al. (2000).*

Figure 3.2 shows the sub-threshold dynamics of the LIFB neuron in three different situations of external current injection. The simulation resemble the behavior measured in real relay cells shown in figure 2.3. All three simulations

are created by first injecting a current keeping the potential at: -55mV, -60mV, and -70mV, respectively. Then a depolarizing current of i_{ext}=-0.9 mA/cm^2 is added. Depending on what *mode* the LIFB neuron is in, it responds differently. In the top plot the neuron responds in a linear tonic mode to the current injection. In the middle plot, the current does not manage to make the neuron fire, i.e., could not raise the potential above -35mV, and the neuron does not fire at all. In the bottom plot the neuron responds with a burst. Here the T-current is de-inactivated by the low membrane potential, below -60mV, and the injected current manages to get the membrane potential above the low threshold of the T-current activating it, leading to the transient burst.

3.3 Models of synapses

The model used for the synapses in the feedforward circuit of LGN are based on the model presented in Destexhe et al. (1998) and are all conductances based. Here models of $GABA_B$, $GABA_A$, AMPA, and NMDA receptors are presented. We are only going to develop the three first models. There is no model developed for the metabotropical mGluR5 receptor, but we suggest using the same basic model as for the $GABA_B$, with some difference in parameters. The argument for doing this is that the $GABA_B$ model, catches salient features of the dynamics of a general metabotropic receptor, and that the $GABA_B$ model has also been used to model mGluR1 receptors in the LGN by Emri et al. (2003). We are going to simplify the models presented in Destexhe et al. (1998) so they fit the effective integration method of exact integration, for more information of this method see section A.2 in the appendix.

3.3.1 Conductance in ionotropic synapses

A ionotropic synapse is fast and the contribution to the total synaptic conductances from this synapse is fairly linear with the number of incoming spikes, i.e., two spikes give rise to twice the amount of conductance than one does. A spike in a presynaptic neuron triggers a fast rise in the concentration of transmitter substance in the synaptic cleft, which is registered by receptors in the postsynaptic membrane. These are attached to ion channels, which then open when the transmitter substance arrives. When opened the conductance across the membrane, for the ion of that channel, is increased. This procedure is very fast and because of the fast unbinding of transmitter substance from the receptors, it does not last long.

Destexhe et al. (1998) present a simple model for the conductance dynamics of the ionotropic AMPA and $GABA_A$ receptors. We are going to use the data presented in that paper and the dynamics of the model to develop our own

	$\alpha\ (M^{-1}ms^{-1})$	$\beta\ (ms^{-1})$	$V_{is}\ (mV)$	$\bar{g}_{is}\ (nS)$	$[T]_{max}\ (M)$	Substance present
AMPA	1.1×10^3	0.19	0	0.35-1.0	10^{-3}	1 ms
$GABA_A$	5×10^3	0.18	-80	0.25-1.2	10^{-3}	1 ms

Table 3.2: *The table shows values for the parameters in the model in eq. 3.6 for the AMPA and $GABA_A$ ionotropic synapses.*

model. Their generic model for a ionotropic synapse looks like:

$$\begin{aligned} \frac{dr}{dt} &= \alpha\,[T]\,(1-r) - \beta r \\ i_{is} &= \bar{g}_{is}\,r\,(V - V_{is})\ . \end{aligned} \tag{3.6}$$

Here r is the fraction of activated ion channels, i.e., channels with transmitter substance attached to them. $[T]$ is the concentration of transmitter substance in the cleft. It is $[T]_{max}$ 1 ms after a presynaptic spike and zero any other time, i.e., a hat-function triggered by a presynaptic spike, with 1 ms width and $[T]_{max}$ hight. i_{is} is the synaptic current from one ionotropic synapse, the subscript *is* is short for *ionotropic synapse*. $\bar{g}_{is}$ and V_{is} are the maximum conductance and the reversal potential of the synaptic current, respectively. The latter determines whether the synapse is excitatory or inhibitory. The maximum conductance is given in absolute units[3].

Table 3.2 shows the parameters presented for the model in eq. 3.6, from Destexhe et al. (1998), and figure 3.3 shows a simulation of the dynamics of r and the following response in the membrane potential for the AMPA and $GABA_A$ synapses, given one and four presynaptic spikes. The figure also shows the concentration of the transmitter substance in the cleft, and the opening and closing of ion channels as a response to the presence and absence of this. We see that the fraction of open channels, r, saturates, during heavy presynaptic spiking. Destexhe et al. (1998) explain that r should represent a fraction, and therefore should saturate at one. From figure 3.3, especially from the AMPA synapse plot, we could suspect that it actually does not do that. If it does not reach one the maximum conductance of the synapse can neither reach $\bar{g}_{\mathrm{is}}$. The exact saturation values together with the time constance for the rise and decay part for the r variable is shown in table 3.3.

We are not going to use eq. 3.6 for the conductance in ionotropic synapses. Instead we are going to use a model which is simpler and faster to simulate but still catches salient features of this model. The model we are going to use

[3]The LIFB model in eq. 3.1 requires conductance in specific units. To use eq. 3.6 in our LIFB model, the synaptic conductances have to be divided by the area of the cell.

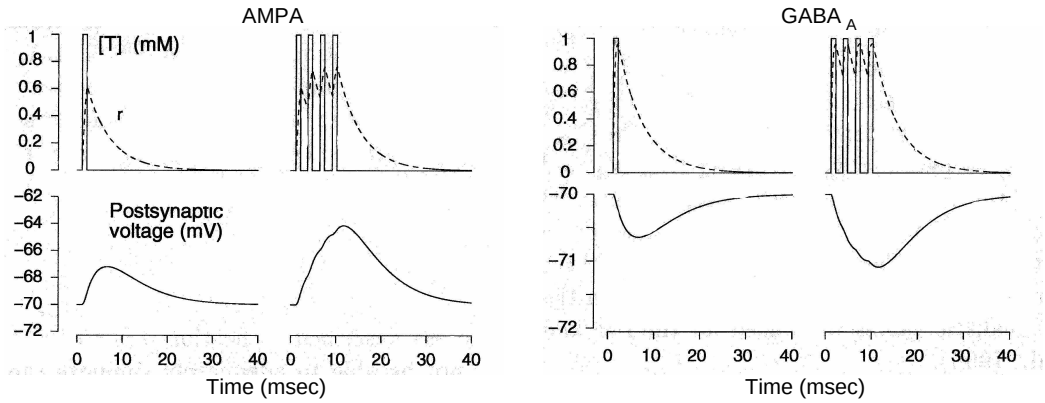

Figure 3.3: *The figure shows response of the synaptic model in eq. 3.6, for both AMPA, left panel, and $GABA_A$, right panel, to one and four presynaptic spikes respectively. The upper plots show the concentration of transmitter substance in the cleft following presynaptic spikes and the fraction of open ion channels, the variable r, in response to the present transmitter molecules. The lower plots show the postsynaptic voltage response to the different inputs. Here we see that AMPA depolarize and $GABA_A$ hyperpolarize the cell. Modified from Destexhe et al. (1998).*

is

$$i_{is} = \bar{g}_{is}(V - V_{is}) \sum_{l=1}^{N} \beta_l(t - t_l) \quad . \tag{3.7}$$

In this model the total synaptic conductance from one synapse is modeled by a sum of *beta-functions*, or difference of exponentials, which each models the dynamics in the postsynaptic conductance given one presynaptic spike. $\bar{g}_{is}$ is the peak conductance caused by one spike, not the maximum conductance of the whole synapse as in the model in eq. 3.6. N is the total number of spikes arriving the synapse, each causing the same postsynaptic conductance. V_{is} is the reversal potential of the corresponding ion current. $\beta_l(t - t_l)$ is the beta-function following the lth spike at time t_l. It models the dynamics of the conductance following a single spike at time t_l. The function is given by

$$\beta_l(t - t_l) = \begin{cases} 0 & : \quad t < t_l \\ C\left(e^{-\frac{t-t_l}{\tau_d}} - e^{-\frac{t-t_l}{\tau_r}}\right) & : \quad t \geqslant t_l \end{cases} , \tag{3.8}$$

where τ_r is the time constant for the rise and τ_d is the time constant for the decay of the beta-function, and $\tau_r < \tau_d$. C is a normalization constant chosen so the peak value of β_l is one, and is given by

$$C = \frac{\tau_d}{\tau_d - \tau_r}\left(\frac{\tau_r}{\tau_d}\right)^{-\frac{\tau_r}{\tau_d - \tau_r}} \quad . \tag{3.9}$$

This model neither uses the concentration of transmitter substance in the cleft, $[T]$, nor the time this substance is present there, as the model in eq. 3.6

	$\tau_r = \frac{1}{\alpha[T]_{max}+\beta}\ (ms)$	$\tau_d = \frac{1}{\beta}\ (ms)$	$r_\infty = \frac{\alpha[T]_{max}}{\alpha[T]_{max}+\beta}$
AMPA	0.78	5.3	0.85
$GABA_A$	0.19	5.6	0.96

Table 3.3: *The table shows the time constants for the rise and decay parts to the r variable from the model in eq. 3.6 combined with the values in table 3.2, and the maximum value the variable can obtain.*

does. But in Destexhe et al. (1998), only fixed values of these parameters are presented, and we do not use any variables that take advantage of them. To save parameters we have considered these as superfluous. The left panel in figure 3.4 shows an example of the beta-function in eq. 3.8. The rising part of the beta-function represents the binding of transmitter substance to the receptors, and the decaying part represents the unbinding of transmitter substance from the receptors. The right panel of the figure shows the sum of beta functions following three incoming spikes.

Fitting to data

Our model have to be fitted to measured currents. The strength of the synapse or the conductance caused by one spike, $\bar{g}_s$, is difficult to measure and is different for different occasions of a synapse. Therefore this value has to be chosen in a meaningful way for each synapse. In general a synapse with large release of transmitter substance from the presynaptic side and a large number of receptors in the postsynaptic side is stronger than a synapse with the opposite attribute. The geometry of the dentrite is also important for the strength of the synapse. If the dendrite leading to soma, is thin, the ion current has difficulties to reach soma and the strength of the synapse is weakened. The

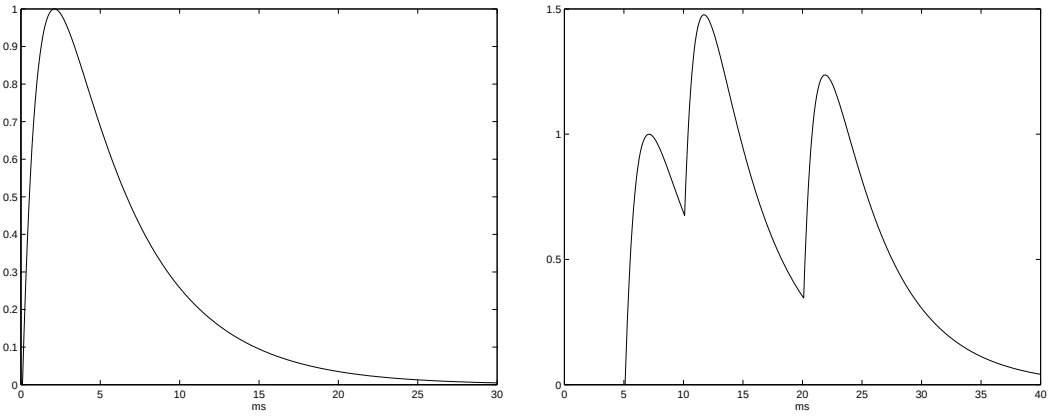

Figure 3.4: *The left panel shows the beta-function from eq. 3.8 with τ_d=5 ms, τ_r=1 ms, and t_1=0 ms. The right panel shows the sum, $\sum_{l=1}^{3} \beta(t-t_l)$, with the same parameters as in the left panel. The spike times are: t_1=5 ms, t_2=10 ms and t_3=20 ms.*

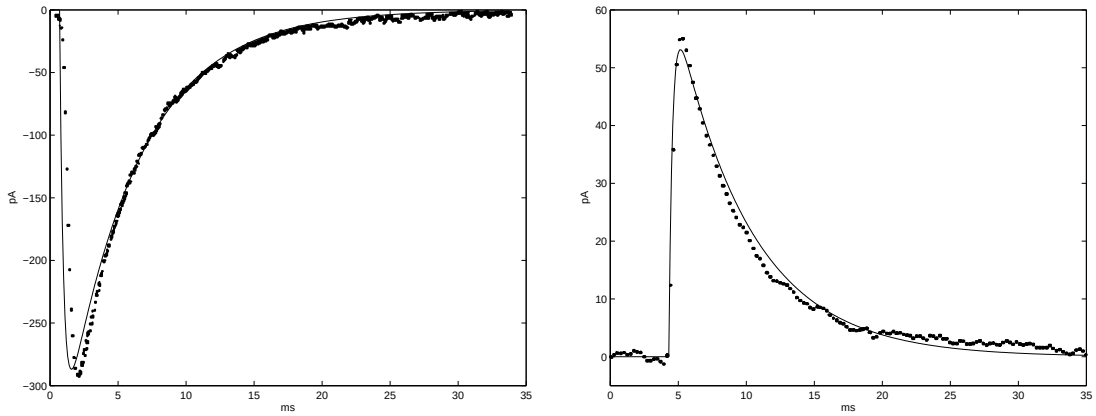

Figure 3.5: *The fitted beta-function and the synaptic currents. The left panel shows the current from the AMPA synape and the right panel shows the current from the $GABA_A$ synapse, following one presynaptic spike. The data are from figures in Destexhe et al. (1998).*

strength of the synaptic current is also weakened, if the synapse is situated far out in the dendritic arbor, thus far from the soma.

The two time parameters, τ_r and τ_d, can be fitted. We have done that for the two ionotropic synapses we use, AMPA and $GABA_A$. We have used the figures in Destexhe et al. (1998) to obtain the data. The fit was done by scanning the figures from Destexhe et al. (1998) and using the fminsearch-function in Matlab to minimize the difference of the area between the two curves. By doing this we minimize the error in the total synaptic current. Figure 3.5 show the figure that was scanned from Destexhe et al. (1998) with our fitted beta-functions. The values obtained from the fit are presented in table 3.4.

	τ_r	τ_d
AMPA	0.28 ms	5.34 ms
$GABA_A$	0.27 ms	5.5 ms

Table 3.4: *The table shows the time constants for the beta-function in the AMPA and $GABA_A$ synapses. The values were obtained by fitting the synaptic currents in eq. 3.7, with data from Destexhe et al. (1998), see also figure 3.5.*

If we examine the rise phase of the AMPA model from figure 3.5 we see that our model rises faster than the measured data. This is a pay-off for the total fit. If we had chosen an other fit criteria, for example the square of the distance between the two curves, we had obtained a better fit in the rise face but a worse fit in the total current. The model in eq. 3.6 actually fit the rise face better. See the slower time constant for the rising part of the AMPA synapse for this model in table 3.3.

3.3.2 Conductance in metabotropic synapses

A metabotropic synapse reacts slowly and non-linearly to presynaptic spikes. This is because of the complex second messenger system, and also due to the fact that the metabotropic receptors often lie at the rim of the synapse. Therefore a metabotropic synapse need strong input from a presynaptic neuron to deliver any postsynaptic current, but when this occurs, it lasts longer. These features have been incorporated in a model for the $GABA_B$ metabotropic receptor, in Destexhe et al. (1998). We are going to use and develop their simplest model for the $GABA_B$ receptor so we can use fast integration routines, lowering the numbers of parameters, but keeping the dynamics features. Their model for the $GABA_B$ synaptic current looks like

$$\begin{aligned} \frac{dr}{dt} &= K_1 [T] (1 - r) - K_2 r \\ \frac{ds}{dt} &= K_3 r - K_4 s \\ i_{\mathrm{GABA_B}} &= \bar{g}_{\mathrm{GABA_B}} \frac{s^n}{s^n + K_d} (V - V_{\mathrm{GABA_B}}) \ . \end{aligned} \tag{3.10}$$

Here r and s represent the fraction of activated receptors, receptors with bound transmitter substance, and the concentration of activated second messenger protein. $[T]$ is the concentration of transmitter substance in the cleft. Following a spike this is $[T]_{max}$ during one ms, otherwise it is zero. K_1 to K_4 are the binding and unbinding rate coefficients. The non-linearity, $\frac{s^n}{s^n+K_d}$, illustrates the dynamics of a hypotheses that it takes n second messenger proteins bound to one receptor to open a channel, see Destexhe et al. (1998) for more details. The dynamics of this non-linearity is shown in the right panel of figure 3.6. K_d is the dissociation constant of the binding of second messenger protein to the ion channels. As for the ionotropic synapse in eq. 3.6, $\bar{g}_{\mathrm{GABA_B}}$ and $V_{\mathrm{GABA_B}}$ are the maximum conductance and the reversal potential for that synapse. Table 3.5 shows the values for the parameters, from Destexhe et al.

Parameter	Value	Parameter	Value
K_1	$90\, M^{-1} ms^{-1}$	$[T]_{max}$	$10^{-3}\, M$
K_2	$1.2 \times 10^{-3}\, ms^{-1}$	Substance present	$1\, ms$
K_3	$0.18\, ms^{-1}$		
K_4	$34 \times 10^{-3}\, ms^{-1}$	$\bar{g}_{\mathrm{GABA_B}}$	$1\, nS$
K_d	$100\, M^4$	$V_{\mathrm{GABA_B}}$	$-95\, mV$
n	4 binding cites		

Table 3.5: *The table shows the parameters for the model in eq. 3.10, from Destexhe et al. (1998).*

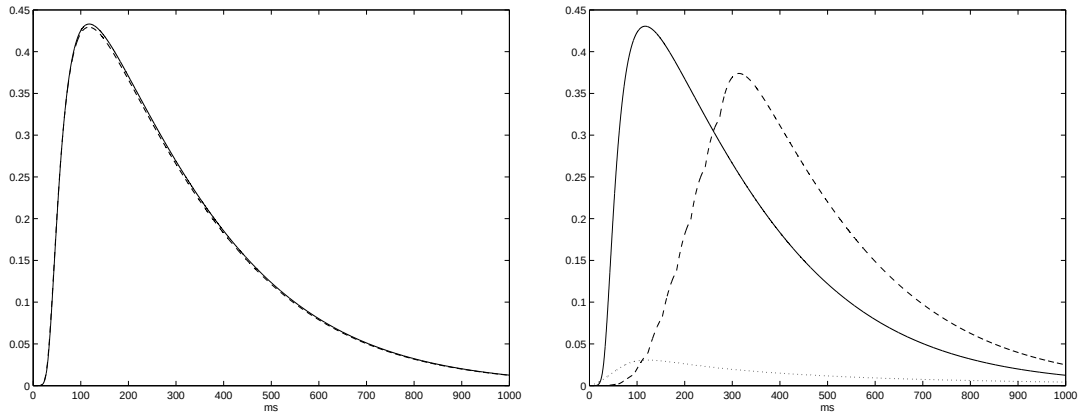

Figure 3.6: *The left panel shows a plot of* $\frac{s^n}{s^n+K_d}$ *from eq. 3.10, solid line and* $\frac{y2^n}{y2^n+K_d}$ *from eq. 3.12, dashed line. The latter fitted to the former. Both models are fed with ten spikes with an interspike interval (isi) of 3 ms.*
The right panel shows the effect of the non-linearity of $\frac{y2^n}{y2^n+K_d}$ *from eq. 3.12. This non-linearity is plotted with different input. The solid line is the response from ten spikes with isi of 3 ms in the same synapse. The dotted line is the total response from five synapses receiving two spikes each, given a total of ten spikes, with an isi of 3 ms. The dashed line is the response from one synapse receiving ten spikes but now with an isi of 30 ms. We clearly see that ten spikes through one synapse give larger respons than ten spikes through five synapses. We also see that ten spikes concentrated in time give rise to a slightly larger response than ten spike spread in time.*

(1998)

We are going to re-write the system in eq. 3.10 so it fit the integration method of exact integration. By setting $[T]$ equal to zero the dynamical system of the r and s variables, begin to resemble the system given by eq. A.17 and eq. A.25, in the appendix. If we introduce a two dimensional state vector $\mathbf{y}$, where the first and second state variables are y_1 and y_2, and setting $y_1 = K_3\, r$, $y_2 = s$, $[T] = 0$, $\tau_d = \frac{1}{K_2}$ and $\tau_r = \frac{1}{K_4}$, we can write eq. 3.10 as

$$\dot{\mathbf{y}} = A\,\mathbf{y} \ , \tag{3.11}$$

with

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \ A = \begin{bmatrix} -\frac{1}{\tau_d} & 0 \\ 1 & -\frac{1}{\tau_r} \end{bmatrix} \tag{3.12}$$

$$i_{\mathrm{GABA_B}} = \bar{g}_{\mathrm{GABA_B}} \frac{y_2{}^n}{y_2{}^n + K_d} \left(V - V_{\mathrm{GABA_B}}\right) \ .$$

The major differences between this model and the one in eq. 3.10 is the registration of a presynaptic spike. In eq. 3.10 this is done by letting the postsynaptic side being exposed to a concentration of transmitter substance,

Parameter	Value	Parameter	Value
τ_r	$29\,ms$	K_d	$100\,M^4$
τ_d	$830\,ms$	n	4 binding cites
y_{1max}	0.18	$\bar{g}_{\mathrm{GABA_B}}$	$1\,nS$
A_{add}	0.017	$V_{\mathrm{GABA_B}}$	$-95\,mV$

Table 3.6: *The table shows the parameters for the model in eq. 3.11, eq. 3.12 and eq. 5.16.*

$[T]_{max}$, during 1 ms. In our model this is done by adding a value to the first state variable in $\mathbf{y}$. This value is dependent on two values y_{1max} and A_{add}, where y_{1max} is the maximum value the first state variable of $\mathbf{y}$ could reach and A_{add} is a value that have to be fitted from the model in eq. 3.10. The procedure of registering a presynaptic spike is explained in section 5.3.

The fit was done by minimizing the area between the non-lineareties in the two models, i.e., $\frac{s^n}{s^n+K_d}$ from eq. 3.10 and $\frac{y2^n}{y2^n+K_d}$ from eq. 3.12, following an input of 10 spikes with an interspike interval of 3 ms. This was done using fminsearch-function in Matlab. The left panel in figure 3.6 shows the fit. The full parameter list for our model is presented in table 3.6, including the value for the fitted A_{add} parameter. We notice the large time constants compared to the ionotropic synapses from table 3.4, showing that this model has a much slower dynamics than a ionotropic synapse.

Metabotropic glutamate receptors
Emri et al. (2003) have used the model in eq. 3.10 to simulate the response from a mGluR1 receptor, a metabotropic receptor receiving glutamate. The only parameter they change to do this, is the reversal potential, making the synapse excitatory. Unfortunately they do not mention which value they change it to.

Chapter 4

NEST

4.1 Introduction

NEST simulator, (Diesmann and Gewaltig 2003) is the simulation program we use to simulate the networks of spiking neurons. It is also this program that the models from chapter 3 is implemented in. The NEST program is coded in C++. It is ported to a lot of different hardware systems, and in this chapter we briefly present the fundamental concepts of NEST.

4.2 Large networks and fixed time steps

The NEST technology is built to simulate large networks of neurons. The communication technique and threading technology used make it possible to simulate networks with tens and hundreds of thousands of neurons. Of course this requires that the simulations are done on large *symmetric multiprocessing* (SMP) machines where the threading technology could be used.

A single neuron in a spiking neuron network is highly non-linear and thus very unpredictable in its dynamics. Some neurons do not receive any input and could in theory be integrated with large time steps. Other neurons receive strong excitatory input, giving strong spiking output, and should in theory be integrated with very small time steps to compensate for the very discontinuous dynamics involved in receiving and sending spikes. But when we have large networks with coupled neurons, this mixture of strategies is very difficult. We do not know what input a neuron would get before we have integrated all the other neurons, and NEST therefore integrate all neurons as simultaneously as possible. This is done with all neurons being integrated with a fixed time step. This imposes a constraint on the design of the update procedure, every unit in the network in a certain time step have to be updated before any other units are updated in the next one.

4.3 Nodes, events and synapses

The main communication unit in NEST is a *node*. Only nodes have the special ability of sending and receiving *events*. For those who are familiar with object-oriented programming, the node is an abstract class which every unit that need the ability to communicate with other nodes via different events, must inherit. The most common event is the *spike event*. This is a δ-spike that is sent between neurons, which all are nodes. The spike event could also be sent to other nodes than a neuron, for example a spike detector, that registers which neuron sent the spike event and at what time. Other nodes, for example a DC or AC generator, are able to send *current events* to neurons, which are used to simulate external input currents to neuron models.

The synaptic models from section 3.3 in chapter 3 are implemented in NEST without being nodes. This means that a singel synapse can not receive any spike events. The synapses is stored in the receiving neuron and the sending neuron only deliver the spike to the receiving neuron which then pass it to the right synapse. The synapses is created and stored when a connection between two neurons is made. Before this implementation, the synapses were singel nodes sending the conductances to the receiving neuron trough the event mechanism. By storing the single synapses inside the receiving neuron, we accelerate the communications of the different conductances.

All the events must be delivered with a delay of at least one time step. Therefore a node cannot send an event in one time step to a receiving node, that have impact on this node in the very same time step. By this, causality in the network is preserved.

4.4 Interface

The language used to make simulations in the NEST environment is called *Simulation Language Interface* (SLI), a language akin to PostScript. It allows you to make simple calculations, write programs with the most common control structures and to build your own networks and simulate these.

4.5 Extendability

NEST is not a finished simulation tool, but rather offers an infrastructure for simulations of large scale networks. Anyone could use NEST to add their own models which they want to simulate in large network. In this thesis I have for example implemented the LIFB model with different integration methods and suggested a new way of implementing conductance-based synapses.

Chapter 5

Numerical implementations

5.1 Introduction

In this chapter we show how two different numerical integration methods are used to integrate the models from chapter 3. We also discuss different numerical errors that could be introduced by doing this and the treatment of these. The implementations and the error treatments are then tested in chapter 6.

The way the models are implemented are important for practical and numerical reasons. The integration methods have to be fast and effective and easy to implement and of course they have to be precise. We use two different types of numerical methods, *Runge-Kutta* (RK) methods and the method of *exact integration*. These are briefly presented in appendix A. A more thoroughly presentation of the RK methods could be accessed through the literature on numerical methods, for example Mathews (1987), and Lambert (1991). A more thorough presentation of the exact integration could be found in Rotter and Diesmann (1999).

The RK methods are powerfull and fairly robust methods of integration. We use two types of these methods, second order (RK2) and fourth order (RK4), to integrate the complex LIFB model from eq. 3.1. They are both implemented in section 5.2. The errors introduced by these methods can be formalized and used in an over all error analysis. In 5.5, we use this frame work to analyze the errors the implementation of the LIFB model, introduces.

The method of exact integration does not introduce any additional errors than the rounding error, limited by the machine. The prerequisite for a model to be integrated with exact integration is that it is linear, time invariant and continuous in all its derivative. All the model we are going integrate with exact integration have continuous derivative of all orders, and we therefore omit the last demand when this method is discussed later. We use this integration method to integrate three variables, used by the RK algorithm. The first

variable is all the synaptic conductances. As explained in section 3.3, we have chosen to model the conductances by beta-functions. This function is a solution of a two dimensional, linear and time invariant system of differential equation, see section A.2 in the appendix, and can therefore be integrated by exact integration. In section 5.3 we present a simplified implementation of the use of exact integration when the synaptic conductances is integrated. In section 5.5 we extend this version. The second variable integrated by exact integration is the h-variable in the T-current from eq. 3.3. As long as the membrane potential is either above or below the T-current threshold, V_h, the rise and decay of the h variable is expressed by a linear and time invariant differential equation. The third variable is the external sinusoidal current, used to drive the test network, see chapter 6. For more details about the implementation of this, see Rotter and Diesmann (1999).

5.2 LIFB model

The RK methods we are going to use to integrate the LIFB model is well known and straight forward to implement. These methods also give us a formal framework for an error analysis, see section 5.5 below.

The LIFB model in equation 3.1 is linear in V and the notation could therefore be simplified. We are going to use a notation similar to the one introduced by Shelley and Tao (2001) for the implementation of the RK method.
It is useful to rewrite eq. 3.1 as

$$\frac{dV}{dt} = f(t, V) = a(t)V + b(t) \ , \tag{5.1}$$

where $a(t)$ is the sum of the different conductances, divided by the membrane capacitance and is given by,

$$a(t) = \frac{1}{c_m}\left(g_{_L} + g_{_T} m_\infty h(t) + \sum_k g^k_{syn}(t)\right), \tag{5.2}$$

where $\sum_k g^k_{syn}(t)$ is the total sum of the different synaptic conductances. $b(t)$ is the sum of the external input currents and the different conductances multiplied by the corresponding reversal potential, and divided by the membrane capacitance and is given by

$$b(t) = \frac{1}{c_m}\left(g_{_L}\, V_{_L} + g_{_T} m_\infty h(t)\, V_{_T} + \sum_k g^k_{syn}(t)\, V^k + i_{ext}\right), \tag{5.3}$$

where $\sum_k g^k_{syn}(t)\, V^k$ is the sum of the different synaptic conductances multiplied by the corresponding reversal potentials, and i_{ext} is the sum of external

currents. It is important to notice that these two formulas are only valid in time steps the V_h threshold is not passed. If the potential is above this threshold, m_∞ is equal to one and if the potential is below it is equal to zero. If the V_h threshold is passed in the time step, a discontinuity occur. This is dealt with in section 5.5.

By setting the potentials in the beginning and end of the nth time step to V_n and V_{n+1}, using the notation from eq. 5.1, and letting Δt be the size of the fixed time step, we can express the next potential from the former by the two RK algorithms. This is done by

$$\begin{aligned} V_{n+1} &= V_n + \frac{\Delta t}{2}(k_1 + k_2) \\ k_1 &= f(t_n,\, V_n) = a_0\, V_n + b_0 \\ k_2 &= f(t_n{+}\Delta t,\, \Delta t\, V_n{+}k_1) \\ &= a_1(V_n + \Delta t(a_0 V_n{+}b_0)) + b_1 \ , \end{aligned} \tag{5.4}$$

where $a_0 = a(t_n)$, $a_1 = a(t_n{+}\Delta t)$ and $b_0 = b(t_n)$, $b_1 = b(t_n{+}\Delta t)$, with the RK2 algorithm and by

$$\begin{aligned} V_{n+1} &= V_n + \frac{\Delta t}{6}(k_1 + 2k_3 + 2k_3 + k_4) \\ k_1 &= f(t_n,\, V_n) = a_0\, V_n + b_0 \\ k_2 &= f\left(t_n{+}\frac{\Delta t}{2},\, V_n{+}\frac{\Delta t}{2}k_1\right) \\ &= a_{1/2}{\times}\left(V_n{+}\frac{\Delta t}{2}k_1\right) + b_{1/2} \\ k_3 &= f\left(t_n{+}\frac{\Delta t}{2},\, V_n{+}\frac{\Delta t}{2}k_2\right) \\ &= a_{1/2}{\times}\left(V_n{+}\frac{\Delta t}{2}k_2\right) + b_{1/2} \\ k_4 &= f(t_n{+}\Delta t,\, V_n{+}\Delta t k_2) \\ &= a_1(V_n{+}\Delta t k_2) + b_1 \ , \end{aligned} \tag{5.5}$$

where a_0, a_1, b_0, b_1 are the same as for RK2 and $a_{1/2} = a(t_n{+}\Delta t/2)$ and $b_{1/2} = b(t_n{+}\Delta t/2)$, with the RK4 algorithm. The errors introduced when the LIFB model is integrated by these RK schemes, are all addressed in section 5.5. Because the synaptic conductance and the h variable are included in a_1, b_1, $a_{1/2}$ and $b_{1/2}$, they have to be calculated before the algorithm in eq. 5.4 and eq. 5.5 are used to calculate the next potential.

By using the time dependent a and b parameters these methods become very effective. During an interval, not containing any discontinuities, we do

not have to evaluate a_0 or b_0 since $a_0|_{t_n+\Delta t} = a_1|_{t_n}$ and $b_0|_{t_n+\Delta t} = b_1|_{t_n}$. An other important feature of both the RK2 and the RK4 schemes is that V_{n+1} is linear in V_n, so the next potential can be written as

$$V_{n+1} = A\,V_n + B \ , \tag{5.6}$$

where A and B is the sum of all the parameters. These are used when errors introduced by discontinuities are treated.

5.3 Synaptic input

The dynamics of the synaptic conductances presented in section 3.3 are all based on the beta-function. This function is the solution of a two dimensional linear and time invariant system of differential equations, and can therefore be integrated by the method of exact integration. This is described in section A.2 in the appendix, and for more details see Rotter and Diesmann (1999). When the beta-function is integrated by exact integration instead of being evaluated as the difference of two exponentials, it introduces some very nice features.

(i) The simulation time for the beta-function is accelerated.

(ii) The registration of a presynaptic spike is much faster.

(iii) One state variable could hold many *lumped* conductances of the same sort.

In access to these, the method of exact integration let us, in an easy way, add a saturation feature to the postsynaptic conductance during heavy presynaptic spiking, as the model in eq. 3.6 also do, see figure 3.3. Because of the non-linearity in the model of the metabotropic synapes they can not be lumped, so the third feature does not apply for these models.

Independent of what spike-adding procedure we choose, see below, for sake of simplicity, we move the actual arrival-time of the lth spike to the time grid in the beginning of the nth time step it arrives the synapse, setting $t_l^{'=t_n}$. This make the registration of the presynaptic spikes much easier but it introduces an error, we are going to address in section 5.5.

Accelerated simulation

If we use eq. 3.8 we have to evaluate two exponentials for every spike that is registered in every time step. This is a very time consuming and in-efficient procedure. By working with a fixed time grid we could instead use exact integration. The beta-function, as described in section A.2 in the appendix, is

a solution to a two dimensional, linear and time invariant system of differential equations. This system is given by

$$\dot{\mathbf{y}} = A\,\mathbf{y} \ , \tag{5.7}$$

where

$$\mathbf{y} = \begin{bmatrix} \frac{1}{\tau_r}\beta + \dot{\beta} \\ \beta \end{bmatrix}, \ A = \begin{bmatrix} -\frac{1}{\tau_d} & 0 \\ 1 & -\frac{1}{\tau_r} \end{bmatrix}. \tag{5.8}$$

β is the value of the beta-function. We notice that the second state variable of $\mathbf{y}(t_n)$ contains the value of the this function, at time t_n. The state vector at time t_{n+1}, $\mathbf{y}(t_{n+1})$, can then be expressed with the former value, $\mathbf{y}(t_n)$, with a simple matrix calculation

$$\mathbf{y}(t_{n+1}) = e^{A\Delta t}\mathbf{y}(t_n) \ , \tag{5.9}$$

where $e^{A\Delta t}$ is the time evolution operator for the beta-function given by the two dimensional matrix of

$$e^{A\Delta t} = \begin{bmatrix} e^{-\frac{\Delta t}{\tau_d}} & 0 \\ \frac{\tau_r\,\tau_d}{\tau_d - \tau_r}\left(e^{-\frac{\Delta t}{\tau_d}} - e^{-\frac{\Delta t}{\tau_r}}\right) & e^{-\frac{\Delta t}{\tau_r}} \end{bmatrix}. \tag{5.10}$$

Instead of evaluating two exponentials every time step, we just make a single matrix multiplication, i.e., eq. 5.9.

The dynamic of the metabotropic synapses is already described by a two dimensional system of linear and time invariant differential equations. If we compare eq. 3.11 and eq. 3.12 with eq. 5.7 and eq. 5.8, we see that they are describing the same type of dynamics. Therefore we can integrate the former with exact integration too, and benefit from the features given by this method. We use the same equations as for the ionotropic synapse to update the state variable of a metabotropic synapse, i.e., eq. 5.9 and eq. 5.10.

Registration of spikes

We have to deal with the registration of presynaptic spikes differently when using the models for ionotropic and metabotropic synapses. First we examine the procedure for ionotropic synapses.

The registration of the lth spike arriving a ionotropic synapse at time t_l, is done by adding a beta-function with time argument zero, i.e. $t - t_l$, to the variable keeping the conductances, i.e. the second state variable of $\mathbf{y}$. As described in section A.2 in the appendix, a beta function is initialized by

adding the value of the derivative of the beta-function when the time argument is zero,

$$\dot{\beta}_0 = C\left(\frac{1}{\tau_r} - \frac{1}{\tau_d}\right) = \frac{1}{\tau_r}\left(\frac{\tau_r}{\tau_d}\right)^{-\frac{\tau_r}{\tau_d - \tau_r}} \quad , \tag{5.11}$$

to the first state variable of $\mathbf{y}$. C is the normalization constant given by eq. 3.9. Because the first state variable of $\mathbf{y}$ only holds the value of the sum of beta-functions, corresponding to the fixed time grid, we move the actual spike arrival time from t_l to the fixed time grid in the beginning of the nth time step the spike arrives, setting $t_n = t_l^{'}$, where $t_l^{'}$ is the new arrival time.

One beta-function is added to the conductance, for every spike that arrives the synapse. Due to the linearity of the matrix multiplication we do not have to keep one vector for each beta-function. The total value of all beta-functions in the sum from eq. 3.7 could be gathered in the same state vector $\mathbf{y}$. In the nth time step is this illustrated by

$$\sum_{l=1}^{N} \beta_l(t_n - t_l^{'}) \Longrightarrow A\mathbf{y_{1,n}} + A\mathbf{y_{2,n}} + \cdots + A\mathbf{y_{N,n}} = A\sum_{l=1}^{N}\mathbf{y}_{l,n} = A\,\mathbf{y}_n \quad . \tag{5.12}$$

Here the β_l is the lth beta-function caused by the lth presynaptic spike arriving the synapse at $t_l^{'}$. N is the total number of spikes. $A\,\mathbf{y}_{l,n}$ holds the value of the lth beta-function when exact integration is used.

The calculation could be simplified further by including the conductance caused by one spike, $\bar{g}_{is}$ from eq. 3.7, in the calculation of the beta-function. This is done by adding

$$y_{1add} = \bar{g}_{is}\,\dot{\beta}_0 \quad , \tag{5.13}$$

instead of $\dot{\beta}_0$ when a spike is registered. The beta-function now has $\bar{g}_{is}$ as peak value.

A presynaptic spike arriving a metabotropic synapse is registered with a different procedure. We see from eq. 3.10 that, independent of how long the postsynaptic side is exposed with transmitter substance, r will saturate at

$$r_\infty = \frac{K_1[T]}{K_1[T] + K_2} \quad , \tag{5.14}$$

imposing a maximum value for our y_1 variable of

$$y_{1max} = K_3\,r_\infty = K_3\,\frac{K_1[T]}{K_1[T] + K_2} \quad . \tag{5.15}$$

We also see from eq. 3.10 that the r variable rise with a rate that is dependent of it self. If r already is large then it will rise toward r_∞ slower than it would if

it was zero. We have implemented this in a phenomenological way by adding

$$y_{1add} = \left(1 - \frac{y_1(t_l)}{y_{1max}}\right) A_{add} \quad , \tag{5.16}$$

to the first state variable in $\mathbf{y}$ every time a presynaptic spike is registered. Here A_{add} is a value that determine how much is added to y_1 when a spike arrive. This value is chosen so our model in eq. 3.12 fits the model in eq. 3.10, see section 3.3 and figure 3.6. The fitted value of A_{add} and the value of y_{1max} is presented together with the other parameters in the model from eq. 3.12 in table 3.6.

To simplify the registration of a presynaptic spike, we moved the time the spike arrive the synapse, t_l to the fixed time grid in the beginning of the time step the spike arrive, $t_l^{'}$. As we shall see in section 5.5, this introduce an error to the conductance and we therefore extend the spike registration procedure in the same section.

Lumped conductances

If a neuron receives synaptic input from M ionotropic synapses, each described by eq. 3.7, with the same time constants and reversal potential, the total synaptic current from those synapses can be expressed as a sum over the M synapses:

$$i_{tot}(t) = (V - V_{is}) \sum_{k=1}^{M} \bar{g}_{is}^k \, {y_2}^k \quad . \tag{5.17}$$

where i_{tot} is the total lumped current and ${y_2}^k$ is the second state variable of $\mathbf{y}^k$, which keeps track of the kth synapse's sum of beta-functions. $\bar{g}_{is}^k$ is the conductance caused by one spike from the kth neuron. The total conductance could then be expressed by

$$g_{tot}(t) = \sum_{k=1}^{M} \bar{g}_{is}^k \, {y_2}^k \quad . \tag{5.18}$$

Because we use the same system matrix for all the beta-functions, the same linearity argument from eq. 5.12 holds. Instead of having M vectors keeping track of the different sums of beta-functions, we could, as with the registration of additional spikes, lump all the different vectors together into one vector. The only difference is that now they represent different synapses and can therefore have different $\bar{g}_{is}$. But this have only implications for the amount added to the first state variable, when a spike is registered. A spike from the kth neuron is now registered by adding

$${y_1}_{add}^k = \bar{g}_{is}^k \, \dot{\beta}_0 \tag{5.19}$$

to the first state variable of the lumped conductances.

Because of the non-linearity of $\frac{y2^n}{y2^n+K_d}$ from eq. 3.12, this could not be done with metabotropic synapses. For these synapses we have

$$i_{tot}(t) = (V - V_{ms}) \sum_{k=1}^{M} \bar{g}_{ms}^{k} \frac{y2_k^n}{y2_k^n + K_d} \ , \tag{5.20}$$

where $y2_k^n$ is variable keeping track of the dynamics of the conductance to the kth metabotropic synapse. We see that we could not express these variables as a singel sum, which is a prerequisite to lump them together.

Saturating conductances

We now have a effective way of implementing the synaptic conductance for the two ionotropic synapses and a full implementation of the metabotropic ones. The only feature from the model in equation 3.6 we have not included so far is conductance saturation during heavy presynaptic firing. This could be done in a phenomenological way by not adding a constant value to the first state variable when a spike is registered, but a value that depends on the present value of this variable.

If a spike arrived at time $t_l^{'}$ to a ionotropic synapse and $y1(t_l^{'})$ is the value of the first state variable in $\mathbf{y}$ at that time, a value given by

$$\tilde{y}1_{add} = \left(1 - \frac{y1(t_l^{'})}{y1_{max}}\right) y1_{add} \ , \tag{5.21}$$

is added to $y1(t_l^{'})$. Here $y1_{add}$ is the same as in 5.13 and $y1_{max}$ is the maximum value $y1$ can have. This value have to be chosen meaningful or be fitted. The conductance following a spike is modeled by a the same beta-function, e.g., the time constants is the same, as in eq. 3.7, but now with a variable maximum conductance.

We have chosen $y1_{max}$ for the AMPA and $GABA_A$ synapses, such that they fit the models for the same synapses from Destexhe et al. (1998). To do this we chose the $\bar{g}_{is}$ parameter in eq. 3.7 such that the conductance caused by the first spike, peak at the same hight as the conductance from eq. 3.6 with $\bar{g}_{is} = 1$, does. By setting $\bar{g}_{is}$ in eq. 3.6 equal to one, we actually compare the conductance from eq. 3.7 with the r variable in eq. 3.6. In table 5.1 the chosen parameters are shown and in figure 5.1 the fits are plotted. In the same figure the conductances for the corresponding non-saturating synapses are also shown.

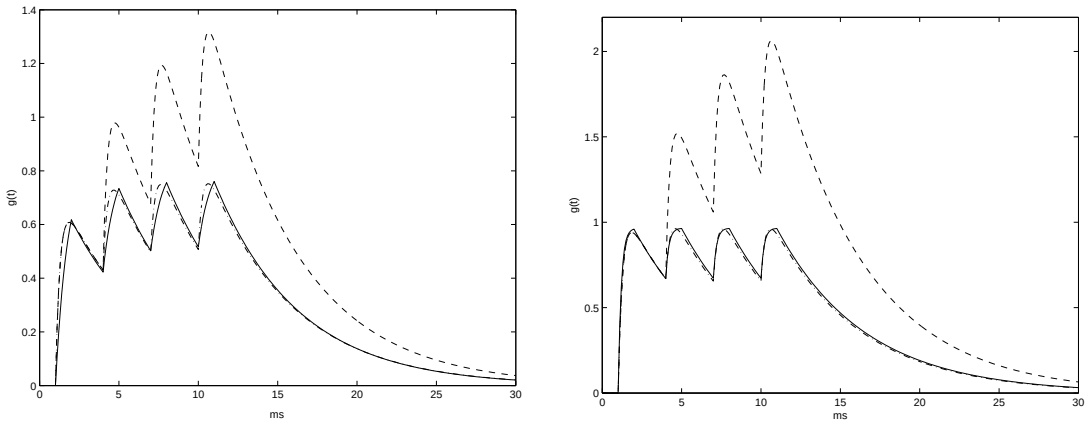

Figure 5.1: *The conductances saturate during heavy input. The left panel shows the conductance from the AMPA models and the right panel the conductance from the $GABA_A$ models. All synaptic models receive spikes at: 1, 4, 7, 10 ms. The solid line shows the conductance from eq. 3.6. Because $\bar{g}_{is}$ is set to one for this model, we actually plot the r variable. The dash-dotted line shows the conductance from the beta-function with saturation ability. The dashed line shows the original beta-function with no saturation ability. The parameters used for the plots are shown in table 5.1 and 3.4 for the conductances with the beta-functions, and in table 3.2 for the r variable, with $\bar{g}_{is}$ equal to one.*

It is not straightforward to implement conductance saturation when operating with lumped conductance as in eq. 5.18. Now the $\mathbf{y}$ vector contains the lumped sum of all the M state vectors, and the value in eq. 5.21 is dependent on only one state variable. So we cannot use this equation to add a value to y_1. If we introduce two vectors of length M, one which keeps track of the last spike from the kth synapse, $\mathbf{sp}$, and one which keep track of the value of the first state variable for the same synapse after the time for the last spike, $\mathbf{y_1}$, we could by pass this obstacle.

If we assume that the kth synapse has received a spike at time t_l^k this value is stored in sp^k. The value of the first state variable of $\mathbf{y}$ for the kth synapse at that time is stored in ${y_1}^k$. This value decays exponentially, with decay constant τ_r, see eq. 5.8. The next time the same synapse receives a spike, at time t_{l+1}^k, the value has decayed to

$$\tilde{y_1}^k = {y_1}^k e^{-\frac{t_{l+1}^k - sp^k}{\tau_r}} \quad . \tag{5.22}$$

We can now use this value in eq. 5.21 to get the $\tilde{y_1}_{add}^k$ value

$$\tilde{y_1}_{add}^k = \left(1 - \frac{\tilde{y_1}^k}{{y_1}_{max}^k}\right) {y_1}_{add}^k \quad , \tag{5.23}$$

where ${y_1}_{max}^k$ is the maximum value of ${y_1}^k$. $\tilde{y_1}_{add}^k$ is then added to the first state variable of $\mathbf{y}$ as a normal registration of a spike. $\tilde{y_1}_{add}^k + \tilde{y_1}^k$ now form the kth value of $\mathbf{y_1}$ and are therefore now stored in ${y_1}^k$. This is illustrated in figure

	$\bar{g}_{is}$	y_{1max}
AMPA	0.61	3.5
$GABA_A$	0.94	3.9

Table 5.1: *The table shows the chosen values for the synaptic model such that, during heavy firing, the conductance saturate at the same level as $\bar{g}_{is}\, r$, where $\bar{g}_{is}$ is set to one, from eq. 3.6 does. The conductance is plotted in figure 5.1 together with the r variable.*

5.2. Here is the first state variable of the kth synapse shown. The synapse receives its lth spike at $t_l = 15\,\text{ms}$ and its $(l+1)$st spike at $t_l = 20\,\text{ms}$.

Using RK4

If we use RK4, because of the $a_{1/2}$ and $b_{1/2}$ parameters, we have to evaluate the conductance in the middle of the time step too. This is done by using $\Delta t/2$ instead of Δt when the time evolution operator in eq. 5.10 is calculated, and then evaluate it twice every time step.

5.4 h variable

The h variable is integrated by the method of exact integration. This is done differently depending on whether the membrane potential at the beginning of the time step is above or below V_h. This is done by exact integration, which is explained in section A.2 in the appendix, by

$$h_{n+1} = e^{-\frac{\Delta t}{\tau_h^-}} h_n \quad , \tag{5.24}$$

in a time step where $V_n \geq V_h$ and by

$$h_{n+1} = 1 - e^{-\frac{\Delta t}{\tau_h^+}} (1 - h_n) \quad , \tag{5.25}$$

in a time step where $V_n < V_h$. Here the exponential is the time evolution operator, evaluated only once. If we use RK4 and the potential is above V_h, T-current is on, because of the $a_{1/2}$ and $b_{1/2}$ parameters, we have to evaluate the h variable in the middle of the time step too. This is done by using $\Delta t/2$ instead of Δt when the time evolution operator in eq. 5.24 is calculated and then evaluate it twice every time step. When the potential is below V_h, we just use eq. 5.25 as it is, because the T-current is inactivated and the h variable is not included in neither of $a_{1/2}$ or $b_{1/2}$.

When the membrane potential crosses the threshold for the T-current, the dynamics of the h variable is switched from one of eq. 5.24 and eq. 5.25 to the

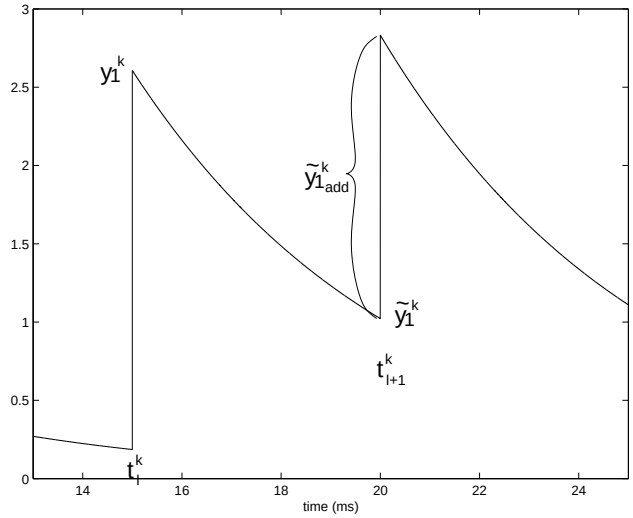

Figure 5.2: *The figure shows the first state variable of the kth synapse. The synapse receives its lth spike at $t_l = 15\,ms$ and its $(l+1)$st spike at $t_l = 20\,ms$. We see that the value at 15 ms is the value stored in ${y_1}^k$. This value decays to ${\tilde{y}_1}^k$ for the time of the next spike and is then used in eq. 5.23 to find $\tilde{y}_1{}^k_{add}$. This value is then used to register a spike by being added to the first state variable of* $\mathbf{y}$*. It is also added to ${\tilde{y}_1}^k$ to form the new value of ${y_1}^k$.*

other. This crossing introduce an error to the h variable and it is dealt with at the same time as the discontinuity of the onset of the T-current is dealt with, see the following section.

5.5 Error analysis

All the error analysis is done with the total error in the membrane potential in mind. An error in, let say the integration of the h variable could be acceptable if it causes an error of the same or higher order in Δt, than the error introduced by the RK method used to integrate the potential.

The potential is integrated by either the RK2 or RK4 method. As described in section A.2 in the appendix, there are two ways of discussing the error introduced by these methods, *i)* the *local truncation error, (LTE),* and *ii)* the *global truncation error, (GTE)*, where the latter is the consequence of the former. The LTE is the error the integration algorithm, used every time step, causes. These algorithms are eq. 5.4 for RK2 and eq. 5.5 for RK4. The LTE for these are of order $\mathcal{O}(\Delta t^3)$ and $\mathcal{O}(\Delta t^5)$ respectively, and is introduced every time step. When the RK algorithms are integrated over several time steps, the LTE accumulates to the GTE and is of one order less than the LTE, i.e., $\mathcal{O}(\Delta t^2)$ for RK2 and $\mathcal{O}(\Delta t^4)$ for RK4. The errors caused by the implementation of our models have to be of higher or equal order than these errors to the corresponding integration method. If they are of lesser order we have to

deal with them. Which truncation error should we then use to compare any introduced error with?

To answer this question we have to understand what limit these errors impose to the precision of the potential. We illustrate this by integrating the LIFB model over a period of T ms, and examine the potential at a time t_i, where $0 \leq t_i < T$. If no other error than the LTE is introduced up to that time, the total error to the potential is the GTE. If we use the RK algorithm to integrate the next potential at V_{i+1}, a new LTE error is introduced, but the total error in V_{i+1} is still of the same order as the GTE.

If we, in the same time step, introduce an additional error, let say of the same order as the LTE, this is added to the potential at V_{i+1}. Because the potential is limited by the GTE, which is of one order lesser than the error, and the order of the total error is not changed, i.e.

$$\text{TotalError} = \text{error} + \text{GTE} = \mathcal{O}(\Delta t^N + 1) + \mathcal{O}(\Delta t^N) = \mathcal{O}(\Delta t^N) \ . \tag{5.26}$$

If the error is of the same order as the GTE, we have

$$\text{TotalError} = \text{error} + \text{GTE} = \mathcal{O}(\Delta t^N) + \mathcal{O}(\Delta t^N) = \mathcal{O}(\Delta t^N) \ , \tag{5.27}$$

and also now the total error follows the same order. Of course if the introduced error is of one or more order less than the GTE, the order in the total error is lowered, for example

$$\text{TotalError} = \text{error} + \text{GTE} = \mathcal{O}(\Delta t^{N-1}) + \mathcal{O}(\Delta t^N) = \mathcal{O}(\Delta t^{N-1}) \ . \tag{5.28}$$

The discussed errors are introduced in the present time step, i.e. $[t_i,\ t_{i+1}]$, and therefore act locally. We therefore compare these with the truncation error that limits the potential in the end of the time step, i.e., the GTE. If the error is introduced not only in the present time step, but for all the remaining too, we have another situation. Now these errors accumulate. If we should compare it with the GTE, we have to keep in mind that it falls one order due to the accumulation. Because the LTE also accumulate we could compare it directly with this.

The different errors we are going to address is summed up by the following lists.

(i) Registration of a presynaptic spike, at the fixed time grid in the beginning of a time step.

(ii) Onset of an external current.

(iii) Reset of the potential after a spike.

(iv) Onset and offset of T-current.

(v) Shifting between rise and decay for the h-variable.

(vi) Onset of postsynaptic conductance.

The first error is an implementation error, and the five remaining is caused by the introduction of a discontinuity, to the first or second derivative of the membrane potential inside a time step.

As described in section A.1 in the appendix, does a discontinuity in the kth derivative introduce a local error, of order $\mathcal{O}(\Delta t^k)$, to the potential in the same time step the discontinuity occur. If the LIFB model in eq. 3.1, which is the first derivative of $V(t)$, is discontinuous an error of order $O(\Delta t)$ is introduced. The second to fourth errors is introduced in the first derivative of the potential and the last two errors is introduced in the second derivative. We shall have in mind that the errors introduced by these discontinuities is local errors and only effects the potential in the end of the time step the discontinuity occur. Because the errors only occur locally, are these errors compared with the GTE of the potential.

The presentations of the errors and how these are dealt with, follows in the following subsections. The third and fourth errors are similar and are therefore treated within the same subsection. The fifth error is closely related to the onset and offset of the T-current and are therefore also treated in that subsection.

5.5.1 Registration of a presynaptic spike

In setion 5.3, due to simplicity, we moved the spike time from the actual arrival time, t_l, to the fixed time grid in the beginning of a time step $t_l^{'} = t_n$, when these are registered. If we do this we introduce an error in the spike time of

$$t_l - t_l^{'} = A\,\Delta t \ , \tag{5.29}$$

where $0 < A < 1$. To investigate what error we introduce in the conductance we look at a time step t_i, further on in time, $t_i > t_l^{'}$. We make a Taylor expansion of the conductance, from that single spike at, $g(t_i - t_l^{'})$, round the actual time argument, $t_i - t_l$, of the conductance. We have

$$\begin{aligned} g(t_i - t_l^{'}) &= g(t_i - t_l + A\,\Delta t) \\ &\simeq g(t_i - t_l) + A\,\Delta t\, g'(t_i - t_l) + \mathcal{O}(\Delta t^2) \ . \end{aligned} \tag{5.30}$$

Here $g(t_i - t_l)$ is the actual contribution of the conductance from the spike that arrived at t_l. We see that, by using the wrong spike arrival time, $t_l^{'}$, we introduce an error in the conductance of order $\mathcal{O}(\Delta t)$. This is an error introduced

to all succeeding conductances following $t_l^{'}$ as well, and we therefore compare it directly with the LTE of the RK methods. The orders of the LTE for RK2 and RK4 are both higher than the order of this error, and we therefore have to deal with it.

To handle this error we have to expand the procedure of adding spikes, introduced in section 5.3. We can no longer register a spike arriving the synapse at time t_l, by just adding, y_{1add} to y_1 at the beginning of the time step the spike arrive, but rather add it at the time the spike actually arrive. The state vector only hold values of the conductance at the fixed time grids and we therefore have to calculate the value that y_{1add} would have evolved to at the fixed time grid at the end of time step. This value is then added to the total conductance in the $\mathbf{y}$ vector.

We first resolve the conductance without the new conductance added, by making an ordinary matrix calculation, see 5.9. A spike is then registered by adding the evolved value of y_{1add},

$$\mathbf{y_{add}} = \begin{bmatrix} e^{-\frac{t_{rest}}{\tau_d}} & 0 \\ \frac{\tau_r \tau_d}{\tau_d - \tau_r}\left(e^{-\frac{t_{rest}}{\tau_d}} - e^{-\frac{t_{rest}}{\tau_r}}\right) & e^{-\frac{t_{rest}}{\tau_r}} \end{bmatrix} \begin{bmatrix} y_{1add} \\ 0 \end{bmatrix}, \tag{5.31}$$

to this value. Here we have used the time evolution operator normally used to get the next value of $\mathbf{y}$ from the former, with Δt exchanged with, $t_{rest} = t_{n+1} - t_l$. This procedure could also be used for lumping synapses, and/or saturating synapses. We just exchange y_{1add} with the respective values.

If we use RK4 we have to check if the spike has come in the first half of the time step. If it did, we set $t_{rest} = t_n + \Delta t/2 - t_l$ and add the expression in eq. 5.31 to $\mathbf{y}(t_n + \Delta t/2)$ instead of $\mathbf{y}(t_{n+1})$.

This implementation is illustrated in figure 5.3. Here we see how the potential is integrated differently with different integration methods and with different spike registration procedures. We see that the methods not using the precise spike timing integrates the potential better than the methods not using it, when compared with the canonical run.

To accelerate the calculation of the exponentials in eq. 5.31, we could use a simplified calculation. Instead of the exponential we use $e^x \simeq 1 + x$ for the RK2 and $e^x \simeq 1 + x + x^2/2 + x^3/6$ for the RK4. The error introduced to the conductance by doing this, is of order $\mathcal{O}(\Delta t^2)$ and $\mathcal{O}(\Delta t^4)$ respectively. This error is caused to all the following conductance values too, and are therefore introduced in every RK algorithm following a registration. Because of this we compare the error with the LTE. When used in the corresponding RK algorithm the error is multiplied by Δt, and become of the same order as the LTE

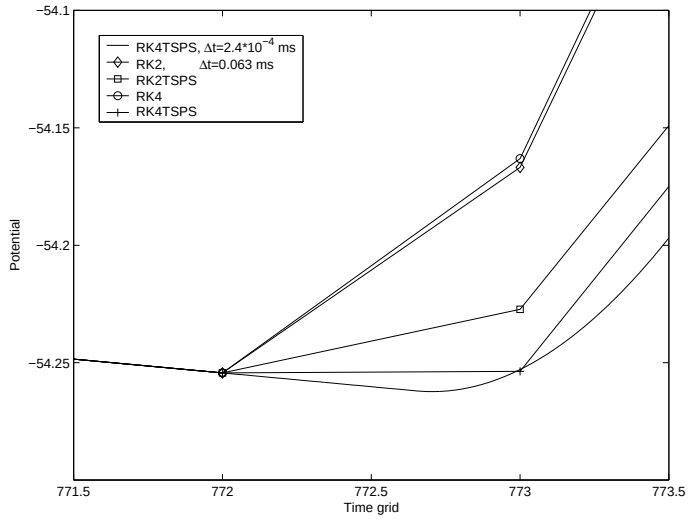

Figure 5.3: *The figure shows how the potential is integrated differently with different integration methods and with different spike registration procedures. All neurons receives a presynaptic spike in the 772:nd time step. The solid line without any symbols, is the potential in a canonical run integrated with a very small time step, $\Delta t = 2.4 \times 10^{-4}$ ms using the RK4TS integration method and with precis spike timing. The diamonds and the circles are the potentials of the RK2 and RK4 methods with out the precise spike timing. The squares and the plus are the RK2TS and RK4TS method with the precis spike timing. We see that the methods with precis spike timing integrate the potential much more precis than the methods that lack this procedure. We also see that the RK4TS with precise spike timing is the method that integrates the potential absolutely best, when compared with the canonical run.*

of the integration algorithm.

5.5.2 Onset and offset of an external current

This is the least difficult discontinuity to deal with. We only have to choose to put on the external current precisely at the fixed time grid. Then as described in secion A.1 in the appendix, no error is introduced. The discontinuity does not occur within the time step the RK algorithm is bridging when calculating the next potential. But we have to be careful when adding the current. If we turn on the current at time t_{n+1} and we are in the nth time step, we shall not include the external current in the b_1 parameter even if this represent the value of $b(t_{n+1})$. This is so, because the interval that we use to calculate the next potential, is the semi open interval $[t_n,\, t_{n+1})$, and this does not include the onset time. But when we change time step to the $(n+1)$st we have to add the external current to the b_0 parameter of that time step. For this two time steps we therefore have $b_0|_{t_n+\Delta t} \neq b_1|_{t_n}$.

5.5.3 Threshold passings

The reset of the potential and the onset of the T-current are events that occur after a threshold is passed, either V_{th} for the resetting, or V_h for the T-current. These discontinuities appear in the first derivative of $V(t)$ and therefore a local error of order $\mathcal{O}(\Delta t)$ is introduced in that time step. Unlike the onset of an external current we can not choose where these discontinuities appear and the potential will certainly not cross the two thresholds precisely at the fixed time grid every time.

Hansel et al. (1998) first introduced a linear interpolation scheme to handle the inaccuracy when resetting the potential after a spike, which could be used with a first or second order integration method. Shelley and Tao (2001) extended this interpolation scheme so it could be used in a fourth order integration method, i.e. RK4. We are going to extend their work to a generic algorithm that could handle both the threshold passings we have. We call this method *time stepping*, TS. To simplify further notation we set, $V_n = V(t_n)$ and $V_{n+1} = V(t_{n+1})$.

A threshold pass is detected by checking if V_{n+1}, which is calculated from the RK schemes, is above V_{thresh}, every time step. Here V_{thresh} is either the threshold for spike, V_{th}, or the threshold for the T-current, V_h. The time when the threshold is passed is, t_{th}, and is estimated by interpolation, see below. The method of Shelley and Tao (2001) is based on calculating two new values of the potential, $\tilde{V}_n$ and $\tilde{V}_{n+1}$, such that the interpolated potential at t_{th}, between these two values, would be the same as the potential should be right after a threshold pass, V_{thresh}^{after}. For the spike threshold this is V_{res} and for the T-current threshold it is the same as the threshold value V_h, but now with changed dynamics for the potential. This is illustrated in figure 5.4.

In the figure the solid line is the interpolated potential between V_n and V_{n+1}, and represent the potential as if the threshold was never passed. The dashed line is the interpolated potential between $\tilde{V}_n$ and $\tilde{V}_{n+1}$, and represent the potential as if the threshold was already passed, with the additional demand of $\tilde{V}(t_{th}) = V_{thresh}^{after}$. We see from panel A, that $\tilde{V}(t_{th}) = V_{res}$, and that the dynamics of $\tilde{V}(t)$ is almost the same as for $V(t)$. It is shifted downwards so it passes V_{res} at t_{th}. In panel B and C we see that $V(t_{th}) = \tilde{V}(t_{th}) = V_h$, but now the dynamics is changed. This is because the T-current is turned on in B and off in in C.

The two new potentials, $\tilde{V}_n$ and $\tilde{V}_{n+1}$, is now connected by the RK algorithm without a discontinuity occurring in the time step. We have therefore effectively by pass this cause of error. Our method is limited to the precision of t_{th}, and the method used for calculating $\tilde{V}_n$ and $\tilde{V}_{n+1}$.

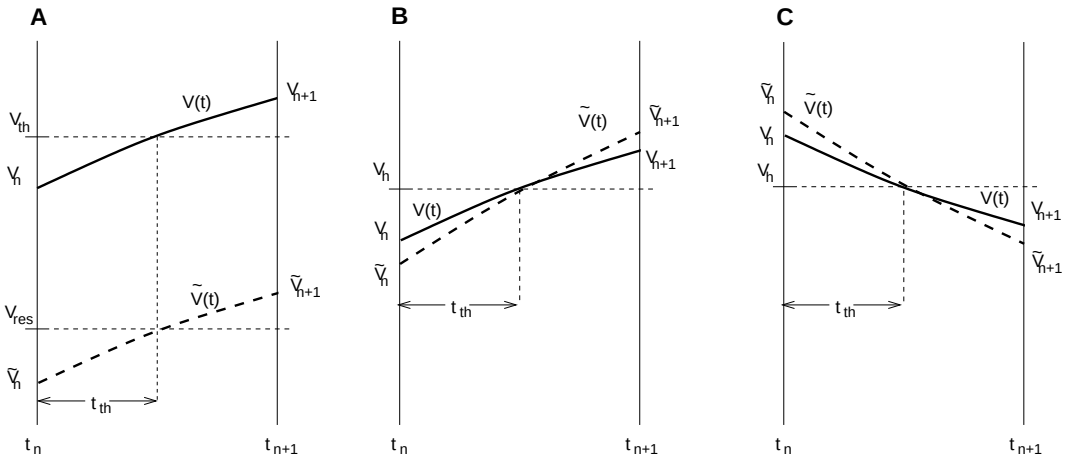

Figure 5.4: *The different panels illustrate what value the new $\tilde{V}_n$ and $\tilde{V}_{n+1}$ would be if the potential should cross V_{res} in A and V_h in B and C at the time for the threshold passing, t_{th}. B illustrates the passing from below and C from above. The solid line represents the interpolated potential between V_n and V_{n+1}, as if the threshold was never passed. The dashed line represents the interpolated potential between $\tilde{V}_n$ and $\tilde{V}_{n+1}$, as if the threshold was passed, with the additional demand that the interpolated potential $\tilde{V}(t_{th})$ equals V_{thresh}^{after}.*

Finding t_{th}
The time for the threshold passing, t_{th} is found by an interpolation polynomial for the potential $V(t)$, between t_n and t_{n+1}. When using RK2 a linear interpolation scheme is sufficient and when using RK4 a cubic interpolation scheme is necessary, see below. The linear interpolation scheme is given by

$$V_{int}(t) = V_n + \frac{V_{n+1} - V_n}{\Delta t}\, t \ . \tag{5.32}$$

To find t_{th} we have to solve this linear equation with $V_{thresh} = V_{int}(t)$, where V_{thresh} is either the threshold for spike, V_{th}, or the threshold for the T-current, V_h. t_{th} is given by

$$t_{th} = \frac{V_{thresh} - V_n}{V_{n+1} - V_n}\, \Delta t \ . \tag{5.33}$$

We see from eq. 5.33, that the error in t_{th} is of order $O(\Delta t^2)$, the same order as the GTE of RK2.

For the cubic interpolation scheme we need the derivatives of the potentials at t_n and t_{n+1} too. These are given by $\dot{v}_n = f(t_n, V_n) = a_0\, V_n + b_0$ and by $\dot{v}_{n+1} = f(t_{n+1}, V_{n+1}) = a_1\, V_{n+1} + b_1$. All the parameters are known from the

calculation of V_{n+1}. The cubic interpolation scheme is given by

$$V_{int}(t) = V_n + \dot{v}_n\, t + \left(\frac{3(V_{n+1} - V_n) - \Delta t(2\dot{v}_n + \dot{v}_{n+1})}{\Delta t^2}\right) t^2 + \left(\frac{-2(V_{n+1} - V_n) - \Delta t(\dot{v}_n + \dot{v}_{n+1})}{\Delta t^3}\right) t^3. \tag{5.34}$$

To find t_{th} we have to solve this cubic equation with $V_{thresh} = V_{int}(t)$. This could be done numerical, with for example Newton-Raphson's method. The error in t_{th} is now of order $O(\Delta t^4)$, the same order as the GTE in RK4.

Finding $\tilde{V}_{n+1}$
To find this value we are going to use two general linear expression for $\tilde{V}_n$ and $\tilde{V}_{n+1}$. The first, is found from the RK algorithm, eq. 5.6, with V_n exchanged with $\tilde{V}_n$ and V_{n+1} exchanged with $\tilde{V}_{n+1}$. We then have

$$\tilde{V}_{n+1} = A\,\tilde{V}_n + B \ . \tag{5.35}$$

The full expressions of A and B is dependent of what order of RK we use. For RK2 these are

$$\begin{aligned} A &= 1 + \frac{\Delta t}{2}\,(a_0 + a_1 + \Delta t\, a_0\, a_1) \\ B &= \frac{\Delta t}{2}\,(b_0 + b_1 + \Delta t\, b_0\, a_1) \ . \end{aligned} \tag{5.36}$$

The parameters for RK4 are presented in appendix B. We note that when the spike threshold is passed, these parameters are calculated from the already calculated a and b parameters, but when the T-current threshold is passed we have to re-calculate these parameters, now including or excluding the T-current.

To get the second linear expression that relates $\tilde{V}_n$ and $\tilde{V}_{n+1}$, we use the interpolation polynomial of the corresponding RK method to express V_{thresh}^{after}, where $V_{thresh}^{after} = V_{res}$ for the spike threshold and $V_{thresh}^{after} = V_h$ for the T-current threshold, as if the potential was passed. We use the same variable exchange as above, $\tilde{V}_n$ for V_n and $\tilde{V}_{n+1}$ for V_{n+1}. This give us a linear expression for $\tilde{V}_n$ and $\tilde{V}_{n+1}$

$$V_{thresh}^{after} = K_1\,\tilde{V}_n + K_1\,\tilde{V}_{n+1} + K_3 \ , \tag{5.37}$$

where K_1, K_2 and K_3 is coefficients, obtained from either of the two interpolation polynomials. The coefficients for the linear interpolation polynomial is straightforward, from eq. 5.32 we have

$$V_{thresh}^{after} = \left(1 - \frac{t_{th}}{\Delta t}\right)\,\tilde{V}_n + \frac{t_{th}}{\Delta t}\,\tilde{V}_{n+1} \ , \tag{5.38}$$

so,

$$\begin{aligned} K_1 &= 1 - \frac{t_{th}}{\Delta t} \\ K_2 &= \frac{t_{th}}{\Delta t} \\ K_3 &= 0 \ . \end{aligned} \tag{5.39}$$

The coefficients for the cubic interpolation polynomial are given in appendix B.

We now have two linear equations eq. 5.35 and eq. 5.37, relating $\tilde{V}_n$ and $\tilde{V}_{n+1}$ with one of them pinpointing the interpolated potential between them to V_{thresh}^{after} at t_{th}. If we solve for $\tilde{V}_{n+1}$ and get

$$\tilde{V}_{n+1} = \frac{(1-A)\, V_{thresh}^{after} + B\, K_1 - (1-A)\, K_3}{K_1 + (1-A)\, K_2} \ . \tag{5.40}$$

We have now got the new potential value after the threshold pass, without any discontinuity error introduced.

Updating the h variable
The including and excluding of the T-current in the a and b parameters are done in three stages. First we calculate the value of the h variable at the time for the threshpass, t_{th}. Second we calculate the value of the h variable as if the potential were crossed, and third we add or withdraw the value of the h variable from the a and b parameters, depending on whether the threshold was crossed from below or from above.

If the threshold, V_h, were crossed from above, turning off the T-current, we first use eq. 5.24 with Δt in the time evolution operator exchanged with $t_{th} - t_n$, to evaluate the value of the h variable at the time of the threshpass

$$h_{th} = e^{-\frac{t_{th}-t_n}{\tau_h^-}}\, h_n \tag{5.41}$$

We then use this value to obtain the value of the h variable at the fixed time grid in the end of the time step, as if the threshpass was passed. We then use eq. 5.25, but now with Δt in the time evolution operator exchange with $t_{n+1} - t_{th}$,

$$\tilde{h}_{n+1} = 1 - e^{-\frac{t_{n+1}-t_{th}}{\tau_h^+}}\, (1 - h_{th}) \ . \tag{5.42}$$

The threshold is passed and the T-current is turned off. Now we have to remove the contribution from the T-current in the a and b parameters before we evaluate the A and B parameters used to get the potential after a threshpass. Note that we have to use the old values of the h variable, i.e., h_n and h_{n+1}

when we do this, not the new $\tilde{h}_{n+1}$ value.

If the threshold, V_h, were crossed from below, turning on the T-current, we first use eq. 5.25 with Δt in the time evolution operator now exchanged with $t_{th} - t_n$, to evaluate the value of the h variable at the time of the threshpass

$$h_{th} = 1 - e^{-\frac{t_{th}-t_n}{\tau_h^+}} (1 - h_n) \ . \tag{5.43}$$

We now use this value to obtain the value of the h variable at the fixed time grid, not only in the end of the time step, but also in the beginning. We need both values when we calculate the potential after the threshpass. To get $\tilde{h}_n$ we actually reverse eq. 5.41 and get

$$\tilde{h}_n = e^{\frac{t_{th}-t_n}{\tau_h^-}} h_{th} \ . \tag{5.44}$$

This value is then used together with the ordinary time evolution operator from eq. 5.24 to obtain the $\tilde{h}_{n+1}$ value. These two values are then used to add the T-current to the a and b parameters, which are then used to get the new potential after the threshold pass.

Small error analysis
The error introduced is limited by the precision of the interpolation polynomial when the time for the threshold pass is estimated. The polynomials used introduces an error of $O(\Delta t^2)$ and $O(\Delta t^4)$ for the respective integration method. This time is then used in the corresponding interpolation polynomial in eq. 5.32 for the RK2 and eq. 5.34 for the RK4 method, to pinpoint the potential at this time to V_{thresh}^{after} and relate this potential to the $\tilde{V}_n$ and $\tilde{V}_{n+1}$ potentials. This is done with the same precision as t_{th} is estimated with. We then use the corresponding RK algorithm to relate the two potentials $\tilde{V}_n$ and $\tilde{V}_{n+1}$ and this is done with a precision of one order more than the precision got from the interpolation polynomial. $\tilde{V}_{n+1}$ is therefore limited by the precision of the interpolation polynomial that is of order $O(\Delta t^2)$ and $O(\Delta t^4)$ for the two RK methods. This is of the same order as the GTE of the corresponding method and we are safe.

The fifth error in the error analysis, concern the discontinuity when the h variable shifts between rise and decay, is now actually by-passed. In both cases, when the threshold is passed from above and below, the h variable is treated as if it has been below or above in the whole time step, when it is used in the RK algorithm, and therefore no discontinuity occur.

If the threshold for a spike is passed with the present implementation, a bug could occur. When the neuron receives heavy excitatory synaptic input

the potential after a threshold pass, $\tilde{V}_{n+1}$, could actually become greater then V_{th}, we therefore have to be moderate when the weights of the synapses are chosen. This bug could also be by-passed if we allowed the neuron to spike many times in a time step. Then we have to check if the potential after a V_{th} threshold pass again is above V_{th}. If it is that we have to use the two linear equations eq. 5.35 and eq. 5.37 to calculate $\tilde{V}_n$ too. $\tilde{V}_n$ and $\tilde{V}_{n+1}$ are then used in the same procedure as described above, first interpolating the spike time between these two values. Then using eq. 5.40 to achieve the potential after this threshold pass.

5.5.4 Incoming spikes and synaptic currents

The beta-function used in chapter 3 to model the conductance following a presynaptic spike, introduce a discontinuity error when a spike is registered. The discontinuity occur when the time argument of the beta function $(t-t_{spike})$ reaches zero, i.e. the spike is registered. Before that time the beta function is zero, and after it promptly starts to rise, introducing a discontinuity in the first derivative. This causes a discontinuity in the second order derivative in the potential and therefore also a local error of order $O(\Delta t^2)$. No matter what precision a spike is registered with, this error will occur. When using RK2 this error will be of the same order as the GTE and therefore not altering the order in Δt the total error follows. When using RK4, this error will be of two orders lower than the GTE, and therefore limiting the total errror to second order.

The time stepping algorithm from the previous subsection, could be expanded to include a way of dealing with this error too. The algorithm is quite complex and cumbersome to implement, e.g. we have to register the incoming spikes to a neuron in a chronological sequence, and then implement a way of registering one spike at a time. We have therefore not implemented the procedure, but we briefly sketch how this could be done.

The implementation is based on the same procedure used in the previous subsection, where the discontinuities of reseting the potential after a spike and the onset and offset of the T-current, were dealt with. There we first calculated the next potential V_{n+1} as if the discontinuity did not occur. We do the same for the registration of presynaptic spikes. We first calculate the next potential as if the neuron did not receive any spikes in the present time step. We then register the spikes, but now instead of registering them with the ordinary beta-function that is zero for $t - t_{spike} < 0$, we now let the beta-function rise from below zero, passing zero when $t = t_{spike}$. For this we use the same function as for $t - t_{spike} \geq 0$. This modified beta-function does not introduce any discontinuities.

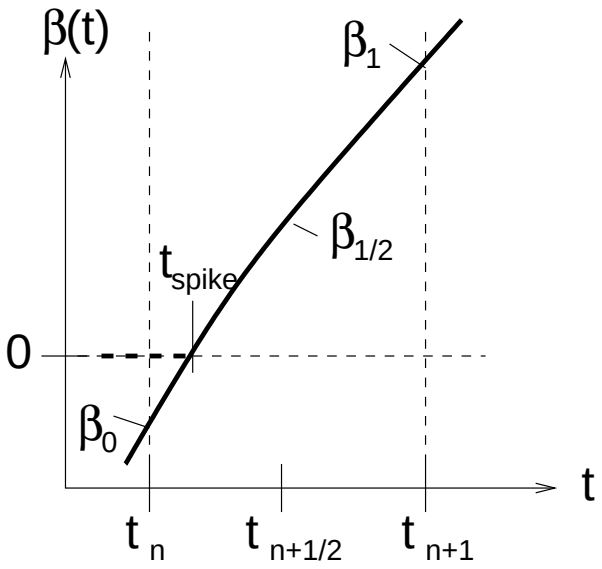

Figure 5.5: *The figure illustrates the extended beta-function, now with negative values for $t < t_{spike}$ instead of zero. The solid thick line is the extended beta-function and the dashed thick line is the ordinary beta-function for negative time argument. The value of β_0, $\beta_{1/2}$ and β_1 is added to the corresponding a and b parameter when the spike is registered.*

The different values of this function that are added for each spike to the corresponding a and b parameters, are illustrated in figure 5.5. These parameters are then used to get the A and B parameters used in eq. 5.40 to get the potential after the spike registrations. Before we use this equation we got to have a corresponding V_{thresh}^{after} value, that pinpoint the potential to a certain value in the timestep. For this we use the potential at the time for the first spike arriving the neuron in the time step. This value is acquired by using the interpolation polynomial betwen V_n and V_{n+1}, i.e. the potentials as if the neuron did not receive any spikes in the time step, together with the spike time for the spike. This procedure is quite straight forward, but if a threshold, either V_{th} or V_h, is passed in the same time step we have a problem. Now the registration of the spikes could alter the exact time for the threshold pass and if we are really unlucky could force a repass of especially the V_h threshold.

To handle this, hopefully not so usually occasion when a threshold is passed in the same time step as the neuron receives a spike, the same procedure from above is followed, but now we only register one spike at a time. First we check if the threshold is passed before the first spike is registered.

If it is, we register the time for the threshold pass and calculate not only $\tilde{V}_{n+1}$ but also $\tilde{V}_n$. These potentials are then used to register the next spike. After that one spike is registered at a time. To do this we have to expand the procedure, now in aditional to $\tilde{V}_{n+1}$ we also calculate $\tilde{V}_n$ for each spike registered. These values are then used to register the next spike. Between

every registration we have to check for repass. If a repass has occurred, we follow the same procedure as for an ordinary threshold pass, but now also calculating $\tilde{V}_n$.

If the threshold is passed after the first spike is registered, we start registering the spikes instead of using the threshold pass procedure. After every spike we check when or if the threshold is passed and compare this time to the next spike time. If it is greater, we register the next spike. Is it lesser we register the threshold pass with the procedure described above.

This is a very cumbersome procedure to implement. First we have to keep track of which order the spikes arrive, not only a singel synapse, but to the whole neuron. Second we have to keep the extra conductance each one of them contributes in separate variables. This could be a very time consuming procedure that could slow down the registration of the presynaptic spikes considerebly. The possible gain and loss this procedure impose have to be evaluated before it is implemented.

Another and not so cumbersome way of avoiding the error caused by the discontinuous first derivative of the synaptic conductance when using RK4, is to use a function that have at least two continuous derivatives for the registration of a presynaptic spike. Shelley and Tao (2001) use

$$G(t) = \begin{cases} 0 & : \quad t < 0 \\ \left(\frac{t}{\tau}\right)^m e^{-\frac{t}{\tau}} & : \quad t \geqslant 0 \end{cases} . \tag{5.45}$$

When $m \geqslant 3$ this function has a continuous second order derivative, and they also present numerical result, proving that this function does not introduce any further error. To test this theory, we are going to use,

$$G(t) = \begin{cases} 0 & : \quad t < 0 \\ \beta^3(t) & : \quad t \geqslant 0 \end{cases} , \tag{5.46}$$

for the conductance, which also has a continuous second derivative. This implementation is tested and the result is presented in the next chapter.

Chapter 6

Result of testing

6.1 Test setup

To test our implementation of the different models from chapter 3 and 5, we have made different tests. All the tests are done with a network of 101 LIFB neurons, simulated with both second order Runge-Kutta (RK2) and fourth order Runge-Kutta (RK4), with and without time stepping schemes, (TS), and with and with out precise spike timing, (PS). The start potentials were initialized randomly, in the range from $-80.0 < V_0^j < -40.0$ where $j = 1, 2, 3, \cdots, 101$.

The total error introduced by a method is estimated by comparing the last potential of all neurons with a canonical run. This run is made, depending on the setup, with the most precise integration method and with a small time step, $\Delta t_c = 2^{-12}\,\mathrm{m}s \simeq 2.44 \times 10^{-4}\,\mathrm{m}s$.

In all tests the error is estimated by taking the membrane potential of the neurons after 500 ms and comparing this with the canonical run after 500 ms.

$$\mathrm{Error}(\Delta t) = \frac{1}{N} \sum_{j=1}^{101} \mid V_{\Delta t}^j(t = 500) - V_c^j(t = 500) \mid \tag{6.1}$$

where $V_{\Delta t}^j$ is the last potential of the jth neuron in the test run and V_c^j is the last potential of the jth neuron in the canonical run. These result are then, for each test, plotted against the time step, used in the simulation. By taking the $\log_{10}$ of both the time steps and the error and then plot the result, we can fit the curve and find out which order in Δt the total error is of.

We have made six different tests, each testing one additional feature in our implementation of the integration methods from chapter 5. The general

equation we integrate in our tests for the jth neuron is

$$\begin{aligned} c_m \frac{dV^j}{dt} &= -i_{L}^{j} - i_{T}^{j} - i_{ext}^{j} - i_{syn}^{j} \\ i_{L}^{j} &= g_{L}(V^j - V_{L}) \\ i_{T}^{j} &= g_{T} m_{\infty}^{j} h^j \times (V_{T} - V^j) \\ i_{ext}^{j} &= H(t - t_0)[A_{\mathrm{DC}} + A_{\mathrm{AC}} \sin(f 2\pi\,(t - t_0) + \phi^j)] \\ i_{syn}^{j} &= \sum_k B_k^j \sum_l G_e(t - t_l^k) + \sum_k C_k^j \sum_l G_i(t - t_l^k) \\ \text{if } V^j(t_l^-) &= V_{th} \Rightarrow \delta(t - t_l) \text{ and } V^j(t_l^+) = V_{res} \end{aligned} \tag{6.2}$$

with the initial conditions

$$\begin{aligned} V^j(0) &= V_0^j \\ h^j(0) &= 0 \end{aligned} \tag{6.3}$$

The dynamics of m_{∞}^j and h^j are described in chapter 3. $H(t - t_0)$ is the Heaviside function, turning the external current on at t_0. A_{DC} and A_{AC} are the amplitudes of the DC and AC-currents, ϕ^j is the phase shift of the AC-input-current for the jth neuron, and f is the frequency of the AC-current. B_k^j and C_k^j denotes the excitatory and the inhibitory weights of the kth neuron and t_l^k is the arrival time of the lth spike from the kth neuron.

All tests where done on a AMD Athlon XP 2000+, with the models from chapter 3 and 5 implemented in NEST, see chapter 4.

(i) The first test is done without any input and T-current, $i_{T}^{j} = i_{ext}^{j} = 0$, i.e. letting all the potentials fall toward the rest-potential, V_{L}.

This tests the precision of the integration methods of RK2 and RK4.

(ii) The second test is done with an external input-current that drives the neurons, but without spiking ability and T-currents, setting $V_{th} = \infty$ and $i_{T}^{j} = 0$. The external currents, i_{ext}^{j}, are turned on at $t_0 = 10.0\,ms$, and their amplitudes are $A_{\mathrm{DC}} = 0.2\mu A/cm^2$ and $A_{\mathrm{AC}} = 0.5\mu A/cm^2$. The frequency and the phase was set to $f = 4Hz$ and $\phi^j = 0$.

This test shows how the integration methods, RK2 and RK4, handles the discontinuous event of an onset of an external current. The onset is chosen to coincide with the fixed time grid.

(iii) In the third test we added the T-current to the equation, letting the membrane potential cross V_h trigging the T-current. We still don't have

any spikes, $V_{th} = \infty$. The external currents are the same as for the second test.

This test shows how the integration methods, RK2, RK2TS, RK4 and RK4TS, handles the discontinuous event of the onset of the T-current.

(iv) The fourth test adds the spiking ability to the neurons, restoring V_{th} to the tabulated value from table 3.1, but keeps the coupling weights B_k^j and C_k^j equal to zero. The parameters for the external current are set to: $t_0 = 1.0$ms, $A_{DC} = -0.2\mu A/cm^2$, $A_{AC} = 0.7\mu A/cm^2$, $f = 4Hz$ and $\phi^j = 0$. The neurons spike about 10 times each.

This test shows how the integration methods, RK2, RK2TS, RK4 and RK4TS, handle the discontinuous event of a spike.

(v) In the fifth test, we have non-zero weights. The parameters for the external currents are set to the same values as in the fourth test except for the phase, which is set to $\phi^j = j\frac{2\pi}{100}$, where $j = 0, 1, 2, \cdots, 100$. An all to all coupling in the network is added. To set the weights, B_k^j and C_k^j, we make a weight array, (WA). The ith element in the weight array, $WA(i)$, is produced by

$$WA(i) = e^{-\frac{1}{2}\left(\frac{i-50}{10}\right)^2} \tag{6.4}$$

where $i = 0, 1, 2, \cdots, 100$. The excitatory and inhibitory weights are then chosen from this array. This is done by letting the excitatory weights from the kth neuron to the jth, B_k^j get the following weights from the weight array

$$B_k^j = WA(i) \ , \tag{6.5}$$

where $j = k + i \bmod 101$ and $i = 0, 1, 2, \cdots, 100$. The inhibitory weights from the kth neuron to the jth, C_k^j get their value from the weight array in a similar way,

$$C_k^j = WA(i) \ , \tag{6.6}$$

but now $j = k + i + 50 \bmod 101$ and $i = 0, 1, 2, \cdots, 100$. The conductances G_e and G_i are the models for the lumping, but not saturating AMPA and $\mathrm{GABA_A}$ synapses, from chapter 3, with $\bar{g}_{\mathrm{AMPA}} = 0.014mS/cm^2$ and $\bar{g}_{\mathrm{GABA_A}} = 0.01mS/cm^2$. We now introduce precis spike timing (PS) to the TS integration methods. This method registers the spikes at the actual time they arrive the neurons. The other integration methods register the spikes to arrive in the beginning of the time step they arrive the neuron. The neurons spike about 11 times each.

This test shows how the integration methods, RK2, RK2TS, RK2TSPS, RK4, RK4TS and RK4TSPS, handle the discontinuous event of receiving a spike.

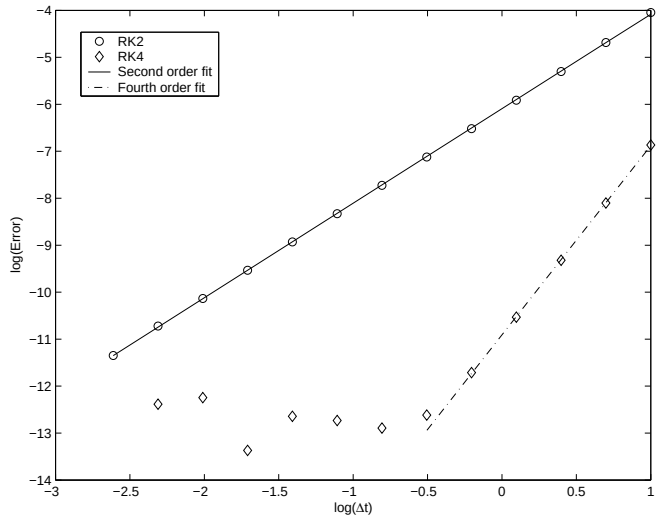

Figure 6.1: *Errors as a function of time step for two different RK algorithms. The test network does not receive any input at all and the T-current is turned off. Circles are the RK2 method and diamonds are the RK4 method. The errors follow the predicted order in Δt from section 5.5, second for RK2 and fourth for rk4. RK4 reaches the machine limit at $\Delta t \simeq 0.3\,ms$.*

(vi) In the sixth test we take the third power of the beta function in G_e and G_i to make the second derivative of the synaptic conductance continuous, see eq. 5.46. The parameters for the external currents are set to the same values as in the fifth test except for the amplitude of the AC-current, which is set to $A_{AC} = 1.0\mu A/cm^2$. The weights are taken from the same weight array as in the fifth test and the maximum conductances are set to $\bar{g}_{\mathrm{AMPA}} = 0.018 mS/cm^2$ and $\bar{g}_{\mathrm{GABA_A}} = 0.01 mS/cm^2$. As with the $\mathrm{GABA_B}$, we can not lump these synapses together, in one state variable, increasing the simulation time dramatical. We now have 20 000 synapses instead of 200. The neurons spike about 5 times each.

A second test with shorter simulation time, now 200 ms instead of 500 ms, and no inhibitory weights, is also done, with almost the same parameter choice as the first test. In this test the neurons spike about two times each.

6.2 Results

The result from the tests were plotted against the size of the time step used in each simulation, and can be seen in figure 6.1 to 6.7, and are commented below.

(i) The result from the first test is shown in figure 6.1. We see that the errors follow the predicted order in Δt from section 5.5, second for RK2

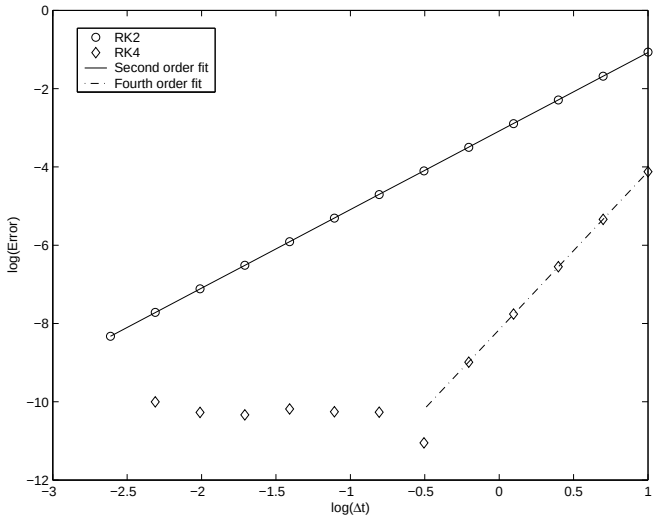

Figure 6.2: *Errors as a function of time step for two different RK algorithms. The test network is receiving external driving current, the T-current is turned off and the neurons are not allowed to spike. Circles are the RK2 method and diamonds are the RK4 method. The errors follow the predicted order in Δt from section 5.5, second for RK2 and fourth for rk4. RK4 reaches the machine limit at $\Delta t \simeq 0.3$ ms.*

and fourth for RK4. RK4 reaches the machine limit at $\Delta t \simeq 0.3$ ms and here the errors is dominated by rounding errors.

(ii) Figure 6.2 shows the result from the second test. We see that if the discontinuous event of the onset of an external current are chosen to coincide with a the fixed time grid no additional error is introduced. The errors follow the predicted order in Δt from section 5.5, second for RK2 and fourth for RK4. RK4 reaches the machine limit at $\Delta t \simeq 0.3$ ms and here the errors is dominated by rounding errors.

(iii) In figure 6.3, the result of the third test is presented. Now our algorithm for dealing with the discontinuous event of onset and offset of the T-current is tested. The errors follow the predicted order in Δt from section 5.5, for the methods with time stepping algorithms, second for RK2TS and fourth for RK4TS, and first order for the methods lacking this algorithm, i.e., RK2 and RK4. RK4TS reaches the machine limit at $\Delta t \simeq 0.02$ ms.

(iv) The result from the fourth test is shown in figure 6.4. In additional to the T-current we now also test the ability to handle the discontinuous event of reseting the potential after a spike. The errors follow the predicted order in Δt from section 5.5, for the methods with time stepping algorithms, second for RK2TS and fourth for RK4TS and first order for the methods lacking this algorithm, i.e., RK2 and RK4. RK4TS reaches the machine

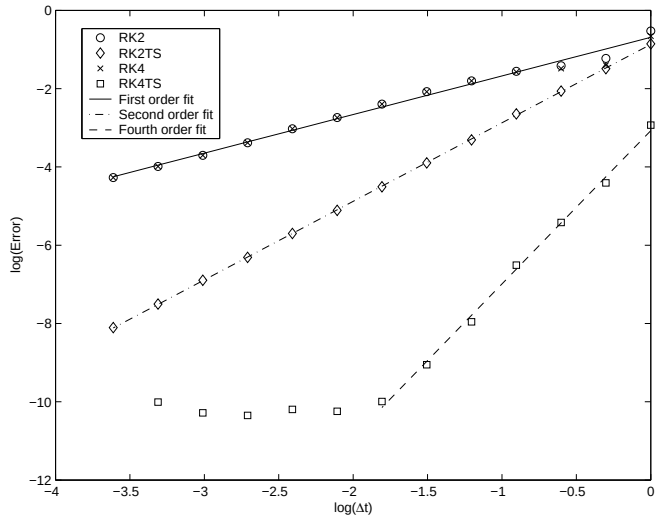

Figure 6.3: *Errors as a function of time step for four different RK algorithms. The test network is receiving external driving current, the T-current is turned on and the neurons are not allowed to spike. Circles are the RK2 method, diamonds are the RK2 method with time stepping algorithm, x's are the RK4 method and squares are the RK4 method with time stepping algorithm. The errors follow the predicted order in Δt from section 5.5, for the methods with time stepping algorithms, second for RK2TS and fourth for RK4TS and first order for the methods lacking this algorithm, i.e., RK2 and RK4. RK4TS reaches the machine limit at $\Delta t \simeq 0.02$ ms.*

limit already at $\Delta t \simeq 0.13$ ms.

(v) In figure 6.5 we see the result from the fifth test. Now the neurons are coupled, and we test how the discontinuous event of registering a presynaptic spike by adding a beta-function to the conductance, bias the system. When the integration of the test potentials stabilized, i.e., not following the flat band in the top of the figure, the errors followed the predicted order in Δt from section 5.5, for the methods with time stepping algorithms and precise spike times, second for both RK2TSPS and RK4TSPS, and first order for the methods lacking the precise spike timing, i.e., RK2, RK4, RK2TS and RK4ts. The errors in the RK4TSPS method do not follow a fourth order fit in Δt because a discontinuity in the second derivative of the potential is introduced when the beta-function is used to register a presynaptic spike, see 5.5. We also see that the RK2TSPS stabilizes at $\Delta t \simeq 0.03$ ms and the RK4TSPS stabilizes at $\Delta t \simeq 0.13$ ms. Even if the errors from the RK4TSPS method do not follow a fourth order fit, it is the most stable integration scheme, and the errors are at least one order of magnitude below the errors from the RK2TSPS method. This means that we can use a time step that is ten times larger when integrating with RK4TSPS than with RK2TSPS, with the same precision as result. The number of time steps used in the

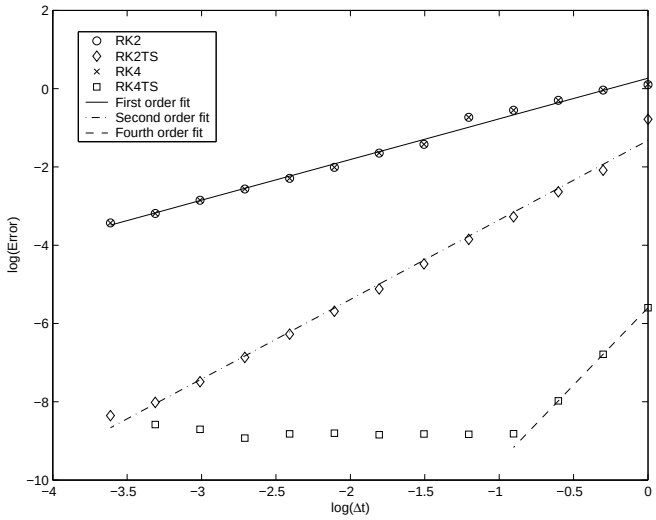

Figure 6.4: *Errors as a function of time step for four different RK algorithms. The test network is receiving external driving current, the T-current is turned on and the neurons are allowed to spike but are not coupled. Circles are the RK2 method, diamonds are the RK2 method with time stepping algorithm, x's are the RK4 method and squares are the RK4 method with time stepping algorithm. The errors follow the predicted order in Δt from section 5.5, for the methods with time stepping algorithms, second for RK2TS and fourth for RK4TS and first order for the methods lacking this algorithm, i.e., RK2 and RK4. RK4TS reaches the machine limit already at $\Delta t \simeq 0.13$ ms.*

simulation is then reduced by a factor of ten.

Shelley and Tao (2001) do not do the RK4TSPS test with a conductance that is discontinuous in its first derivative, and they therefore do not get the second order fit for their fourth order scheme. They are using a conductance that is continuous in its second derivative, and consistent with our error analysis, follows a fourth order fit. In the sixth and last test we are trying to reproduce this result.

(vi) The result from our sixth test are shown in figure 6.6. We now see that the only methods that stabilize are the RK2TSPS and the RK4TSPS methods. All the other methods are unstable and do not manage, no matter what precision, to produce the same amount of spikes as our canonical run. The plot also show us that the estimated errors to RK4TSPS, follows a second order fit in Δt, and not a fourth order fit. This is not what Shelley and Tao (2001) got and certainly not what we predicted from the analysis in section 5.5. The error caused by the RK2TSPS method, after stabilization, follows second order in Δt, indicating that at least the result from this method are consistent with our analytical results.

To investigate the method a bit more we did a second test, now only 200 ms long, with only excitatory input. The result is plotted in figure 6.7.

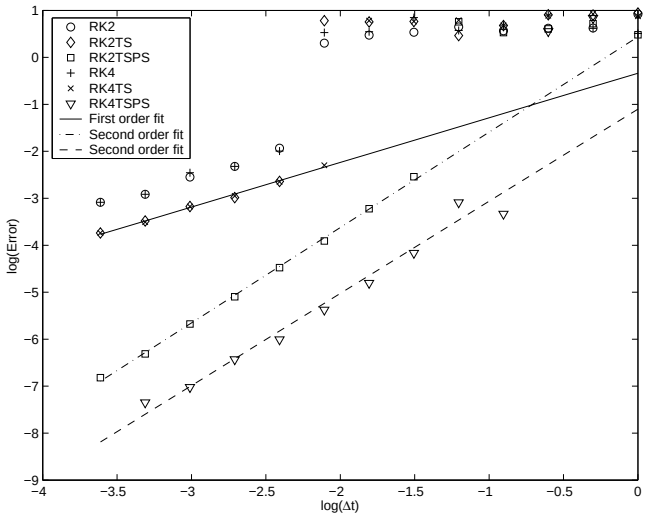

Figure 6.5: *Errors as a function of time step for six different RK algorithms. The test network is receiving external driving current, the T-current is turned on and the neurons are allowed to spike and are coupled. Circles are the RK2 method, diamonds are the RK2 method with time stepping algorithm, squares are the RK2 method with time stepping and precise spike timing, crosses are the RK4 method, x's are the RK4 method with time stepping algorithm and triangles are the RK4 method with time stepping algorithm and precise spike timing. When the integration of the test potentials stabilized, i.e., not following the flat band in the top of the figure, the errors followed the predicted order in Δt from section 5.5, for the methods with time stepping algorithms and precise spike times, second for both RK2TSPS and RK4TSPS and first order for the methods lacking the precise spike timing, i.e., RK2, RK4, RK2TS and RK4ts.*

We see the same results as in the former test, but here the tests with a time step in the range, $0.1\,\mathrm{m}s \leq \Delta t \leq 1\,\mathrm{ms}$, with RK4TSPS method indicates, with a little good will, that the estimated error follows a fourth order fit in Δt. But for $\Delta t \leq 0.1\,\mathrm{ms}$ it drops to second order. It appears that the error reach the machine limit for $\Delta t \leq 2 \times 10^{-3}\,\mathrm{ms}$. The error from the RK2TSPS method, follows a second order fit in Δt and the other methods do not stabilize at all.

The last two test-runs indicates that an error we have not anticipated, is introduced to the RK4TSPS method. At the time of writing we have not yet resolved what these could be. One may speculate if the difference in number of synapses between test five and six has any thing to do with the failure. Because of the lumping and non-lumping features of the synapses used we have a total of 200 individual synapses in the fifth test and a total of 20 000 individual synapses in the sixth test. If more time were given, we would try to reproduce the results from Shelley and Tao (2001), now with the T-current turned of and we would use the same function for the conductances as they use.

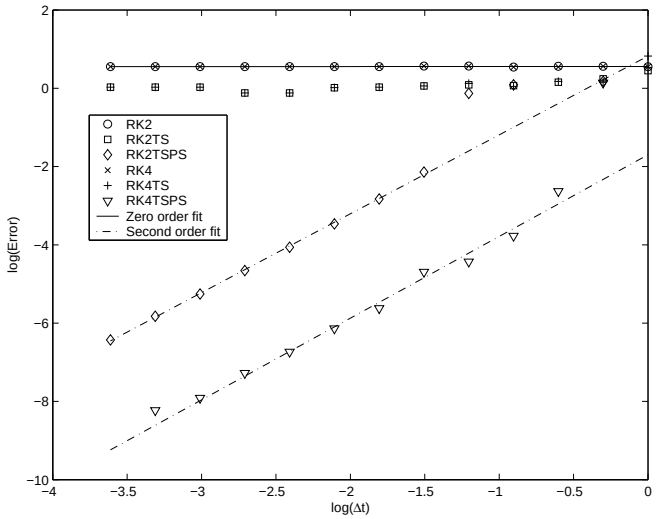

Figure 6.6: *Errors as a function of time step for six different RK algorithms. The test network is receiving external driving current, the T-current is turned on and the neurons are allowed to spike and are coupled. The postsynaptic conductances is now modeled by a beta function in third. Circles are the RK2 method, squares are the RK2 method with time stepping algorithm RK2TS, diamonds are the RK2 method with time stepping and precise spike timing, RK2TSPS, x's are the RK4 method, crosses are the RK4 method with time stepping algorithm, RK4TS and triangles are the RK4 method with time stepping algorithm and precise spike timing, RK4TSPS. Here only the errors from the RK2TSPS method follows the predicted order from section 5.5. The errors from the RK4TSPS method should follow a fourth order fit but follows a second order. The rest of the methods are not stable, and their errors do not follow any order in .*

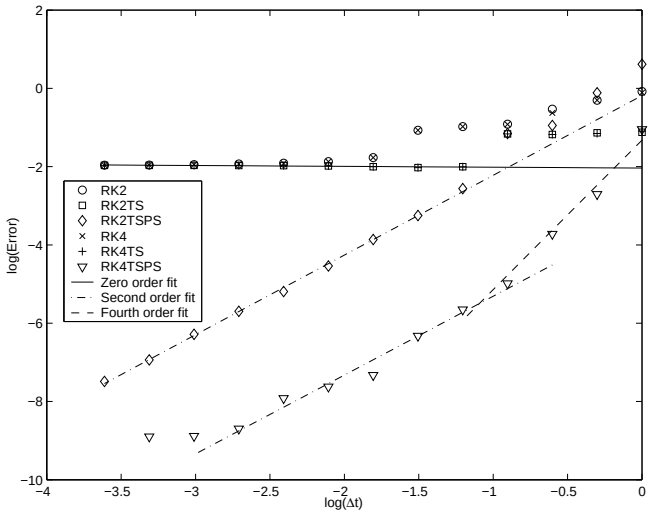

Figure 6.7: *Errors as a function of time step for six different RK algorithms. The test network is receiving external driving current, the T-current is turned on and the neurons are allowed to spike and are coupled. The postsynaptic conductances is now modeled by a beta function in third. Circles are the RK2 method, squares are the RK2 method with time stepping algorithm RK2TS, diamonds are the RK2 method with time stepping and precise spike timing, RK2TSPS, x's are the RK4 method, crosses are the RK4 method with time stepping algorithm, RK4TS and triangles are the RK4 method with time stepping algorithm and precise spike timing, RK4TSPS. The errors from the RK2TSPS method follows the predicted order from section 5.5. The errors from the RK4TSPS method have tendency to follow a fourth order fit in the interval of* $0.1\ ms \leq \Delta t \leq 1\ ms$*, but flattens to a second order fit for* $\Delta t \leq 0.1\ ms$*. The rest of the methods are not stable, and their errors do not follow any order in .*

Chapter 7

Discussion

The overall goal I had for this thesis was to implement efficient and reliable models of spiking neurons and conductance-based synapses, present in the *dorsal lateral geniculate nucleus* (LGN) in a cat, in the framwork of the NEST neuronal network simulator. These models are escential for the development of network models for the LGN circuit.

I have used the *leaky-integrate-and-fire-or-burst* (LIFB) model (Smith et al. 2000) as a primary spiking neuron model for the bursting relay cell in the LGN. This model can also be used to model other neurons in the LGN, which also show bursting behavior (Smith and Sherman 2002). The models of the synapses were based on the models developed by Destexhe et al. (1998). These are conductance-based models of the ionotropic AMPA and $GABA_A$ receptors and of the metabotropic $GABA_B$ receptor. The latter could be used to model the metabotropic glutamate receptor, mGluR5, that is present in the LGN. Emri et al. (2003) used the same model to simulate the metabotropic, mGluR1 glutamate receptor.

To satisfy the demand of reliable implementation I integrated the LIFB model with the second and fourth order of the general purpose methods of Runge-Kutta (RK). These methods also gave me a formal framework to analyze the different errors introduced when the model was integrated. The LIFB model introduces five different discontinuities that when integrated with the RK methods, cause errors additional to the truncation errors introduced by the RK methods itself. Three especially important discontinuities are: the reseting of the potential after a spike, the onset and offset of the T-current causing the bursting activity, and the onset of postsynaptic currents following a presynaptic spike. I extend the former work of Hansel et al. (1998) and Shelley and Tao (2001) to a generic procedure for handling these kinds of discontinuity errors. The procedure for the first and second type of discontinuity, were implemented in NEST and successfully tested. A method for handling

the third discontinuity when the model is integrated by the fourth order RK method is proposed, based on the same generic procedure, but unfortunately I was not able to test it. Another procedure to handle the third discontinuity when using the RK4 method, is introduced by Shelley and Tao (2001). They use a model of the conductance based on a function that is continuous in its second derivative, thus only introducing a discontinuity error that is of the same order as introduced by the integration method. I show in my analysis that this methods should work but I fail to reproduce the numerical results when the implementation is tested. At the time of writing I unfortunatly do not know wy I could not reproduce the result. But I show that the present RK4TSPS method integrate a network of coupled LIFB neurons with ten times the precision then the RK2TSPS method, thus reducing the number of time steps used in the simulation by a factor of ten.

By registering a presynaptic spike by moving it to the fixed time grid, i.e, to the beginning of the time step during which the spike arrives, the registration procedure is simplified. I show, both analytically and numerically, that this introduces an error that is greater than the error introduced by the integration methods. I suggest a solution to this error and it is successfully tested.

The method of exact integration (Rotter and Diesmann 1999) provides an efficient and reliable framework for the integration of functions that are solutions of linear, time invariant and continuous systems of differential equations. I implemented the synapses from Destexhe et al. (1998) with a function that could be integrated by exact integration, the beta-function, so I could benefit from this effective integration method. The method also provides a straight forward procedure of registering presynaptic spikes. A single value is just added to one of the variables used to integrate the conductance, and several synapses with the same dynamics could also be lumped together in the same state variable, accelerating the calculation of the synaptic conductance.

By using the two integration methods together with the generic procedure for handling discontinuity errors, I benefit from the strengths from both integration methods, while the limitations each one of them impose is by-passed. The RK method is a generic integration method, extensiely used. But as explained in section A.1 in the appendix and in section 5.5, errors are introduced when discontinuities in the integrated model occur. This is handled by our error handling procedures. The effective and precise method of exact integration is unfortunately limited to linear, time invariant and continuous systems. It could therefore not be used to integrate the whole LIFB model, but I successfully used it to simulate smaller parts of the model, especially the synaptic conductances.

Almost all models and error handling procedures, presented and discussed in this thesis are implemented in NEST. This provides the group in Ås, and others that are using NEST, with important simulation tools, when network models of the LGN are developed. I have also developed and implemented a new technique dealing with conductance-based synapses in NEST. The former was based on the event system of NEST, briefly discussed in chapter 4. This introduces delays and unnecessary time spent on handling events. The event handling is effective when communications are made between many nodes. As each synapse only delivers synaptic conductance to just one receiving neuron, it may be integrated into this neuron. I have implemented a way of doing this. The actual synapse is created when the connection betwen two neurons is made, and is then stored in the receiving neuron. If the synapse were able to lump incomming spikes and another connection is made to the same recieving neuron trough the same synapse, instead of creating a new synaspe the old one is used. Thus the synaptic handling is accelerated, and I was able to implement the lumping ability of the ionotropic synapses in a straight forward way.

The next step would of course be to simulate the feedforward circuit of LGN presented in chapter 2. To do this, a model for the triadic circuit, in the X-pathway has to be developed and tested. Other neurons in the circuit should also be included in the network model, so I could explore more of the signal processing ability of the EVP in a cat. I should also try to locate the error done when the method of RK4TSPS and a conductance that is continuous in its second derivative were used. A numerical implementation of the sketched procedure, that handles the error introduced by the discontinuous event of registering a presynaptic spike by a beta-function, should also be done, so the method could be tested.

Appendix A

Integration methods

The differential equations from chapter 3 are all *initial value problems* (IVP). In general an IVP can be expressed as

$$\frac{dv}{dt} = f(t, v) \; ; \; v(0) = v_0 \; . \tag{A.1}$$

There exists a pletora of integration methods to solve this kind of problems, all of which try to solve for $v(t)$, where t $>$ 0. This is most typical done by dividing the time course between 0 and t into M intervals and these could be of equal or non-equal size. The value of $v(t)$ is then calculated by iteration from one value, on the grid $v(t_i)$, to another $v(t_{i+1})$, where $t_{i+1} = t_i + \Delta t$, until $v(t)$ is reached. The integration methods presented here, Runge-Kutta (RK) and exact integration rely fundamently on the Taylor expansion of a function. If $v(t)$ has continuous derivatives of all orders in the interval $[t_i, \, t_{i+1}]$, it could be represented as

$$v(t_{i+1}) = v(t_i) + v'(t_i)\Delta t + \frac{1}{2}v^{(2)}(t_i)\Delta t^2 + \frac{1}{6}v^{(3)}(t_i)\Delta t^3 + \ldots \; . \tag{A.2}$$

This series give a robust fundation for the integration methods and the error analysis.

A.1 Runge-Kutta metods

The exposition of the Runge-Kutta methods is based on Mathews (1987).

Before we present the RK methods we first present Taylor's integration method which the RK methods are derived from. Taylor methods are derived directly from the Taylor series and are often called Taylor's formula. For numerical puposes, we do not use eq. A.2 but a finite sum to approximate the $v(t_{i+1})$ value. If $v(t)$ has derivatives of all orders in the semi open interval

of $[t_i,\ t_{i+1})$[1] and we fix a number N, $v(t_{i+1})$ can be expressed by the Taylors formula, (Finney and Thomas 1994)

$$v(t_{i+1}) \simeq v(t_i) + v'(t_i)\Delta t + \frac{1}{2}v^{(2)}(t_i)\Delta t^2 + \ldots + \frac{1}{N!}v^{(N)}(t_i)\Delta t^N \quad . \tag{A.3}$$

The error in the approximation in equation A.3 is the *local truncation error* LTE and is given by

$$\frac{1}{(N+1)!}v^{(N+1)}(c)\Delta t^{N+1} \quad , \tag{A.4}$$

where $t_i < c < t_{i+1}$ and $\Delta t > 0$. If the N'th order of the Taylor method is used to integrate one step then the LTE is of order $\mathcal{O}(\Delta t^{N+1})$. N can be chosen large so the LTE becomes small. The LTE is the error introduced when we go from one step to an other, and this error are accumulated to the *global truncation error* (GTE), which is of one order lower than the LTE, i.e. of order $\mathcal{O}(\Delta t^N)$ (Mathews 1987).

If, the kth derivative is discontinuous in the open interval of $[t_i,\ t_{i+1})$, it introduce a local error of $\mathcal{O}(\Delta t^k)$. This error is introduced, to the next potential, $v(t_{i+1})$. If the GTE is of order $\mathcal{O}(\Delta t^N)$ the Nth or higher derivative can actually be discontinuous and the GTE is not altered. But if the discontinuity occurs at the time step, t_{i+1}, it is not contained in the semi open interval of $[t_i,\ t_{i+1})$ and no error from the discontinuity is introduced to $v(t_{i+1})$. When the next value, $v(t_{i+2})$, is calculated from the values of the derivatives at t_{i+1}, there exists no discontinuities either in this interval. So neither now is any error introduced to $v(t_{i+2})$. This is illustrated in figure A.1.

A drawback of the Taylor method is that we have to evaluate the N'th derivative of $v(t_i)$ to approximate $v(t_{i+1})$. This could be hard work if the IVP from equation A.1 is complicated. Then the method of RK is a more convenient method to use. Every RK method of order N is derived from a Taylor method in such a way that the GTE is of order $\mathcal{O}(\Delta t^N)$ (Mathews 1987). To approximate $v(t_{i+1})$ from $v(t_i)$ with RK methods we only have to evaluate the first derivative of the $v(t)$ function. From equation A.1 we see that this is the value of the $f(t, v(t))$ function. A trade-off to this is that we have to make several functional evaluations in the time step. These methods could be constructed for any order of N, but we are only going to use two versions, one of the second and one of the fourth order in this thesis. These are also the most used.

In general the RK methods use a linear combination of the function values $f_0, f_1, \cdots, f_{N-1}$, evaluated somewhere in the semi open interval of $[t_i,\ t_{i+1})$,

[1] In the original formula in Finney and Thomas (1994) the interval is open, and it should contain the evaluation point, for us it is t_i. When this point is included the interval becomes semi open.

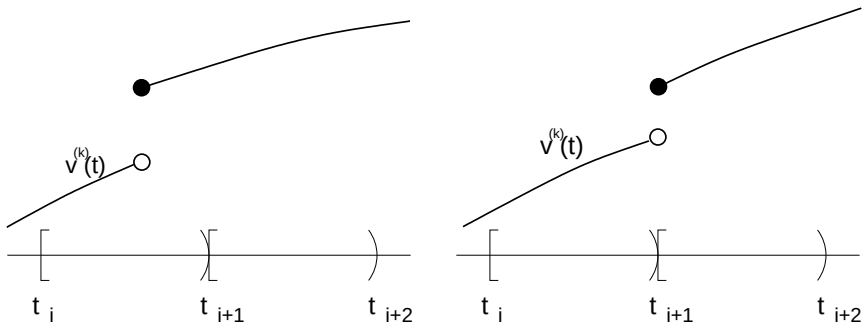

Figure A.1: *The two panels show the two semi open intervals of* $[t_i,\, t_{i+1})$ *and* $[t_{i+1},\, t_{i+2})$*, and the* k*th derivative of* $v(t)$*, which is discontinuous. In the left panel the discontinuity appears inside the first interval and therefore imposes a LTE of order* $O(\Delta t^k)$*, to* $v(t_{i+1})$*. If the GTE is of the same or lower order the discontinuity does not alter it, but if the GTE is of higher order it is now lowered to* $O(\Delta t^k)$*. In the right panel the discontinuity appear at* t_{i+1}*. Neither of the two semi open intervals include the discontinuity and therefore does it not introduce any additional error to the integrated* $v(t)$*.*

to approximate the mean rise of the function in that time step. This value is then used to approximate, by extrapolation, the next value,

$$v(t_{i+1}) = v(t_i) + \Delta t(a_0 f_0 + a_1 f_1 + \cdots + a_{N-1} f_{N-1}) \ , \tag{A.5}$$

where

$$\begin{aligned} f_0 &= f(t_i, v(t_i)), \\ f_1 &= f\left(t_i + b_1 \Delta t,\ v(t_i) + c_{10} \Delta t f_0\right), \\ &\vdots \\ f_{N-1} &= f\left(t_i + b_{N-1}\Delta t,\ v(t_i) + \right. \\ &\quad \left. \Delta t\left[c_{(N-1)0} f_0 + c_{(N-1)1} f_1 + \cdots + c_{(N-1)(N-2)} f_{N-2}\right]\right), \end{aligned} \tag{A.6}$$

where N is the order of the RK method. f_0 is the derivative of $v(t)$ at time t_i, i.e., at the fixed time grid at beginning of the time step. The other f-functions are an approximation of the derivative of $v(t)$ somewhere inside the semi open time interval. b_k is bounded by $0 < b_k \leq 1$ and determine where in the time step the f-function is evaluated. The c_{kj} coefficients are the weights of the kth f-function, f_k and these weights are bounded by b_k, as $b_k = \sum_j^{N-2} c_{kj}$. Note that the kth f-function, f_k, where $k = 1,\, 2 \cdots N-1$, takes an approximation of $v(t)$ at $t_i + b_k \Delta t$, formed by a linear combination of the former function values, as argument when the approximated first derivative is evaluated.

Second order Runge-Kutta (RK2)

We are going to use eq. A.5 and eq. A.6 together with the Taylor series, eq. A.2, to show how the coefficients are chosen for a second order RK method. To simplify notation we set, $t_i = t$ and $v(t_i) = v$ or $v(t)$. We start with the Taylor formula for $v(t)$

$$v(t + \Delta t) = v(t) + \Delta t v'(t) + \frac{1}{2}\Delta t^2 v''(t) + D\Delta t^3 + \cdots \ , \tag{A.7}$$

where D is a constant involving the third derivative of $v(t)$. The other terms in the serie involve powers of Δt^j for $j > 3$. We can express these derivatives with $f(t, v)$ The first derivative is

$$v'(t) = f(t, v) \ , \tag{A.8}$$

and the second derivative, with use of the chain rule is

$$v''(t) = f_t(t, v) + f_v(t, v)f(t, v) \ . \tag{A.9}$$

Using eq. A.8 and eq. A.9 in eq. A.7 we get an expression for $v(t + \Delta t)$

$$\begin{aligned} v(t + \Delta t) &= v(t) + \Delta t f(t, v) + \frac{1}{2}\Delta t^2 f_t(t, v) \\ &\quad + \frac{1}{2}\Delta t^2 f_v(t, v)f(t, v) + D\Delta t^3 + \cdots \ . \end{aligned} \tag{A.10}$$

Now consider the RK eq. A.5 and eq. A.6. For the second order we have

$$\begin{aligned} v(t + \Delta t) &= v(t) + \Delta t(a_0 f_0 + a_1 f_1) \\ f_0 &= f(t, v), \\ f_1 &= f\left(t + b_1\Delta t,\ v + c_{10}\Delta t f_0\right), \end{aligned} \tag{A.11}$$

We then use the Taylor polynomial approximation to expand f_1

$$f_1 = f(t, v) + b_1\Delta t f_t(t, v) + c_{10}\Delta t f_v(t, v)f(t, v) + d\Delta t^2 + \cdots \ , \tag{A.12}$$

where d includes the second partial derivatives of $f(t, v)$. Then eq. A.12 is used in eq. A.11 to get the expression for $v(t + \Delta t)$

$$\begin{aligned} v(t + \Delta t) &= v(t) + (a_0 + a_1)\Delta t f(t, v) + a_1 b_1 \Delta t^2 f_t(t, v) \\ &\quad + a_1 c_{10}\Delta t^2 f_v(t, v)f(t, v) + a_1 d\Delta t^3 + \cdots \ . \end{aligned} \tag{A.13}$$

Comparing similar terms in eq. A.10 and eq. A.13 yields the following conclusions,

$$\begin{aligned} a_0 + a_1 &= 1 \\ a_1 b_1 &= \frac{1}{2} \\ a_1 c_{10} &= \frac{1}{2} \ . \end{aligned} \tag{A.14}$$

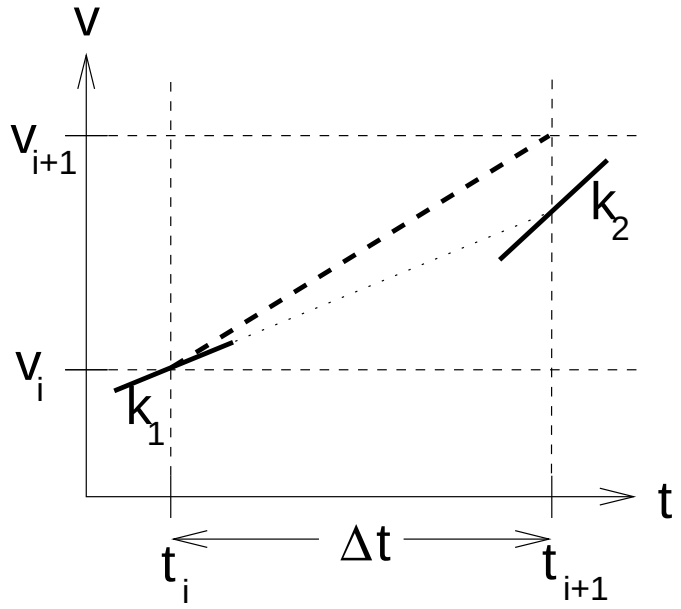

Figure A.2: *The figure illustrates the geometry of eq. A.15. The solid small lines are approximations of the slope in the beginning, k_1 and the end, k_2. The dashed line is the mean of k_1 and k_2 and are used to find the approximation of the final $v(t_{i+1})$ value. The dotted line are the extrapolation of the first slope, used to find an approximation of $v(t)$ in the end of the timestep. This value is then used in $f(t, v)$ to find an approximation of the slope in the end of the time step.*

So if we require that a_0, a_1, b_1 and c_{10} satisfy the relations in eq. A.14, then the RK2 method in eq. A.13 will have the same order of accuracy as the Taylor's method in eq. A.10. With four unknown and three equations the system is underdetermined, and we choose one of the coefficients. We choose $a_0 = \frac{1}{2}$, and the other coefficients follows, $a_1 = \frac{1}{2}$, $b_1 = 1$ and $c_{10} = 1$. With these, eq. A.13 becomes

$$\begin{aligned} v(t + \Delta t) &= v(t) + \frac{\Delta t}{2}(k_1 + k_2) + \mathcal{O}(\Delta t^3) \\ k_1 &= f(t, v) \\ k_2 &= f(t + \Delta t, v + \Delta t k_2) \ . \end{aligned} \tag{A.15}$$

We now have a RK2 method with a LTE of order $\mathcal{O}(\Delta t^3)$ giving a GTE of order $\mathcal{O}(\Delta t^2)$. The fraction $\frac{k_1+k_2}{2}$ is an approximation of the mean slope to $v(t)$, in the time step. This slope is then used to approximate the new $v(t_{i+1})$ value, from $v(t_i)$. This is illustrated in figure A.2, where the small solid lines are the local slopes of, k_1 and k_2, and the dashed line is the mean slope, used to find the final $v(t_{i+1})$ value.

Fourth order Runge-Kutta (RK4)

To develop equations like eq. A.15 for the fourth order Runge-Kutta (RK4) from the general eq. A.5 is a much more complex issue and beyond the scope of this Appendix.

The RK4 method is a commonly used integration method because it is fairly straightforward and it evaluates the new $v(t_{i+1})$ value with a small LTE, of order $\mathcal{O}(\Delta t^5)$, leading to a GTE of order $\mathcal{O}(\Delta t^4)$. The RK4 method is of the following form,

$$\begin{aligned} v(t+\Delta t) &= v(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(\Delta t^5) \\ k_1 &= f(t, v) \\ k_2 &= f(t + \frac{\Delta t}{2}, v + \frac{\Delta t}{2} k_1) \\ k_3 &= f(t + \frac{\Delta t}{2}, v + \frac{\Delta t}{2} k_2) \\ k_4 &= f(t + \Delta t, v + \Delta t k_3) \ . \end{aligned} \tag{A.16}$$

Even if the choice of coefficients in (A.16) is complex we can study the geometry of them in the same way as we did for the RK2 method. The k_i coefficients are approximations of the slope to $v(t)$ in the time step, in the beginning, k_1, in the middle, k_2 and k_3, and in the end, k_4 . These are then weighted to form an approximation of the mean slope of $v(t)$ in the time step. This slope is then used to get the approximation of the final $v(t_{i+1})$ value. See figure A.3.

A.2 Exact integration

Exact integration is as the name says, an integration method that solve the IVP of eq. A.1 with no truncation error introduced, and it is a very efficient integration method too. To be integrated with exact integration the IVP have to be time invariant, linear and the integrated function got to have continuous derivatives of all orders. We are only breifly going to present these method, for a more thorough presentation we recommend Rotter and Diesmann (1999).
Consider the following homogenous, linear and timeinvariant system of first order differential equations

$$\dot{\mathbf{y}} = A\mathbf{y} \ ; \ \mathbf{y}(\mathbf{0}) = \mathbf{y_0} \ . \tag{A.17}$$

Here A is a square matrix with fixed constants that characterize the system. Any higher-order linear systems could be substituted by a first-order system. The solution of this homogeneus system is given by the columns in the matrix

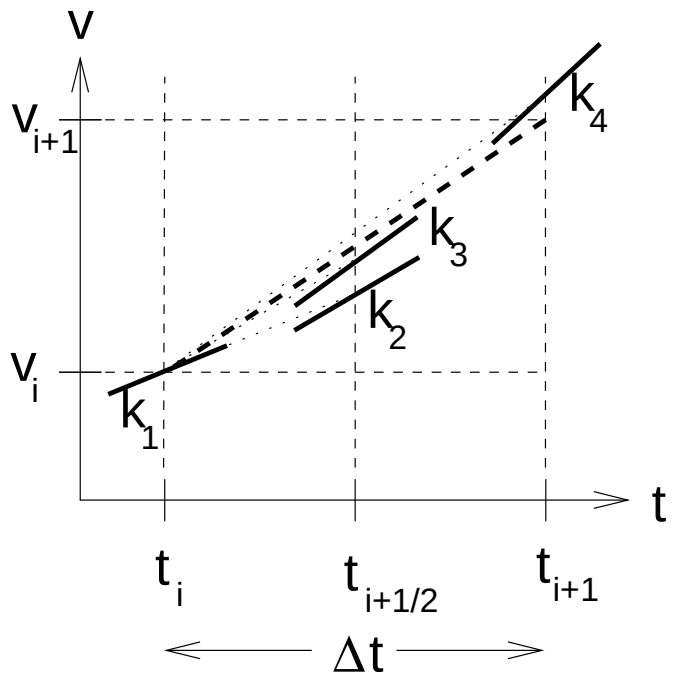

Figure A.3: *The figure illustrate the geometry of eq. A.16. The solid small lines are approximations of the slope: in the beginning, k_1, in the middle: k_2 and k_3, and in the end, k_4. The dashed line is the weighted mean of those and are used to find the approximation of the final $v(t_{i+1})$ value. The dotted lines are extrapolations of the first three slopes, used to find an approximation of $v(t)$ in the timestep. This value is then used in $f(t, v)$ to find an approximation of the next slope.*

exponential $\mathbf{y} = \mathbf{y_0} e^{A\,t}$. This can be checked by substituting all derivatives in a higher dimensional version of Taylor's formula, eq. A.2, with $\mathbf{y}^{(k)} = A^k \mathbf{y}$. If we fix the time to a grid with the size of Δt, we can express the value of, $\mathbf{y}(t_{i+1})$, with the former, $\mathbf{y}(t_i)$, as:

$$\mathbf{y}(t_{i+1}) = e^{A\Delta t}\mathbf{y}(t_i) \quad . \tag{A.18}$$

$e^{A\Delta t}$ is now called a *time-evolution operator* and could be used iterative to reach values on the grid for y

Exponential decay

One of the most elementary functions is the exponential decay expressed by

$$\dot{\eta} + a\eta = 0,\ \eta(0) = \eta_0 \quad , \tag{A.19}$$

for a scalar variable η. The explicit solution of this system is:

$$\eta(t) = \eta_0 e^{-at} \quad . \tag{A.20}$$

If we bring eq. A.19 to the normal form of eq. A.17 we identify

$$y = \eta,\ y(0) = \eta_0,\ A = -a\ . \tag{A.21}$$

With a fixed time grid we get a time-evolution operator for the exponential decay:

$$y_{i+1} = e^{-a\Delta t} y_i\ . \tag{A.22}$$

Alpha- and beta- functions

Two commonly used functions in neural simulations are the alpha- and beta-functions. These two are solutions of

$$\ddot{\eta} + (a+b)\dot{\eta} + (ab)\eta = 0,\ \eta(0) = 0,\ \dot{\eta}(0) = \dot{\eta}_0\ , \tag{A.23}$$

a second order differential equation. For $a = b$ the solution is an alpha-function and with $a \neq b$ the solution is the beta-function. The explicit solution, with the alpha-function comming first, of eq. A.23 is

$$\eta(t) = \dot{\eta}_0 t e^{-at} \ \text{ and } \ \eta(t) = \frac{\dot{\eta}_0}{b-a}\left(e^{-at} - e^{-bt}\right), \tag{A.24}$$

To bring this equation over to the normal form of eq. A.17 we have to rephrase it into a two-dimensional first-order system. This could be done in many different ways, but a convinient way to this for the system in eq. A.23 is

$$\mathbf{y} = \begin{bmatrix} b\eta + \dot{\eta} \\ \eta \end{bmatrix},\ \mathbf{y}(\mathbf{0}) = \begin{bmatrix} \dot{\eta}_0 \\ 0 \end{bmatrix},\ A = \begin{bmatrix} -a & 0 \\ 1 & -b \end{bmatrix}. \tag{A.25}$$

With the time fixed to a grid, the time-evolution operator or the matrix exponential, is then

$$e^{A\Delta t} = \begin{bmatrix} e^{-a\Delta t} & 0 \\ \Delta t e^{-a\Delta t} & e^{-a\Delta t} \end{bmatrix}, \tag{A.26}$$

for the alpha funtion, and

$$e^{A\Delta t} = \begin{bmatrix} e^{-a\Delta t} & 0 \\ \frac{1}{b-a}\left(e^{-a\Delta t} - e^{-b\Delta t}\right) & e^{-b\Delta t} \end{bmatrix}, \tag{A.27}$$

for the beta function. When this is used in eq. A.18 we get the next value in the state vector $\mathbf{y}(t_{i+1})$ from the former $\mathbf{y}(t_i)$. The actual value of the alpha- or beta-function at time t_{i+1} is given by the second state variable of $\mathbf{y}(t_{i+1})$.

Appendix B

Interpolation parameters

In chapter 5, the procedure for calculating the potential after a threshold pass, with no further numerical error, is presented. The equation that finally gives us the potential comes from two linear equations that relate $\tilde{V}_{n+1}$ and $\tilde{V}_n$ with each other. The first potential is the one we want to solve for. The two equations are

$$\tilde{V}_{n+1} = A\,\tilde{V}_n + B \ , \tag{B.1}$$

and

$$V_{thresh}^{after} = K_1\,\tilde{V}_n + K_1\,\tilde{V}_{n+1} + K_3 \ . \tag{B.2}$$

The first equation is obtained from the second or fourth order Runge-Kutta integration schemes, in eq. 5.4 and eq. 5.5. The second equation is obtained from the linear or cubic interpolation polynom from eq. 5.32 and eq. 5.34. We solve these for the potential after the threshold pass, and get

$$\tilde{V}_{n+1} = \frac{(1-A)\,V_{thresh}^{after} + B\,K_1 - (1-A)\,K_3}{K_1 + (1-A)\,K_2} \ . \tag{B.3}$$

This equation is general and gives the potential after a threshold pass for both RK2 and RK4. The only thing that differd are the parameters. For RK2 are these given in chapter 5, by eq. 5.36 and eq. 5.39, and the parameters for the RK4 are given by

$$\begin{aligned} A &= 1 + \frac{\Delta t}{6}\,(A_1^{help} + 2\,A_2^{help} + 2\,A_3^{help} + A_4^{help}) \\ B &= \frac{\Delta t}{6}\,(B_1^{help} + 2\,B_2^{help} + 2\,B_3^{help} + B_4^{help}) \ , \end{aligned}$$

where

$$
\begin{aligned}
A_1^{help} &= a_0 \\
A_2^{help} &= a_{1/2}\left(1+\frac{\Delta t}{2}A_1^{help}\right) \\
A_3^{help} &= a_{1/2}\left(1+\frac{\Delta t}{2}A_2^{help}\right) \\
A_4^{help} &= a_1\left(1+\Delta t\, A_3^{help}\right)
\end{aligned}
$$

$$
\begin{aligned}
B_1^{help} &= b_0 \\
B_2^{help} &= b_{1/2}+\frac{\Delta t}{2}B_1^{help}a_{1/2} \\
B_3^{help} &= b_{1/2}+\frac{\Delta t}{2}B_2^{help}a_{1/2} \\
B_4^{help} &= b_1+\Delta t\, B_3^{help}a_1 \quad ,
\end{aligned}
$$

and

$$K_1 = 1+a_0\frac{t_{th}}{\Delta t}\Delta t-\left(\frac{t_{th}}{\Delta t}\right)^2(2\,\Delta t\,a_0+3)+\left(\frac{t_{th}}{\Delta t}\right)^3(\Delta t\,a_0+2) \quad \text{(B.4)}$$

$$K_2 = \left(\frac{t_{th}}{\Delta t}\right)^2(3-\Delta t\,a_1)+\left(\frac{t_{th}}{\Delta t}\right)^3(\Delta t\,a_1-2) \quad \text{(B.5)}$$

$$K_3 = b_0*\frac{t_{th}}{\Delta t}\Delta t-\left(\frac{t_{th}}{\Delta t}\right)^2\Delta t\,(2\,b_0+b_1)+\left(\frac{t_{th}}{\Delta t}\right)^3\Delta t\,(b_0+b_1)\,. \quad \text{(B.6)}$$

Appendix C

Selected C++ code

C.1 Introduction

In this appendix selected C++ code files are presented. They are chosen so they give a consistent view of my contribution to NEST, with the discontinuity handling and the implementation of the synapses emphasized.

C.2 LIFB Neuron with RK4TS method

The LIFB model is implemented in several different neuron models in NEST, which this is the most advanced. It implements the LIFB model with a Runge-Kutta algorithm of fourth order, with the time stepping mechanism for dealing with the discontinuities of reseting the potential after a spike and the onset and offset of the T-current. It also shows how the synapse-connect mechanism works and how we have implemented the lumping and non-lumping feature of the synaptic integration.

lifb neuron rk4ts.h

```
#ifndef LIFB_NEURON_RK4TS_H
#define LIFB_NEURON_RK4TS_H

#include "nest.h"
#include "event.h"
#include "node.h"
#include "synapse.h"

/* BeginDocumentation
Name: LiFBNeuron_rk2ts - Leaky integrate, fire and burst neuron with 2:nd order time-stepping schemes
Description:

  The LiFB neuron implements Rinzel's model of geniculate cells
  with low-threshold calcium spikes activated by hyperpolarization

  (1) Naming
  The neuron is based on the leaky integrate-and-fire neuron, but a
  low-threshold Ca++ current is included, leading to burst firing;
```

```
    see [1].

    (2) Inputts
    The neuron can recive inputt from current and spike event. The current
    event should be received in microA/cm^2. The spike event is recived
    through a synapse.

    (3) Synapses
     a) The neuron can only recive spike event through synapses. This is
        done by the SLI-command:
        Neuron1 Neuron2 SynapseType SynapseConnect -> port rport
     b) The SynapseType should be registered in the special synapsemodel
        dictionary and could be voltage dependent.
     c) The synapses could be lumping or not lumping. If the synapse is
        lumping, additional connection to this neuron with the same synapse,
        is made to the allready existing synapse. This accelerate the
        synapse handling.
     d) If the neuron is connected to a spiking source without the
        ordinary Connect command, a default synapse is created based on the
        default rport value of the SpikeEvent. If the default is 0 then a
        synapse with library entry 0 is created.
     e) The synapse inputt should be in mS/cm^2.

    (4) Time Stepping Schemes
    To handle the nonlineareties during threshold passes of V_th and V_h,
    the modified time-stepping schemes for  second order rungekutte, from [2]
    is implemented. This give an analytic precision of the same order as the
    global timestep.
    A forth order algorithm could also be implemented. This give an analytic
    precision of the third order of global timestep. This require that the
    synapses have precis spike handling and gives conductance information to
    the neuron in between the time steps.

    (5) Parameters
    Default parameters are from [1], Table 1; see GetStatus to find out
    about parameters; all params and V, h can be set.
    OBS! The total area of the neuron, (the area of the compartment) should
    be set to ensure the right synaptic inputt.
    If the neuron is "synapseconnected" to another neuron, the GetStatus
    command also give information about the synapse.

  References:
    [1] G. D. Smith et al, J Neurophysiology 83:588--610 (2000)
    [2] M. J. Shelley & L. Tao, J Computational Neuroscience 11:111-119 (2001)

  Author:  Johan Hake, Spring 2003

  SeeAlso:
   */

  namespace nest {

    using std::vector;
    using std::map;

    class Network;

    class lifb_neuron_rk4ts:
      public Node
    {

    public:

      typedef Node base;
```

```
    lifb_neuron_rk4ts();
    lifb_neuron_rk4ts(const lifb_neuron_rk4ts&);
    ~lifb_neuron_rk4ts();

    port    connect(SpikeEvent &);
    port    connect(Node *);

    void    handle(thread, SpikeEvent &);
    void    handle(thread, CurrentEvent &);
    void    handle(thread, PotentialRequest &);

    port connect_sender(SpikeEvent &);
    port connect_sender(CurrentEvent &);
    port connect_sender(PotentialRequest &) { return 0;}

    /**
     * Return current membrane potential.
     * This function is thread-safe and should be used in threaded
     * contexts to access the current membrane potential values.
     * @param steptime - the current network time
     *
     */
    double_t get_potential(steptime) const;

    /**
     * Define current membrane potential.
     * This function is thread-safe and should be used in threaded
     * contexts to change the current membrane potential value.
     * @param steptime  the current network time
     * @param float_t   new value of the mebrane potential
     *
     */
    void    set_potential(steptime, double_t);

protected:

    void init();
    void calibrate(realtime);
    void resize_buffers(delay);
    void update(thread, steptime, realtime);

    void get_properties(DictionaryDatum &) const;
    void set_properties(const DictionaryDatum &) ;

    std::string get_name(void) const ;

private:

    RingBuffer currents_0_;
    RingBuffer currents_1_;
    RingBuffer currents_2_;

    /** variables holding the currents */
    double_t c_0_;
    double_t c_1_;
    double_t c_2_;

    // neuron parameters
    // Voltages in mV, currents in pA, conductances in mS,
    //   capacitance in muF, times in ms
    double_t V_th_;         //!< threshold
    double_t V_res_;        //!< reset voltage
    double_t V_L_;          //!< leak current reversal potential (Cl-)
    double_t V_T_;          //!< T-current reversal potential (Ca++)
    double_t V_h_;          //!< T-current activation threshold
```

```
150     double_t g_L_;          //!< leak conductance
        double_t g_T_;          //!< T-current conductance

        double_t C_;            //!< capacitance

155     double_t tau_h_r_;      //!< time constant of T-rice
        double_t tau_h_d_;      //!< time constant of T-decay

        // state variables
        double_t V_0_;            //!< membrane potential in the begining of the timestep
160     double_t V_1_;            //!< membrane potential in the end of the timeste

        bool is_above_V_h_;                //!< Is true if V is above V_h

        /** inactivation level.
165      *  h==0: no T-current, h==1: max T-current
         */
        double_t h_0_;    //!< inactivation level in the begining of the time step
        double_t h_1_;    //!< inactivation level in the middle of the time step
        double_t h_2_;    //!< inactivation level in the end of the time step
170
        double_t idt_;         //!< internal time resolution

        // parrameter used to update the h variable
        double_t h_rise_;        //!< used when the potensial is below V_h
175     double_t h_decay_;       //!< used when the potensial is above V_h

        /** parrameters used in the Runge Kutte algorithm
         *  dV/dt = a*V + b
         *
180      *  X_0 the value in the begining of the timestep, used by the RK algorithm
         *  X_1 the value in the end of the timestep, used by the RK algorithm
                    */
        double_t a_0_;   //!< parrameter used by the RK algorithm
        double_t a_1_;   //!< parrameter used by the RK algorithm
185     double_t a_2_;   //!< parrameter used by the RK algorithm

        double_t b_0_;   //!< parrameter used by the RK algorithm
        double_t b_1_;   //!< parrameter used by the RK algorithm
        double_t b_2_;   //!< parrameter used by the RK algorithm
190
        double_t pot_[2];      //!< MT safe buffer for membrane potential.

        vector<Synapse*> sv_;  //! vector storing the different synapses, the index corresponds to the rport of the
        map<int_t,port> ls_;   //! map keeping track of the rports of the registered lumping synapses
195
        void update_h ( double_t t_th_,realtime dt );  //!< Updates the h variable after a threshpassing

        // Gives the next potensial value
        double_t rk4 (realtime dt);  //:<
200
        // Gives the potential after a threshpassing
        double_t potential_after_th( double_t V_thresh, double_t thresh_ratio , realtime dt);

         // Finds the time for threshhold passing.
205     double_t thresh_find( double_t V_thresh, realtime dt );

    };

    // Rungekutte 4:nd order algorithm
210 inline
    double_t lifb_neuron_rk4ts::rk4(realtime dt)
    {
      double_t k_1 = a_0_ * V_0_ + b_0_;
```

```
    double_t k_2 = a_1_ * ( V_0_ + dt * k_1 / 2 ) + b_1_;
    double_t k_3 = a_1_ * ( V_0_ + dt * k_2 / 2 ) + b_1_;
    double_t k_4 = a_2_ * ( V_0_ + dt * k_3 ) + b_2_;
    return  V_0_ + dt * ( k_1 + 2 * k_2 + 2 * k_3 + k_4 ) / 6;
  }

  inline
  double_t lifb_neuron_rk4ts::get_potential(steptime now) const
  {
    return pot_[now%2];
  }

  inline
  void lifb_neuron_rk4ts::set_potential(steptime now, double_t u)
  {
    pot_[(now+1)%2]=u;
  }

} // namespace

# endif
```

lifb neuron rk4ts.cpp

```
#include <cmath>
#include <map>

#include "exceptions.h"
#include "network.h"
#include "dict.h"
#include "integerdatum.h"
#include "doubledatum.h"
#include "dictutils.h"
#include "numerics.h"
#include "synapse.h"

#include "lifb_neuron_rk4ts.h"

nest::lifb_neuron_rk4ts::lifb_neuron_rk4ts()
  : Node(),
    currents_0_(network()->get_threads(),1),
    currents_1_(network()->get_threads(),1),
    currents_2_(network()->get_threads(),1),
    V_th_(-35.0),    // mV
    V_res_(-50.0),
    V_L_(-65.0),
    V_T_(120.0),
    V_h_(-60.0),
    g_L_(0.035),     // mS
    g_T_(0.07),
    C_(2.0),         // muF
    tau_h_r_(100.0),  // ms
    tau_h_d_(20.0),
    V_0_(V_L_),         // start at rest
    V_1_(V_0_),
    h_0_(1.0),    // The calsium current is de-inactivated
    h_1_(h_0_),
    h_2_(h_0_)
{
  init();
}

nest::lifb_neuron_rk4ts::lifb_neuron_rk4ts(const nest::lifb_neuron_rk4ts &n)
  : Node(n),
    currents_0_(network()->get_threads(),1),
```

```
    currents_1_(network()->get_threads(),1),
    currents_2_(network()->get_threads(),1),
    V_th_(n.V_th_),    // mV
    V_res_(n.V_res_),
    V_L_(n.V_L_),
    V_T_(n.V_T_),
    V_h_(n.V_h_),
    g_L_(n.g_L_),     // mS
    g_T_(n.g_T_),
    C_(n.C_),         // muF
    tau_h_r_(n.tau_h_r_),  // ms
    tau_h_d_(n.tau_h_d_),
    V_0_(n.V_0_),
    V_1_(n.V_1_),
    h_0_(n.h_0_),
    h_1_(n.h_1_),
    h_2_(n.h_1_)
{
  init();
}

nest::lifb_neuron_rk4ts::~lifb_neuron_rk4ts()
{
  for(vector<Synapse *>::iterator i=sv_.begin();
      i != sv_.end(); ++i)
    {
      delete *i;
    }
  sv_.clear();

}

std::string nest::lifb_neuron_rk4ts::get_name(void) const
{
  return std::string("lifb_neuron_rk4ts");
}

void nest::lifb_neuron_rk4ts::resize_buffers(delay d)
{
  currents_0_.set_delay(d);
  currents_1_.set_delay(d);
  currents_2_.set_delay(d);
  for (uint_t i = 0; i < sv_.size(); ++i)
  {
    sv_[i]->set_delay(d);
  }
}

void nest::lifb_neuron_rk4ts::init()
{
  c_0_ = c_1_ = c_2_ = 0.0;
  V_0_ = V_1_;
  h_0_ = h_1_ = h_2_;
  is_above_V_h_ = V_0_ >= V_h_;
  a_0_ = a_1_ = a_2_ = -( g_L_ + ( is_above_V_h_ ? h_1_ : 0.0 ) * g_T_ );
  b_0_ = b_1_ = b_2_ = g_L_ * V_L_ + ( is_above_V_h_ ? h_1_ : 0.0 ) * g_T_ * V_T_;

  for (uint_t i = 0; i < sv_.size(); ++i)
  {
    // *(sv_[i]->get_g()+1) is used to get the second value in the
    // conductance array.
    a_2_  -= *(sv_[i]->get_g()+1);
    b_2_  += *(sv_[i]->get_g()+1) * sv_[i]->rev_pot();
  }
```

```
  realtime dt = network()->get_resolution();
  h_rise_  = std::exp( -dt / tau_h_r_ );
  h_decay_ = std::exp( -dt / ( 2 * tau_h_d_) );

  a_2_ /= C_;
  b_2_ /= C_;
  pot_[0]= pot_[1] = V_1_;
}

void nest::lifb_neuron_rk4ts::calibrate(realtime dt)
{
  h_rise_ = std::exp( -dt / tau_h_r_ );
  h_decay_ = std::exp( -dt / ( 2 * tau_h_d_) );
  for (uint_t i = 0; i < sv_.size(); ++i)
  {
    sv_[i]->calibrate(dt);
  }
}


void nest::lifb_neuron_rk4ts::update(thread t, steptime T, realtime dt)
{
  // Shifting variables to the next timestep
  V_0_ = V_1_;
  h_0_ = h_2_;
  a_0_ = a_2_;
  b_0_ = b_2_;

  // Collecting the currents in the first part of the time step
  c_0_ = currents_0_.collect(T);

  // Is the current in the last part of the previous time step
  // different from the first part of this? (Has a current been turned on/off?)
  // Then use the "new" current instead of the "old"
  if (c_0_ != c_2_)
  {
    b_0_ -= c_2_/ C_;
    b_0_ += c_0_/ C_;
  }
  // Collecting the currents in the second part of the time step
  c_1_  =currents_1_.collect(T);
  c_2_  =currents_2_.collect(T);

  // Updating the h variable diffrent if V is above V_h or not
  if (is_above_V_h_)
  {
    h_1_ = h_decay_ * h_0_;
    h_2_ = h_decay_ * h_1_;

    a_1_ = -( g_L_ +  h_1_ * g_T_ );
    b_1_ = g_L_ * V_L_ + h_1_ * g_T_ * V_T_ + c_1_;

    a_2_ = -( g_L_ +  h_2_ * g_T_ );
    b_2_ = g_L_ * V_L_ + h_2_ * g_T_ * V_T_ + c_2_;
  }
  else
  {
    h_2_ = 1.0 - h_rise_ * ( 1.0 - h_0_);

    a_1_ = a_2_ = - g_L_ ;
    b_1_ = g_L_ * V_L_ + c_1_;
    b_2_ = g_L_ * V_L_ + c_2_;
  }
```

```
// Updates the sum of the inputts a and b

// Collects the a and b inputts
for (uint_t i = 0; i < sv_.size(); ++i)
{
  sv_[i]->update(T);
  // get_g gives a pointer to an array of size two, *(sv_[i]->get_g()) gets
  // the first value and  *(sv_[i]->get_g()+1) gets the second value
  a_1_  -= *(sv_[i]->get_g());
  b_1_  += *(sv_[i]->get_g()) * sv_[i]->rev_pot();
  a_2_  -= *(sv_[i]->get_g()+1);
  b_2_  += *(sv_[i]->get_g()+1) * sv_[i]->rev_pot();
  //    if (get_lid() == 70 && i == 20)
  //      std::cerr << *(sv_[i]->get_g()) << ", " << std::flush;
}

// Divide by the conductanse
a_1_ /= C_;
b_1_ /= C_;

a_2_ /= C_;
b_2_ /= C_;

// gets the next potensial value
V_1_ = rk4(dt);

// threshold
if ( V_1_ >= V_th_ )
{
  // The time for the threshpassing
  double_t t_th = thresh_find( V_th_, dt );

  // If the resetpotensial is below the threshhold for the T-current
  if ( V_h_ > V_res_ )
    update_h( t_th, dt );

  // Getting the potensial after the threshpassing
  V_1_ = potential_after_th( V_res_, t_th / dt, dt );

              // Create the spike event to send
  SpikeEvent sp;

  sp.set_time( T*dt + t_th );   // include precise time
  network()->send_to_targets(t,this, T, sp );  // produce a spike
}

// Passing threshhold for the T-current?
if ( ( V_1_ >= V_h_ && !is_above_V_h_ ) ||( V_1_ < V_h_ && is_above_V_h_ ) )
{
  // The time for the threshpassing
  double_t t_th = thresh_find( V_h_, dt );

  // Swiching the flag
  is_above_V_h_ = !is_above_V_h_;

  // Updating the h variable
  update_h( t_th, dt );

  // Getting the potensial after the threshpassing
  V_1_ = potential_after_th( V_h_, t_th / dt , dt);

}

// Update "export potential"
set_potential(T,V_1_);
```

```
}

void nest::lifb_neuron_rk4ts::update_h( const double_t t_th, realtime dt )
{
  if ( is_above_V_h_ )
  {
    /** Calculating the new values of th inactivation variable as if
     *  the potensial was above the threshhold V_h
     */
    const double h_th =  1.0 - std::exp( -t_th / tau_h_r_ ) * ( 1 - h_0_);
    h_0_ = std::exp(  t_th / tau_h_d_ ) * h_th;
    h_1_ = h_decay_ * h_0_;
    h_2_ = h_decay_ * h_1_;

    // Updating the a and b parrameters used to calculate the potensial after a threshpass
    a_0_ += -h_0_ * g_T_ / C_;
    a_1_ += -h_1_ * g_T_ / C_;
    a_2_ += -h_2_ * g_T_ / C_;

    b_0_ += h_0_ * g_T_ * V_T_ / C_;
    b_1_ += h_1_ * g_T_ * V_T_ / C_;
    b_2_ += h_2_ * g_T_ * V_T_ / C_;
  }
  else
  {
    // substracting the contribution of the inactivating variable to the a and b values.
    a_0_ -= -h_0_ * g_T_ / C_;
    a_1_ -= -h_1_ * g_T_ / C_;
    a_2_ -= -h_2_ * g_T_ / C_;

    b_0_ -= h_0_ * g_T_ * V_T_ / C_;
    b_1_ -= h_1_ * g_T_ * V_T_ / C_;
    b_2_ -= h_2_ * g_T_ * V_T_ / C_;

        /** Calculating the new values of th inactivation variable as if
     *  the potensial was below the threshhold V_h
     */
    const double h_th =  std::exp( -t_th / tau_h_d_ ) * h_0_;
    h_2_ = 1.0 - std::exp( -( dt - t_th) / tau_h_r_ ) * ( 1 - h_th);
  }

}
  // Finds the time for threshhold passing.
nest::double_t nest::lifb_neuron_rk4ts::thresh_find( double_t V_thresh, realtime dt )
{
  // Setting the derivativs of the begining and end of the timestep
  // The last koefisient without / dt^3 (for numerical reasons)
  double_t dVdt_0 = a_0_ * V_0_ + b_0_;
  double_t dVdt_1 = a_2_ * V_1_ + b_2_;

  // The interpolation polynom
  double_t koef_0 = V_0_ - V_thresh;
  double_t koef_1 = dVdt_0;
  double_t koef_2 = ( 3 * ( V_1_ - V_0_ ) - dt * ( 2 * dVdt_0 + dVdt_1 ) ) / pow( dt, 2 );
  double_t koef_3 = ( -2 * ( V_1_ - V_0_ ) + dt * ( dVdt_0 + dVdt_1 ) );

  // Newtons method
  bool find = false;
  double_t toleranse =  dt * 1e-12;
  double_t t_help_0 = ( V_thresh - V_0_ ) * dt / ( V_1_ - V_0_ );
  double_t t_help_1;
  int_t    count = 0;
  while (!find && count < 20)
  {
    t_help_1 = t_help_0 - ( koef_0 + koef_1 * t_help_0 + koef_2 * pow( t_help_0, 2 ) + koef_3 * pow( t_help_0 / dt, 3
```

```
    if ( fabs( t_help_0 - t_help_1 ) < toleranse)
      find = true;
    else
      t_help_0 = t_help_1;
    count++;
  }
  return t_help_0;
}

nest::double_t nest::lifb_neuron_rk4ts::potential_after_th( double_t V_thresh, double_t thresh_ratio, realtime
{
  // Helping variables used to calculate the A and B parrameter for the time step when a threshhold is passed.
  double_t A_help_0 = a_0_;
  double_t A_help_1 = a_1_ * ( 1 + dt * A_help_0 / 2 );
  double_t A_help_2 = a_1_ * ( 1 + dt * A_help_1 / 2 );
  double_t A_help_3 = a_2_ * ( 1 + dt * A_help_2 );

  double_t B_help_0 = b_0_;
  double_t B_help_1 = b_1_ + dt * B_help_0 * a_1_ / 2;
  double_t B_help_2 = b_1_ + dt * B_help_1 * a_1_ / 2;
  double_t B_help_3 = b_2_ + dt * B_help_2 * a_2_;

  // The A and B parrameters calculated from the rk4 algorithm
  // V_1 = A * V_0 + B
  double_t A = 1 + dt * ( A_help_0 + 2 * A_help_1 + 2 * A_help_2 + A_help_3 ) / 6;
  double_t B = dt * ( B_help_0 + 2 * B_help_1 + 2 * B_help_2 + B_help_3 ) / 6;

  // Helping variables used insted of calling the function pow()
  double_t thresh_ratio_2 = pow( thresh_ratio, 2 );
  double_t thresh_ratio_3 = pow( thresh_ratio, 3 );

  // Helping variables from the interpolationpolynom.
  // V_thresh = K_0 * V_0 + K_1 * V_1 + K_2
  double_t K_0 = 1 + a_0_ * thresh_ratio * dt - thresh_ratio_2 * ( 2 * dt * a_0_ + 3 ) + thresh_ratio_3 * ( dt
  double_t K_1 = thresh_ratio_2 * ( 3 - dt * a_2_ ) + thresh_ratio_3 * ( dt * a_2_ - 2 );
  double_t K_2 = b_0_ * thresh_ratio * dt - thresh_ratio_2 * dt * ( 2 * b_0_ + b_2_ ) + thresh_ratio_3 * dt * (

  // Returning the value solved for V_1_
  return  ( ( V_thresh - K_2 ) * A + B * K_0 ) / ( K_0 + A * K_1 );

}


void nest::lifb_neuron_rk4ts::get_properties(DictionaryDatum &d) const
{
  def<double_t>(d, "V_thresh", V_th_);
  def<double_t>(d, "V_reset", V_res_);
  def<double_t>(d, "V_leak", V_L_);
  def<double_t>(d, "V_T", V_T_);
  def<double_t>(d, "V_h", V_h_);
  def<double_t>(d, "g_leak", g_L_);
  def<double_t>(d, "g_T", g_T_);
  def<double_t>(d, "C", C_);
  def<double_t>(d, "tau_h_rise", tau_h_r_);
  def<double_t>(d, "tau_h_decay", tau_h_d_);
  def<double_t>(d, "V", V_0_);
  def<double_t>(d, "h", h_0_);

  for (uint_t i=0; i < sv_.size();++i )
  {
    sv_[i]->get_properties(d);
  }

```

```
}

// Todo: When an attribute is changed it should have consequenses for the rest... like a and b par.
void nest::lifb_neuron_rk4ts::set_properties(const DictionaryDatum &d)
{
  updateValue<double_t>(d, "V_thresh", V_th_);
  updateValue<double_t>(d, "V_reset", V_res_);
  updateValue<double_t>(d, "V_leak", V_L_);
  updateValue<double_t>(d, "V_T", V_T_);
  updateValue<double_t>(d, "V_h", V_h_);
  updateValue<double_t>(d, "g_leak", g_L_);
  updateValue<double_t>(d, "g_T", g_T_);
  updateValue<double_t>(d, "C", C_);
  updateValue<double_t>(d, "tau_h_rise", tau_h_r_);
  updateValue<double_t>(d, "tau_h_decay", tau_h_d_);
  updateValue<double_t>(d, "V", V_1_);
  updateValue<double_t>(d, "h", h_2_);

  for (uint_t i=0; i < sv_.size();++i )
  {
    sv_[i]->set_properties(d);
  }

  calibrate(network()->get_resolution());
  init();
}

nest::port nest::lifb_neuron_rk4ts::connect_sender(SpikeEvent &e)
{
  /*  Connects the neuron with a spiking sender through a chosen
   *  synapse. The information about which synapse the connection
   *  is made with, is stored in the rport of the event.
   *  If the synapse allready excist and it is lumping then the
   *  function return the index of synapse in the synapsevector.
   *  If not, a synapse is created and put in the synapsevector
   *  and the index is returned as the rport value of the connection.
   */

  int_t syn_type = e.get_rport();
  map<int_t,port>::iterator it = ls_.find(syn_type);

  // Is the synapse registerd as a lumping synapse?
  if ( it != ls_.end() )
  {
    // Get the rport for the lumping synapse
    port rp = it->second;

    // Set the delay for the connection
    sv_[rp]->set_delay(e.get_delay());

    // return the rport value of the synapse
    return rp;
  }
  else
  {
    // Creates a synapse by using the requested model from the network
    SynapseModel *synapse_model = network()->get_synapse_model(syn_type);
    Synapse *new_synapse = synapse_model->allocate();

    // The index in the synapsevector is used as the rport of the connection
    port rp = sv_.size();
    sv_.push_back(new_synapse);

    // Check the delay with the synapse
    new_synapse->set_delay(e.get_delay());
```

```
    // if the synapse is lumping, register it.
    if (new_synapse->is_lumping())
      ls_[syn_type]=rp;

    return rp;
  }
}

nest::port nest::lifb_neuron_rk4ts::connect_sender(CurrentEvent &e)
{
  // We are the target neuron and
  // check the delay with our input buffer.
  currents_1_.set_delay(e.get_delay());
  currents_2_.set_delay(e.get_delay());
  // return the default rport=0
  return 0;
}

nest::port nest::lifb_neuron_rk4ts::connect(SpikeEvent &e)
{
  return register_connection(e);
}

nest::port nest::lifb_neuron_rk4ts::connect(Node *target)
{
  assert(target !=NULL);
  SpikeEvent se;
  return se.connect(*this, *target);
}

void nest::lifb_neuron_rk4ts::handle(thread p, SpikeEvent & e)
{
  // we assume that the time stamp of the event contains
  // the absolute time of arrival.

  sv_[e.get_rport()]->register_spike( p, e);
}

void nest::lifb_neuron_rk4ts::handle(thread p, CurrentEvent& e)
{
  // we assume that the time stamp of the event contains
  // the absolute time of arrival.

  const double_t c_0=*(e.get_current_pointer());
  const double_t c_1=*(e.get_current_pointer()+1);
  const double_t c_2=*(e.get_current_pointer()+2);
  const double_t w=e.get_weight();

  currents_0_.add_value(p, e.get_stamp(), w*c_0);
  currents_1_.add_value(p, e.get_stamp(), w*c_1);
  currents_2_.add_value(p, e.get_stamp(), w*c_2);
}

void nest::lifb_neuron_rk4ts::handle(thread p, PotentialRequest& e)
{
  /*
   * If we receive a potential request, we construct a
   * potential event and send it immediately to the requestor.
   */
  PotentialEvent pe;

  pe.set_sender(*this);
  pe.set_receiver(e.get_sender());
  pe.set_port(e.get_port());
```

```
    // Here, we have a +/- one problem, depending on
    // whether the node is already updated or not.
500 // node::get_time() takes care of this problem.
    pe.set_stamp(get_time()+1);

    // pe.delay defaults to 1, so we need not set it.

505 pe.set_potential(get_potential(network()->get_time()));
    network()->send(p,pe);
  }
```

C.3 Abstract synapse class

Here follows the code for the abstract synapse class which has to be inherit by any implemented synapse model. synapse.h also include the SynapseModel class and the GenericSynapseModel class that are used to create synapse objects during runtime.

synapse.h

```
1   #ifndef SYNAPSE_H
    #define SYNAPSE_H

    #include "instance.h"
5   #include "node.h"
    #include "event.h"
    #include <string>
    #include "ring_buffer.h"

10  namespace nest {

      class Synapse;

      class SynapseModel
15    {
      public:
        /**
         * Default constructor.
         * class SynapseModel only has one constructor.
20       * @param char[] C-style string with the name of the synapsemodel.
         *
         * The name should be a single word, uniquely defining the model.
         * Ideally, this name should be identical to the name, returned by
         * the associated Synapse class.
25       */
        SynapseModel(const char []);
        virtual ~SynapseModel(){};

        /**
30       * Allocate new Synapse and return its pointer.
         * allocate() is not const, because it
         * is allowed to modify the Model object for
         * 'administrative' purposes.
         */
35      virtual Synapse* allocate(void) =0;

        /**
         * Reserve memory for n Synapse.
         * A number of memory managers work more efficently, if they have
```

```
     * an idea about the number of Nodes to be allocated.
     * This function prepares the memory manager for the subsequent
     * allocation of n Synapse.
     * @param n Number of Synapse to be allocated.
     */
    virtual void reserve(size_t n)=0;

    /**
     * Initialise synapse model-wide client data.
     * Synapse model objects can be used to store data which is shared by a
     * larger group of Nodes similar to static members.
     * This function can be used to initialize such client data. The
     * default implementation is empty.
     * @see calibrate
     */
    virtual void init(){};

    /**
     * Notify SynapseModel of a change of the temporal resolution.
     * Model objects can be used to store data which is shared by a
     * larger group of Synapse similar to static members.
     * Such client data may depend on the temporal resolution of the
     * system. Thus, this function can be used to adjust internal
     * parameters to a change in resolution.
     * @param realtime New value of the temporal resolution.
     */
    virtual void calibrate(realtime){};   //!< re-calibrate

    /**
     * Return name of the SynapseModel.
     * This function returns the name of the SynapseModel as C++ string. The
     * name was defined through the constructor.
     * @see SynapseModel::SynapseModel
     */
    const std::string& get_name() const;

protected:
    /**
     * Holds the name of the synapse model.
     * @internal
     * @see get_name
     * @see SynapseModel::SynapseModel
     */
    const std::string name_;
};

inline
SynapseModel::SynapseModel(const char n[])
  :name_(n)
{}

inline
const std::string& SynapseModel::get_name() const
{
  return name_;
}

/**
 * Generic SynapseModel template.
 * The template GenericSynapseModel should be used
 * as base class for custom model classes. It already includes the
 * element factory functionality, as well as a pool bases memory
 * manager, so that the user can concentrate on the "real" model
 * aspects.
 * @ingroup user_interface
```

```
105     */
        template <typename ElementT>
        class GenericSynapseModel: public SynapseModel
        {

110     public:
          GenericSynapseModel(const char[]);

          Synapse* allocate(void);
          void  reserve(size_t s);
115     };

        template< typename ElementT>
        inline
        GenericSynapseModel<ElementT>::GenericSynapseModel(const char s[])
120       : SynapseModel(s)
        {}

        template< typename ElementT>
        inline
125     Synapse* GenericSynapseModel<ElementT>::allocate(void)
        {
          return new Instance<ElementT>;
        }

130     template< typename ElementT>
        inline
        void GenericSynapseModel<ElementT>::reserve(size_t s)
        {
          Instance<ElementT>::reserve(s);
135     }

      /*BeginDocumentation
      Name: Synapse

140   Description:

        The base class for all synapse objects.

        Every child-synapse has to implement its own dynamics. The only parrameters
145     that are common for all synapses are the reversal potential, maximal
        conductance, a spike buffer and a variable saying if the synapse is lumping
        or not.

        The synapse could be lumping or not lumping. If a synapse is lumping and a
150     connection is registered throught this, all further connection through the
        same type of synapse is registered through the first one. This accelerates
        the spike handling and the uppdate mechanism.

        Comment:
155     As for now, only synapse with linear conductanse and where spikes
        are registered with addition of a constant value could be lumping. The last
        criteria disables the synapse to saturated when many spikes come in from
        one neuron through the same synapse.

160     This could  be omitted by implementeing a way of knowing which
        neuron fired the spike (maybe through a port system) and by keeping
        record of when that neuron last time fired a spike. This together with
        a record of the "r" variable, the fraction of open receptors, of each
        synapse will do the cake. This could be easy implemented with a central
165     management of spikeevent.

        If the spike handling is manage localy, a modification has to be done. The
        present ringbuffer has two drawbacks. 1) It "sort" the spike dependent on
        thread, not by sending neuron. 2) The collect() function of the ringbuffer
```

```
    just sum upp the incomming weighted spikes in that time step, our
    implementation have to deal with every spike independently.


  Author: Johan Hake, Spring 2003
  */

    class Synapse {

    public:

      Synapse();
      Synapse(const Synapse &);

      virtual ~Synapse();

      // Register a spike in the ringbuffer
      virtual void register_spike(thread p, SpikeEvent & e);

      // Updates the synaps
      virtual void update(steptime T)=0;

      virtual void set_properties(const DictionaryDatum&)=0;

      virtual void get_properties(DictionaryDatum&) const=0;

      virtual void calibrate(realtime)=0;

      // Return a pointer to double so it could return an array ov conductance
      virtual double_t* get_g()=0;

      double_t rev_pot() const;

      virtual std::string get_name() const=0;

      bool is_lumping() const ;

      realtime get_resolution();

      Network* network();

      ulong_t get_now() const;

      virtual void set_delay(delay d);

    protected:

      RingBuffer spikes_;
      bool is_lumping_;

      double_t rev_pot_;
      double_t g_max_;


    };

    inline
    void Synapse::register_spike(thread p, SpikeEvent & e)
    {
      // we assume that the time stamp of the event contains
      // the absolute time of arrival.
      spikes_.add_value(p, e.get_stamp(), e.get_weight());
    }

    inline
```

```
235   bool Synapse::is_lumping() const
      {
        return is_lumping_;
      }

240   inline
      double_t Synapse::rev_pot() const
      {
        return rev_pot_;
      }
245 }  //Namespace


    #endif
```

synapse.cpp

```
 1  #include "synapse.h"
    #include "network.h"
    #include "node.h"

 5  namespace nest {
      Synapse::Synapse()
        : spikes_(network()->get_threads(),1),
          is_lumping_(true),
          rev_pot_(0.0),
10        g_max_(0.0)
      {
      }

      Synapse::Synapse(const Synapse &s)
15      :spikes_(network()->get_threads(),1),
          is_lumping_(s.is_lumping_),
          rev_pot_(s.rev_pot_),
          g_max_(s.g_max_)
      {
20    }

      Synapse::~Synapse()
      {
      }
25
      realtime Synapse::get_resolution()
      {
        return Node::network()->get_resolution();
      }
30
      ulong_t Synapse::get_now() const
      {
        return Node::network()->get_time();
      }
35
      Network* Synapse::network()
      {
        return Node::network();
      }
40
      std::string Synapse::get_name() const
      {
        return std::string("Synapse");
      }
45
      void Synapse::set_delay(delay d)
      {
        spikes_.set_delay(d);
```

```
    }
50
}
```

C.4 $GABA_A$ synapse

The most complex implementation of the $GABA_A$ synapse is presented here. This model register a presynaptic spike at the time it actually arrives the synapse. Therefore the *ps* (precise spike timing), in the name. It provides the conductance from two times in the time steps, in the middle and in the end, therefore the *2* in the name. It can therefore only be used by neuron models that use conductances in the middle of a time step, for example model integrated with RK4.

gabaasynapse 2ps.h

```
1   #ifndef GABAASYNAPSE_2PS_H
    #define GABAASYNAPSE_2PS_H

    #include "nest.h"
5   #include "event.h"
    #include "synapse.h"
    #include "spike_event_buffer.h"

    /* BeginDocumentation
10  Name: GabaASynapse_2ps
    Description:

      The GabaAsynapse is a ionotropic exitatory synapse.

15    The GabaAsynapse implements a modified model of Destexhe's ampasynapse
      model from [1], chapter 4. The synaptic conductanse is given in nS.

      The synaptic conductance is based on a beta function. This is a simplyfied
      model compared to [1], chapter 4. The parameters that determine the chape of
20    the  total conductance caused by a singel spike, the two time constants
      tau_d_ and tau_r_ and the maximum conductance g_max_ are fitted to data
      from [1] correnspondingto a singel spike.

      The conductance are updated with exact integration scheme, presented in
25    [2]. A spike is registered by adding a constant value, weighted by the weight
      of the connection, to the first state variable of the beta function, g_0_.

    References:
      [1] A. Destexhe et al, Kinetic Models of Synaptic Transmission,
30        Methods in Neuronal Modeling (2nd ed.) MIT Press, (1998)
      [2] S. Rotter and M. Diesmann, Exact digital simulation of time-
          invariant linear systems with applications to neuronal modeling,
          Biol. Cybern. 81, 381-402 (1999)

35  Author:  Johan Hake, Spring 2003
     */

    namespace nest {

40    class GabaASynapse_2ps:
        public Synapse
      {
```

```
  public:

    GabaASynapse_2ps();
    GabaASynapse_2ps(const GabaASynapse_2ps&);

    void update(steptime T);

    void set_properties(const DictionaryDatum&);

    void get_properties(DictionaryDatum&) const;

    void register_spike(thread , SpikeEvent & e);

    void calibrate(realtime);

    double_t fast_exp(double_t x);

    // Returns the conductance
    double_t* get_g();

    // Overload set_delay
    void set_delay(delay){};

    std::string get_name() const;

  private:

    SpikeEventBuffer spike_events_;

    // The timeconstants for the beta function. tau_r_ is for the rise and
    // tau_d_ is for the decay.
    double_t tau_r_;
    double_t tau_d_;

    // State variables for the conductance. g_1_ holds the total conductance
    double_t g_0_;
    double_t g_1_[2];
    double_t g_out_[2];

    // A constant value first weighted by weight to the connection and
    // then added to g_0_ when ever a spike arrives at the synapse
    double_t spike_add_;

    // The update matrix
    double_t exp_A_00_;
    double_t exp_A_10_;
    double_t exp_A_11_;


  };

  inline
  double_t* GabaASynapse_2ps::get_g()
  {
    return g_1_;
  }

  inline
  void GabaASynapse_2ps::register_spike(thread p, SpikeEvent & e)
  {
    realtime sat; // Spike Arival Time
    if (! (sat = e.get_time() ) > 0.0)
    {
      sat =  e.get_stamp() * get_resolution();
    }else
```

```
    {
      sat += e.get_delay() * get_resolution();
    }

    spike_events_.add_spike(p, sat , e.get_weight() );
  }

  inline
  double_t GabaASynapse_2ps::fast_exp(double_t x)
  {
    const double_t x_2 = x*x;
    return 1 + x + x_2/2 + x_2 * x/6;
  }


  inline
  std::string GabaASynapse_2ps::get_name() const
  {
    return std::string("GabaASynapse_2ps");
  }

}// namespace
#endif
```

gabaasynapse 2ps.cpp

```
#include "dict.h"
#include "integerdatum.h"
#include "doubledatum.h"
#include "dictutils.h"
#include "numerics.h"
#include "network.h"

#include "gabaasynapse_2ps.h"
#include <cmath>

nest::GabaASynapse_2ps::GabaASynapse_2ps( )
  : Synapse(),
    spike_events_(network()->get_threads()),
    tau_r_(0.27),    // ms calculated to fitt [1]
    tau_d_(5.5),     // ms calculated to fitt [1]
    g_0_(0.0)
{
  g_out_[0] = g_out_[1] = 0.0;
  g_1_[0] = g_1_[1] = 0.0;
  is_lumping_ = true;
  rev_pot_ = -80.0;     // mV from [1]

  /** g_max_ is the maximum conductance value produced by one spike. This is
      not the same value as the G_max in [1].
      In [1] G_max is multiplied by a value r, which is the fraction of open
      receptors in the synapse. If r_max is the maximum value of open receptors
      caused by a singel spike, then our g_max_is given by G_max * r_max.
      From [1] we fitted r_max to be 0.93 and from [1] we have that G_max
      is between 0.25 and 1.2.
      This give us a g_max_, caused by a singel spike, to be 0.23 < g_max_< 1.1.
   */
  g_max_ = 0.5;    //nS
  calibrate(get_resolution());
}

nest::GabaASynapse_2ps::GabaASynapse_2ps(const GabaASynapse_2ps &as)
  : Synapse(as),
    spike_events_(network()->get_threads()),
```

```
    tau_r_(as.tau_r_),     // ms
    tau_d_(as.tau_d_),
    g_0_(as.g_0_)
{
  g_1_[0]=as.g_1_[0];
  g_1_[1]=as.g_1_[1];
  calibrate(get_resolution());
}

void nest::GabaASynapse_2ps::update(steptime T)
{
  realtime now_plus_dt = (T+1)*get_resolution();

  // Any spikes in this time step?
  if ( spike_events_.any_spike( now_plus_dt ) )
  {
    double_t g_0_sum_add[] = {0.0, 0.0};
    double_t g_1_sum_add[] = {0.0, 0.0};
    // Get the spikes from the buffer
    std::vector< std::pair< realtime, weight> > sp(spike_events_.collect( now_plus_dt ));
    realtime dt = get_resolution();
    for (std::vector<std::pair<realtime,weight> >::iterator it = sp.begin();
         it < sp.end(); ++it)
    { //                          now_plus_dt
      //                              |
      //  spike time                  v
      //    n  |       n+0.5         n+1
      //    |__v_________|____________|
      //       <-rest_t-> <---dt/2--->
      //     <----------dt----------->

      // Has the spike come in the first part of the time step?
      realtime rest_t = now_plus_dt - it->first;
      uint_t part;
      if (rest_t > dt/2)
      {
        part = 0;
        rest_t -= dt/2;
      }
      else
      {
        part = 1;
      }

      const double_t exp_A_00_temp = fast_exp(-rest_t/tau_d_);
      const double_t exp_A_10_temp = (exp_A_00_temp - fast_exp(-rest_t/tau_r_) ) / ( 1/tau_r_ - 1/tau_d_ );

      g_1_sum_add[part] += exp_A_10_temp * spike_add_ * it->second;
      g_0_sum_add[part] += exp_A_00_temp * spike_add_ * it->second;
    }

    // Calculate the new values of the conductance
    g_1_[0] = exp_A_10_ * g_0_ + exp_A_11_ * g_1_[1];
    g_0_ = exp_A_00_ * g_0_;

    g_1_[0] += g_1_sum_add[0];
    g_0_    += g_0_sum_add[0];

    g_1_[1] = exp_A_10_ * g_0_ + exp_A_11_ * g_1_[0];
    g_0_ = exp_A_00_ * g_0_;

    g_1_[1] += g_1_sum_add[1];
    g_0_    += g_0_sum_add[1];
  }
  else
```

```
    {
      // Calculate the new values of the conductance
      g_1_[0] = exp_A_10_ * g_0_ + exp_A_11_ * g_1_[1];
      g_0_ = exp_A_00_ * g_0_;

      g_1_[1] = exp_A_10_ * g_0_ + exp_A_11_ * g_1_[0];
      g_0_ = exp_A_00_ * g_0_;
    }
    //g_out_[0] = std::pow(g_1_[0],3)*g_max_;
    //g_out_[1] = std::pow(g_1_[1],3)*g_max_;
  }


  void nest::GabaASynapse_2ps::get_properties(DictionaryDatum &d) const
  {
    def<double_t>(d, "GabaA_tau_rise", tau_r_);
    def<double_t>(d, "GabaA_tau_decay", tau_d_);
    def<double_t>(d, "GabaA_reversal_potensial", rev_pot_);
    def<double_t>(d, "GabaA_g_max", g_max_);
    def<double_t>(d, "GabaA_g_0", g_0_);
    def<double_t>(d, "GabaA_g_1", g_1_[1]);
  }

  void nest::GabaASynapse_2ps::set_properties(const DictionaryDatum &d)
  {
    updateValue<double_t>(d, "GabaA_tau_rise", tau_r_);
    updateValue<double_t>(d, "GabaA_tau_decay", tau_d_);
    updateValue<double_t>(d, "GabaA_reversal_potensial", rev_pot_);
    updateValue<double_t>(d, "GabaA_g_max", g_max_);
    updateValue<double_t>(d, "GabaA_g_0", g_0_);
    updateValue<double_t>(d, "GabaA_g_1", g_1_[1]);
    calibrate(get_resolution());
  }

  void nest::GabaASynapse_2ps::calibrate( realtime dt )
  {
     // Two step in every time step.
    dt /=2;
    // Sets the maximum of the "base" betafunction to be one.
    spike_add_ = std::pow( tau_r_ / tau_d_, tau_r_ / ( tau_r_ - tau_d_ ) ) / tau_r_;

    // Sets the maximum of the conductance for a singel spike to be g_max_
    //spike_add_ *= g_max_;

    // The update matrix.
    exp_A_00_ = std::exp( -dt / tau_d_ );
    exp_A_10_ = ( std::exp( -dt / tau_d_ ) - std::exp( -dt / tau_r_ ) ) / ( 1/tau_r_ - 1/tau_d_ );
    exp_A_11_ = std::exp( -dt / tau_r_ );
  }
```

C.5 $GABA_B$ synapse

Here is the only implementation of the $GABA_B$ synapse presented. It does not use neither precise spike timing nor does it provides conductances in the middle of a time step. It can thus only be used by RK2 methods.

gababsynapse.h

```
#ifndef GABABSYNAPSE_H
#define GABABSYNAPSE_H
```

```
#include <cmath>

#include "nest.h"
#include "event.h"
#include "synapse.h"

/* BeginDocumentation
Name: GabaBSynapse
Description:

  The GabaBsynapse is a metabotropic, inhibitory synapse.

  The GabaBsynapse implements a modified model of Destexhe's GabaBsynapse
  model from [1], chapter 4. The synaptic conductanse is given in nS and have
  to be adjusted to fitt the model that recive the synaptic current.

  The original model from [1] and my modified model

  Original                 |   Modified
  -------------------------------------------------------
  dr                       |   dr*
  -- = K_1*T(1-r) - K_2*r  |   -- =  - K_2 r*
  dt                       |   dt
                           |
  ds                       |   ds
  -- = K_3r - K_4 s        |   -- = r - K_4 s
  dt                       |   dt

                      s^n
  I_GabaB = g_max * --------- (V - V_rev) (no changes)
                  s^n + K_d
  -------------------------------------------------------

  Model explanation
  r   : Fraction of activated receptors.
  s   : Concentration of G-proteins binded to K+ channels
  n   : Number of individual binding sites for the G-protein in  a singel
        K+ channel
  K_1 : Binding rate for transmitter substance in the cleft to receptors
  K_2 : Unbinding rate for the transmitter substance to the receptors
  K_3 : Binding rate for the G-proteins to the K+ channel
  K_4 : Unbinding rate for the G-proteins to the K+ channel
  K_d : Dissociation constant of binding G-protein to K+ channels
  T   : Consentration of transmitter substance in the cleft. They expose the
        postsynaptic side during a pulse of 1 ms. T is zero when there are
        no spike and T during 1 ms after a spike.

  The following difference are made from [1]:
  1) The equation is modified so it fitts the exact integration scheme from [2].
     The r variable is scaled by the parrameter K3, given r*.
  2) Instead of letting r rise during the time the postsynaptic side is exposed
     to transmitter substance, a singel value are added to the variabel r ones.
     This value depend upon the fraction of allready activated receptors, r,
     the maximum value of r, r_inf, the pulse length, pulse, the time constant
     for r when rising, tau_r_add.

     r_add = (r_inf - r)*(1 -  exp(-pulse / tau_r_add)

     tau_r_add = 1 / (k_1_* T_+ k_2_),
     r_inf     = k_1_* k_3_* T_ / (k_1_* T_+ k_2_)

     In [1] the pulse is 1 ms. In my model I have changed it to adjust for
     the differences in the two models.
```

```
  The dynamic of the r* and s variables are the same as for the two state
  variables for the betafunction in [2] letting us use the exact integration
  scheme from [2].

References:
  [1] A. Destexhe et al, Kinetic Models of Synaptic Transmission,
      Methods in Neuronal Modeling (2nd ed.) MIT Press, (1998)
  [2] S. Rotter and M. Diesmann, Exact digital simulation of time-
      invariant linear systems with applications to neuronal modeling,
      Biol. Cybern. 81, 381-402 (1999)

Author:  Johan Hake, Spring 2003
 */

namespace nest {

  class GabaBSynapse:
    public Synapse
  {

  public:

    GabaBSynapse();
    GabaBSynapse(const GabaBSynapse&);

    void update(steptime T);

    void set_properties(const DictionaryDatum&);

    void get_properties(DictionaryDatum&) const;

    void calibrate(realtime);

    // Returns the conductance
    double_t* get_g();

    std::string get_name() const;

  private:

    // Variables from
    double_t k_1_;   //!< Binding rate for transmitter substance in the cleft to receptors
    double_t k_2_;   //!< Unbinding rate for the transmitter substance to the receptors
    double_t k_3_;   //!< Binding rate for the G-proteins to the K+ channel
    double_t k_4_;   //!< Unbinding rate for the G-proteins to the K+ channel
    double_t k_d_;   //!< Dissociation constant of binding G-protein to K+ channels

    double_t g_out_; //!< The total synaptic conductance

    double_t T_;     //!< Consentration of transmitter substance in the cleft.
    double_t pulse_; //!< The duration of the pulse of transmitters in the cleft
    uint_t n_;       //!< Number of individual binding sites for the G-protein in  a singel K+ channel

    double_t r_;     //!< Fraction of activated receptors.
    double_t s_;     //!< Concentration of G-proteins binded to K+ channels

    // The exact integration matrix
    double_t exp_A_00_;
    double_t exp_A_10_;
    double_t exp_A_11_;

    // Values used to register a spike
    double_t exp_r_add_;
    double_t r_inf_;
```

```
};

inline
double_t* GabaBSynapse::get_g()
{
  return &g_out_;
}

inline
void GabaBSynapse::update(steptime T)
{
  // A spike is registered
  r_ += (r_inf_ - r_)*exp_r_add_ * spikes_.collect(T);

  // The exact integration step
  s_ = exp_A_10_ * r_ + exp_A_11_ * s_;
  r_ = exp_A_00_ * r_;

  g_out_ =  g_max_ / ( k_d_ / std::pow(s_,static_cast<int>(n_)) + 1);
}

inline
std::string GabaBSynapse::get_name() const
{
  return std::string("GabaBSynapse");
}

}// namespace
#endif
```

gababsynapse.cpp

```
#include "dict.h"
#include "integerdatum.h"
#include "doubledatum.h"
#include "dictutils.h"
#include "numerics.h"
#include "network.h"

#include "gababsynapse.h"
#include <cmath>

nest::GabaBSynapse::GabaBSynapse( )
  : Synapse(),
    // All parrameter values from [1]
    k_1_(90.0),     // (M*ms)^-1
    k_2_(0.0012),   // ms^-1
    k_3_(0.18),     // ms^-1
    k_4_(0.034),    // ms^-1
    k_d_(100),
    g_out_(0.0),
    T_(1e-3),       // M
    pulse_( 1.105), //ms The pulse length are altered to fitt the model in [1]
    n_(4),
    r_(0.0),
    s_(0.0)
{
  is_lumping_ = false;
  rev_pot_ = -95.0;   // mV from [1]
  g_max_ = 0.06;      // nS
  calibrate(get_resolution());
}

nest::GabaBSynapse::GabaBSynapse(const GabaBSynapse &s)
  : Synapse(s),
```

```
    k_1_(s.k_1_),     // M*ms
    k_2_(s.k_2_),     // ms
    k_3_(s.k_3_),     // ms
    k_4_(s.k_4_),     // ms
    k_d_(s.k_d_),
    g_out_(s.g_out_),
    T_(s.T_),         // M
    pulse_(s.pulse_), // ms
    n_(s.n_),
    r_(s.r_),
    s_(s.s_)
{
  calibrate(get_resolution());
}

void nest::GabaBSynapse::get_properties(DictionaryDatum &d) const
{
  def<double_t>(d, "GabaB_k_1", k_1_);
  def<double_t>(d, "GabaB_k_2", k_2_);
  def<double_t>(d, "GabaB_k_3", k_3_);
  def<double_t>(d, "GabaB_k_4", k_4_);
  def<double_t>(d, "GabaB_k_d", k_d_);
  def<double_t>(d, "GabaB_transmitter_conc" , T_);
  def<double_t>(d, "GabaB_transmitter_pulse" , pulse_);
  def<long_t>(d, "GabaB_binding_site" , n_);
  def<double_t>(d, "GabaB_reversal_potensial", rev_pot_);
  def<double_t>(d, "GabaB_g_max", g_max_);
  def<double_t>(d, "GabaB_r", r_);
  def<double_t>(d, "GabaB_s", s_);
}

void nest::GabaBSynapse::set_properties(const DictionaryDatum &d)
{
  updateValue<double_t>(d, "GabaB_k_1", k_1_);
  updateValue<double_t>(d, "GabaB_k_2", k_2_);
  updateValue<double_t>(d, "GabaB_k_3", k_3_);
  updateValue<double_t>(d, "GabaB_k_4", k_4_);
  updateValue<double_t>(d, "GabaB_k_d", k_d_);
  updateValue<double_t>(d, "GabaB_transmitter_conc" , T_);
  updateValue<double_t>(d, "GabaB_transmitter_pulse" , pulse_);
  updateValue<long_t>(d, "GabaB_binding_site" , n_);
  updateValue<double_t>(d, "GabaB_reversal_potensial", rev_pot_);
  updateValue<double_t>(d, "GabaB_g_max", g_max_);
  updateValue<double_t>(d, "GabaB_r", r_);
  updateValue<double_t>(d, "GabaB_s", s_);
  calibrate(get_resolution());
}

void nest::GabaBSynapse::calibrate(realtime dt)
{
  double_t tau_d = 1/k_2_;
  double_t tau_r = 1/k_4_;

  // The exact integration matrix
  exp_A_00_ = std::exp( -dt / tau_d );
  exp_A_10_ = ( std::exp( -dt / tau_d ) - std::exp( -dt / tau_r ) ) / ( 1/tau_r - 1/tau_d );
  exp_A_11_ = std::exp( -dt / tau_r );

  double_t tau_r_add = 1 / (k_1_* T_+ k_2_);
  r_inf_ = k_1_* k_3_* T_* tau_r_add;
  exp_r_add_ = 1 - std::exp(-pulse_ / tau_r_add);

}
```

C.6 Spike event buffer

This is an extension of the ringbuffer class already implemented in NEST. In additional to the connection weights this buffer stores the precise spike time of the incoming spikes too. They are stored in a chronological order.

spike event buffer.h

```
#ifndef SPIKE_EVENT_BUFFER_H
#define SPIKE_EVENT_BUFFER_H
#include "nest.h"
#include "node.h"
#include <vector>

namespace nest
{

  class SpikeEvent;

  class SpikeEventBuffer {
  public:

    SpikeEventBuffer(thread max=1);

    /**
     * Collect all values at the buffer
     * origin so that the sum becomes available
     * through get_value().
     * Collect() should only be called once per time-slice
     */
    std::vector< std::pair<realtime, weight > > collect(realtime);

    /**
     * Initialize the buffer with noughts.
     */
    void clear();


    /**
     * Resize the buffer according to max_thread.
     */

    void resize();

    /**
     * Set the maximum number of threads.
     * Note that there must be at least one thread.
     * @preconditions p>0
     * @postconditions p>0
     */
    void set_max_threads(thread p);

    /**
     * Add a spike to the spike bufferds.
     */
    void add_spike(thread p, realtime st, weight w);
    /**
     *  Return true if the buffer does not contain any SpikeEvents in time step T
     */
    bool any_spike(realtime);

  private:
```

```
  thread max_threads_;
  std::vector<std::list < std::pair<realtime, weight> > > buffer_;
};


inline
  void SpikeEventBuffer::add_spike(thread p, realtime st, weight w )
{

  if (buffer_[p].empty())
  {
    buffer_[p].push_back(std::make_pair(st,w));
  }
  else
  {
    std::list< std::pair<realtime,weight> >::iterator pos = buffer_[p].end();

    /** For clarification
        if st = 14.0 and the following spike times are registrated in the
        list, the while loop stop at position showed here

            pos
             |
             v
        10.0 12.0 15.0 17.0 end()
     To add the spike the iterator pos have to be moved one step up
    */

    while ( pos != buffer_[p].begin() && (--pos)->first>st)
      {
        ;
      }
    buffer_[p].insert(++pos,std::make_pair(st,w));
  }
}

inline
std::vector< std::pair<realtime, weight> > SpikeEventBuffer::collect(realtime now_plus_dt)
{
  std::vector< std::pair<weight, realtime> > spikes;

  // Steping through the threads
  for(thread p=0; p< max_threads_; ++p)
  {
    std::list< std::pair<realtime,weight> >::iterator it = buffer_[p].begin();
    while( it != buffer_[p].end() && (*it).first < now_plus_dt )
    {
      spikes.push_back( (*it) );
      buffer_[p].erase(it++);
    }
    // Clear the collected vector
  }
  return spikes;
}

inline
bool SpikeEventBuffer::any_spike(realtime now_plus_dt)
{
  for(thread p=0; p< max_threads_; ++p)
  {
    // Check first if the buffer is empty then if the first
    if (! buffer_[p].empty() && buffer_[p].front().first < now_plus_dt)
      return true;
```

```
    }
    return false;
  }
}

#endif
```

spike event buffer.cpp

```
#include "debug.h"
#include "spike_event_buffer.h"

/**
   Buffer Layout.
   The SpikeEventBuffer contains a vector of list. The number of vectors correspond
   to the number of threads. Every list hold the information of an incomming spike.
   The spike time and the weight. A spike is added to the list so the list contains
   an ordered secuenze of spikes.
*/


nest::SpikeEventBuffer::SpikeEventBuffer(thread max_t)
  :max_threads_(max_t)
{
  std::list< std::pair<weight, realtime> > sl;
  buffer_.insert(buffer_.begin(),max_t,sl);
}


void nest::SpikeEventBuffer::resize()
{
  std::list< std::pair<weight, realtime> > sl;
  buffer_.resize(max_threads_,sl);
}

void nest::SpikeEventBuffer::set_max_threads(thread max_t)
{
  assert(max_t >=1);
  max_threads_=max_t;
  resize();
}

void nest::SpikeEventBuffer::clear()
{
  buffer_.clear();
}
```

Appendix D

Selected SLI code

D.1 Introduction

In this appendix a selected SLI code file is presented. The selected code is taken from the simulation of the fifth test, from chapter 6. Here the neurons are all coupled and the new *SynapseConnect* command is used, see line 88 and 109.

D.2 The fifth test

```
1   /method [(2) (2ts) (2tsps) (4) (4ts) (4tsps)] def

    modeldict begin
      userdict begin
5
        0 1 5  % Loop for the different neuron types
        {
          /neuron_type Set

10        0 1 12 % Loop for the different time step length
          {
            /dt_step Set
            ResetKernel
            [0] << /threads 1 >> SetStatus
15
            % Sets the resolution
            /dt 1.0 def
            1 1 dt_step
              {; dt 2 div /dt Set}
20          for
            dt SetResolution

            % Sets the delay for the spikes
            /spike_delay 0.5 dt div round cvi def
25          spike_delay 0 lt
              {/spike_delay 1 def}
            if

            % Sets the delay on the input devices
30          /device_delay 1.0 dt div round cvi def
```

```
% Create an array with random number between -80.0 -40.0
rngdict /knuthlfg get 12 CreateRNG /myrand Set
/V_init [101] {; myrand drand 40 mul -80 add} Table def

% Creates 101 lifb neurons and an array with the adresses to the neurons
mark
  neuron_type 1 lt {lifb_neuron_rk2 exit} case
  neuron_type 3 lt {lifb_neuron_rk2ts exit} case
  neuron_type 4 lt {lifb_neuron_rk4 exit} case
  neuron_type 6 lt {lifb_neuron_rk4ts exit} case
switch
101 CreateMany ;
/lifb_ad [1 101] {0 exch 2 arraystore} Table dup join def

% Creates a weight array
/gauss {10.0 div dup mul -0.5 mul exp} def
/weights [-50 50] {gauss} Table dup join def

% Creates 101 ac generators and 1 dc generator
neuron_type 3 lt
  { ac_generator }
  { ac_generator_2 }
ifelse
101 CreateMany ;
/ac_gen_ad [102 202] {0 exch 2 arraystore} Table dup join def
/dc
neuron_type 3 lt
  { dc_generator }
  { dc_generator_2 }
ifelse
Create def

% Creates the mesuring devices
/sd prec_spike_det Create def

% Return the lifb and the ac_gen adress
/lifb_ad_get {lifb_ad exch get} def
/ac_gen_ad_get {ac_gen_ad exch get} def

% Set upp the network!
0 1 100 {
  /out Set
  0 1 100 {
    % Puts an arrays with the source node on the stack
    /in Set out lifb_ad_get

    %Puts an array with the target node on the stack
    out in add lifb_ad_get
    mark
      neuron_type 2 lt {AmpaSynapse exit} case
      neuron_type 3 lt {AmpaSynapse_ps exit} case
      neuron_type 5 lt {AmpaSynapse_2 exit} case
      neuron_type 6 lt {AmpaSynapse_2ps exit} case
    switch

    % Synapse connect two neurons
    SynapseConnect ; /port Set

    % Sets the weight
    out lifb_ad_get port weights in get SetWeight

    % Sets the delay
    out lifb_ad_get port spike_delay SetDelay

    % Puts an arrays with the source node on the stack
```

```
        out lifb_ad_get

        %Puts an array with the target node on the stack
        out in add lifb_ad_get
        mark
          neuron_type 2 lt {GabaASynapse exit} case
          neuron_type 3 lt {GabaASynapse_ps exit} case
          neuron_type 5 lt {GabaASynapse_2 exit} case
          neuron_type 6 lt {GabaASynapse_2ps exit} case
        switch

        % Synapse connect two neurons
        SynapseConnect ; /port Set

        % Sets the weight
        out lifb_ad_get port weights in 50 add get SetWeight

        % Sets the delay
       out lifb_ad_get port spike_delay SetDelay

      } for

      out ac_gen_ad_get << /amplitude 0.7
                           /frequency 4.0
                           /phase out 100.0 div 360 mul
                        >> SetStatus


      %Put two ac_gen adress on the stack one for connection one for delay
      out ac_gen_ad_get out ac_gen_ad_get
      out lifb_ad_get Connect device_delay SetDelay

      dc dc out lifb_ad_get Connect device_delay SetDelay

      out lifb_ad_get sd Connect ;

    } for

    % Setts the weights of the synapses and the initial values of the potentials
    % and the h variables
    0 1 100
    {
      /out Set
      out lifb_ad_get << /GabaA_g_max 0.01
                         /Ampa_g_max 0.014
                         /V V_init out get
                         /h 0.0
                      >> SetStatus
    } for

    dc << /amplitude -0.2 >> SetStatus

    /name (rk) method neuron_type get join (_dt_-) dt_step cvs join join def
    /ospks name (_spike.dat) join ofstream ; def

    sd << /output_stream ospks
          /withtime true
          /withpath true
       >> SetStatus

    % For registration of the potential of one neuron
    %vm_one voltmeter Create def
    %vm_one 99 lifb_ad_get Connect ;
    %opot_one  name (_onepot.dat) join ofstream ; def
    %vm_one << /output_stream opot_one
```

```
            %          /withpath false
            %          /start 0.0
            %>> SetStatus
165
            % SIMULATE!!
            (Simulating: ) name join =

            % The length of the simulation
170         /simlength 500 def
            simlength dt div cvi 1 add Simulate

            % Recording the last potentials
            /opot_all name (_lastpot.dat) join (w) file def
175
            opot_all () <- 16 setprecision
            0 1 100
              { lifb_ad_get GetStatus /V get <- (\n) <- }
            for
180         flush ;

            ospks close
            opot_all  close
            %opot_one  close
185

          } for

        } for
190
    end

end
```

References

Coutinho, V. and T. Knöpfel (2002). Metabotropic glutamate receptors: Electrical and chemical signaling properties. *The neuroscientist 8*, 551–561.

Dayan, P. and L. F. Abott (2001). *Theoretical Neuroscience*. Cambridge, MA: MIT Press.

Destexhe, A., Z. F. Mainen, and T. J. Sejnowski (1994). An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Comput 6*, 14–18.

Destexhe, A., Z. F. Mainen, and T. J. Sejnowski (1998). Kinetic models of synaptic transmission. In C. Koch and I. Segev (Eds.), *Methods in Neuronal Modeling*. Cambridge, MA: MIT Press.

Diesmann, M. and M.-O. Gewaltig (2003). Nest: An environment for neural systems simulations. In T. Plesser and V. Macho (Eds.), *Beiträge zum Heinz-Billing-Preis 2001*, Forschung und wissenschaftliches Rechnen, pp. 43–70. Göttingen: Gesellschaft für wissenschaftliche Datenverarbeitung mbH.

Diesmann, M., M.-O. Gewaltig, and A. Aertsen (1999). Conditions for stable propagation of synchronous spiking in cortical neural networks. *Nature 402*, 529–533.

Emri, Z., K. Antal, and V. Crunelli (2003). The impact of corticothalamic feedback on the output dynamics of a thalamocortical neurone model: The role of synapse location and metabotropic glutamate receptors. *Neuroscience 117*, 229–239.

Finney, R. L. and G. B. Thomas (1994). *Calculus* (Second ed.). New York: Addison-Wesley.

Hansel, D., G. Mato, C. Meunier, and L. Neltner (1998). On numerical simulations of integrate-and-fire neural networks. *Neural Comput 10*, 467–483.

Heggelund, J. J. Z. . P. (2001). Muscarinic regulation of dendritic and axonal outputs of rat thalamic interneurons: A new cellular mechanism for uncoupling distal dendrites. *J Neurosci 21*, 1148–1159.

Hugenard, J. R. and D. A. McCormick (1992). Simulation of the currents involved in rhythmic oscillations in thalamic relay neurons. *J Neurophysiol 68*, 1373–1383.

Lambert, J. D. (1991). *Numerical methods for ordinary differential systems*. Chichester: John Wiley & Sons.

Lapicque, L. (1907). Recherches quantitatives sur l excitation electrique des nerfs traitee comme une polarization. *J Physiol Pathol Gen 9*, 620–635.

Mathews, J. H. (1987). *Numerical methods: For computer science, engineering, and mathematics*. London: Prentice-Hall.

McCormick, D. A. and J. R. Hugenard (1992). A model of the electrophysological properties of thalamocortical relay neurons. *J Neurophysiol 68*, 1384–1400.

Rotter, S. and M. Diesmann (1999). Exact digital simulation of tim-invariant linear systems with application to neural modeling. *Biol. Cybern. 81*, 381–402.

Shelley, M. J. and L. Tao (2001). Efficient and accurate time-stepping schemes for integrate-and-fire neuronal networks. *J Comp Neurosci 11*, 111–119.

Sherman, S. M. and R. W. Guillery (2001). *Exploring the Thalamus*. New York: Academic Press.

Sherman, S. M. and C. Koch (1998). Thalamus. In G. M. Shepherd (Ed.), *The synaptic organization of the brain* (Fourth ed.). New York: Oxford University Press.

Smith, G. D., C. L. Cox, S. M. Sherman, and J. Rinzel (2000). Fourier analysis of sinusoidally driven thalamocortical relay neurons and a minimal integrate-and-fire-or-burst model. *J Neurophysiol 83*, 588–610.

Smith, G. D. and S. M. Sherman (2002). Detectability of excitatory versus inhibitory drive in an integrate-and-fire-or-burst thalamocortical relay neuron model. *J Neurosci 22*, 10242–10250.

Sterling, P. (1998). Retina. In G. M. Shepherd (Ed.), *The synaptic organization of the brain* (Fourth ed.). New York: Oxford University Press.