SOLVERS FOR DIFFERENTIAL EQUATIONS FOR A NEURON MODEL WITH NON-LINEAR DYNAMICS

LØSERE FOR DIFFERENSIALLIGNINGER FOR EN NEVRONMODELL MED IKKE-LINEÆR DYNAMIKK

ANDERS GRØNVIK JAHNSEN

NORWEGIAN UNIVERSITY OF LIFE SCIENCES
DEPARTMENT OF MATHEMATICAL SCIENCES AND TECHNOLOGY
MASTER THESIS 30 CREDITS 2010

Preface

This master thesis marks the end of my five years as a Master of Science student at the Department of Mathematical Sciences and Technology at the Norwegian University of Life Sciences. I have attended the study programme "Environmental physics and renewable energy", and the master thesis is the result of the last half years work (30 study points).

Hans Ekkehard Plesser has been my supervisor for this master thesis. Thank you for good discussions and helpful supervision. I would also thank you for introducing me to informatics and programming.

To good friends: Thank you for your support and discussions, both of scientific and non-scientific character.

Finally I want to thank Aud Jorunn Karlsen Hovda for all help and support you kindly have given me.

Ås, 12th of May, 2010

Anders Grønvik Jahnsen

Abstract

The human kind knows a lot about nature but it is still overwhelming many phenomena the humans in total do not know much about. This is influenced by the fact that many of the systems in nature are complex. A possible way of investigating such systems is by simulations. The increase in computational power during the last years has made it possible to do research on more and more complex systems.

Neuroscience is the science which deals with the understanding of the brain and the nervous system. Such systems are very complex. Simulations are therefore a necessary tool to increase the understanding in this field.

Neurons are an important building brick of the brain and the nervous system. Different types of neurons exist and are specialized for different purposes. Before it is possible to simulate their behaviour, a mathematical model of equations has to be made. Depending on what effects it is desirable to investigate, the description of the neuron varies.

Neural Simulation Toolbox (NEST) is a simulator aimed to simulate networks of neurons and measure the network activity. NEST supports different neuron models. Among them is the conductance based model of a leaky integrate-and-fire neuron, named iaf_cond_alpha.

The aim of this master thesis has been to decrease the simulation time for this model. Two new versions of it have therefore been made. They are named iaf_cond_alpha_ei and iaf_cond_alpha_ei_step.

The simulation times for the new models are approximately 25% for the iaf_cond_alpha_ei model and 30% for the iaf_cond_alpha_ei_step model, when comparing to the simulation time for the original iaf_cond_alpha model.

Neuron models are described by differential equations. These equations are integrated numerically in a simulator. This will take the system forwards in time.

In the models used in this master thesis there are five equations that have to be evaluated. Four of the equations describe the synaptic dynamics of

the neuron and the last one describes the membrane potential.

The original and the new models differ in which methods for numerical integration that are used. Iaf_cond_alpha uses a Runge-Kutta-Fehlberg method of fourth-fifth order with adaptive step size control from GNU Science Library. The new models use Exact Integration for updating of the synaptic dynamics. Updating of the membrane potential is done with a manually implemented Runge-Kutta method. The iaf_cond_alpha_ei model uses a fourth order method and the iaf_cond_alpha_ei_step uses a fifth order method.

It is, however, not enough to reduce the simulation time. The numerical precision and error is also important. The new models are tested for numerical precision and error, and are as good as the original iaf_cond_alpha model.

Both the original and the new neuron models are written in the C++ programming language. The testing is done in the Python programming language. NEST Python interface PyNEST has been used to easily create neurons, adjust parameter values, make simulations and pick out the desired recorded values.

Sammendrag

Menneskene vet mye om naturen, men det er fortsatt veldig mange fenomener som menneskene samlet sett ikke vet stort om. At mange av systemene som finnes i naturen er komplekse påvirker til dette. En mulig måte å finne ut mer om slike systemer er å simulere dem. I de senere årene har det blitt mulig å gjøre undersøkelser på stadig mer komplekse systemer. Dette skyldes en kraftig økning av regnekraft.

Nevrovitenskap er vitenskapen der en prøver å forstå hjernen og nervesystemet. Dette er svært komplekse system. Simuleringer er derfor et nødvendig hjelpemiddel og verktøy for å kunne øke kunnskapen på dette området.

Nevroner er viktige byggesteiner i hjernen og i nervesystemet. Det eksisterer forskjellige typer nevroner som alle er spesialisert for ulike formål. Før det er mulig å simulere et nevrons oppførsel må det lages en matematisk modell som i ligninger beskriver oppførselen til nevronet. Hvordan nevronet modelleres er avhengig av hvilke egenskaper som ønskes undersøkt.

NEST (Neural Simulation Toolbox) er en simulator utviklet med det til hensikt å kunne simulere nettverk av nevroner og måle aktiviteten i nettverket. NEST har støtte for bruk av mange ulike nevronmodeller. Blant dem er en konduktansebasert modell av et integrer-og-fyr nevron med lekkstrøm (Engelsk betegnelse: Integrate-and-fire neuron). Denne modellen heter iaf_cond_alpha.

Arbeidet utført i denne mastergradsoppgaven har hatt til hensikt å redusere simuleringstida for nevronmodellen iaf_cond_alpha. Det er derfor blitt lagd to nye versjoner basert på den originale modellen. De heter iaf_cond_alpha_ei og iaf_cond_alpha_ei_step.

Simuleringstida for de nye modellene iaf_cond_alpha_ei og iaf_cond_alpha_ei_step er henholdsvis omtrent 25% og 30% av simuleringstida til den originale iaf_cond_alpha-modellen.

For å beskrive nevronmodeller brukes differensialligninger. I en simulator blir disse integrert numerisk. Dermed tas systemet framover i tid.

I modellene som brukes i denne mastergradsoppgaven er det fem ligninger som må oppdateres. Fire av dem beskriver signaler inn til nevronet via synapsen. Den siste beskriver membranpotensialet.

Den originale og de nye modellene bruker ulike metoder for numerisk integrasjon. Iaf_cond_alpha bruker en Runge-Kutta-Fehlberg metode av fjerde-femte orden med skrittlengdekontroll fra GNU Science Library. De nye modellene bruker Eksakt Integrasjon for å oppdatere dynamikken i synapsen. Oppdatering av membranpotensialet gjøres med en manuelt implementert Runge-Kutta metode. Modellen iaf_cond_alpha_ei bruker en fjerdeordens metode og iaf_cond_alpha_ei_step bruker en femteordens metode.

Uansett er det ikke tilstrekkelig å redusere simuleringstida. Numerisk presisjon og feil er også viktig. De nye modellene har blitt testet på både numerisk presisjon og feil, og er like gode som den originale iaf_cond_alpha-modellen.

Både den originale nevronmodellen og de to nye er skrevet i programmeringsspråket C++. Testing er gjort med programmeringsspråket Python. Brukergrensesnittet til NEST via Python, PyNEST, har blitt brukt for å enkelt kunne skape nevroner, forandre parameterverdier, kjøre simuleringer og hente ut ønskede lagrede verdier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A simulator is trying to recreate nature. They are necessary tools when studying complex systems, as done in neuroscience. The latest years it has been an enormous increase in accessible computational power. This has made it possible to simulate more complex systems than before and with more variations of parameter values. This has caused a simultaneous increase in the demand for such possibilities. The effiency of the simulators used is therefore important.

The precision of the results is important. Results delivered efficiently are not enough if the results not are correct enough. The desired degree of error depends on the research that should be done. In some cases results delivered efficiently is more important than as little error as possible, and in other cases it is the other way around. No matter what the researchers actually want it is necessary that simulators are able to deliver precise results.

There are developed simulators which emphasize research on different systems in neuroscience. The neuron is an important building brick of the brain. Some simulators are used to simulate, for example, the internal changes inside a neuron. Others are used to simulate networks of neurons to understand more about the network activity.

In this master thesis the precision and efficiency of one neuron model in the simulator NEST (Neural Simulation Toolbox) have been investigated. The aim was to decrease the simulation time, but still keep the numerical precision and error as good as they were.

1.1 NEST - Neural Simulation Toolbox

NEST[1] is a simulation tool made for modelling the behaviour of networks of neurons. It supports a large number of neuron types and has several tools necessary when modelling different kinds of networks. NEST is aimed to be used for simulations of large neuronal networks (more than 10^4 neurons and 10^7 to 10^9 synapses) [Eppler et al., 2009].

The computational performance is important in NEST. It is therefore written in the programming language C++ which has many possibilities for computational optimization. Before any user interface had been developed, simulations were set up by writing the C++ code for the whole simulations. This is not basic knowlegde, and the user has to know quite a lot about programming and C++.

The original user interface for NEST was made in its own simulation language SLI. A more convenient programming language for NEST was required and PyNEST was created. It combines NEST and the programming language Python. PyNEST keeps both the efficient simulation kernel in NEST and the simplicity and flexibility which Python gives [Eppler et al., 2009].

1.2 Scope of the thesis

In NEST it is possible to simulate several different neuron models. Among them is the conductance based leaky integrate and fire model (see Section 2.3) named iaf_cond_alpha. This model delivers precise results, but the simulations could be done faster and more efficient.

Every neuron model is described by some differential equations. These equations must often, and also in the case of the iaf_cond_alpha model, be solved numerically with use of some integration routine (see Chapter 3). This is necessary for updating variables which describes the state of the neuron. The iaf_cond_alpha model uses a routine from the GNU Science Library. The numerical integration could be done faster if using Exact Integration (see Section 3.4) instead of the GSL routine.

The aim of this master thesis has been to make the iaf_cond_alpha neuron model faster and still keep the numerical precision and error at the same level as it is today. It is therefore possible to say that this thesis regards giving a contribution to the development of this simulator.

The work done during this master thesis has resulted in two new neuron models which both deliver faster simulations and keeps the precision. All

[1] http://www.nest-initiative.org

the work done is described in this master thesis.

Chapter 1 is this introduction to the master thesis. Here the simulator NEST (Neural Simualtion Toolbox) has been described. The aim and results of the thesis are also explained briefly. An overview of the thesis, with more details than in the table of contents, is given.

Chapter 2 deals with neurons. The nervous system is treated briefly, but the biological explanation of a neuron and neuron models are emphasized. Important effects as neuron spiking and the connection between information sent by a neuron and its firing rate is explained. Different types of neuron models are explained before variations of the used leaky integrate-and-fire neuron are described.

Neuron models are described by differential equations. Often they have to be solved numerically. Chapter 3 is an introduction to numerical integration. Several different methods for numerical integration exist. Three of them are explained more in detail. These are Euler's method, Runge-Kutta methods of different orders, and Exact Integration.

Chapter 4 concerns the methods used. The development of the new neuron models are explained, and also the updating of them. Testing of the different neuron models has been an important part of this thesis. This chapter therefore also contains a description of the different test methods used. Parameters used for testing are given in this chapter.

The results of this master thesis work is described in Chapter 5, 6 and 7. Chapter 5 is about some differences found in this master thesis which are caused by the fact that numbers have to be represented numerically. Precision and error in general is treated in Chapter 6. Chapter 7 concerns simulation times.

The last chapter in the main part of this master thesis is the summary in Chapter 8. The conclusions based on the results are drawn and some suggestions for further work are given.

There are two appendices in this master thesis. Appendix A gives an introduction to PyNEST, the Python user interface of NEST. The source code of the new neuron models and the scripts used for production of data or figures included in this master thesis can be found in Appendix B. The bibliography of used references is included at the very end.

Chapter 2

Neurons

This chapter aims to give an introduction to neurons. Section 2.1 deals with the nervous system in general. Section 2.2 concerns a biological view on neurons and includes both a description of the different parts of a neuron and its properties. The last section, Section 2.3, explains about neuron models. They are necessary when trying to recreate the behaviour of a neuron in a simulator.

The whole chapter is based on sources from the neuroscience books "*Theoretical Neuroscience: Computational and Mathematical Modeling of Neuronal Systems*" [Dayan and Abbott, 2001] and "*The Brain: A Neuroscience Primer*" [Thompson, 2000].

2.1 The nervous system

The first kind of nervous system that was developed was in animals such as jellyfish. Jellyfish can react on nutrients in the water and swim to capture it. This was a great development from the sponge, which just sits on the bottom of the sea and whose survival depends on whether nutrients are passing nearby or not. The jellyfish's possibility to react on the surroundings shows that different cells must have been created and that the cells are communicating. Muscle tissue contracts and is controlled by signals sent from neurons.

All animals have a nervous system which controls the behaviour of the body. The neurons are also thought to act the same way in all animals, from jellyfish to humans. The main difference separating animals are the number of neurons and the ways they are put together. Humans have a huge number of neurons put together in a complicated way and therefore have the possibility of a sophisticated and adaptive behaviour.

The centre of the nervous system is located in the brain. A brain with a large number of neurons compared to the body mass gives the species an advantage compared to a species with a smaller number of neurons. This advantage ensured human survival.

The humans do not have the biggest brain among mammals. The whales have the biggest and both porpoises and elephants have bigger. But the human brain is largest compared to the body size [Thompson, 2000].

The presence of the human nervous system makes us more adaptable to the constantly changing surroundings. We can process the signals from our sensory instruments (eyes, ears and so on) and make better decisions.

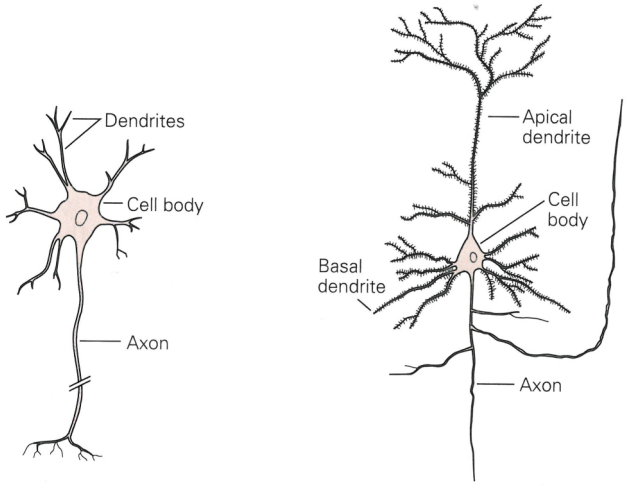
But not all things that live on the Earth have a nervous system. Plants are an example of this. Instead of a nervous system the plants interact with the surroundings based on reflexes. One reflex could be to produce sugar during photosynthesis when there is enough light. The cycles that plants go through, both daily and seasonally, are controlled by these reflexes.

2.2 Biology of neurons

Neurons consist of a well-developed cell nucleus that contains chromosomes with the genetic material (DNA). The same applies to cells in multi-cellular plants and animal organisms. In contrast to other cells, the reproduction of neurons stops when a human is born. The number of neurons in the brain and nervous system is therefore constant, or decreasing, throughout a human life.

Some fibres extend from the nucleus of a neuron. These are used to create connections between neurons where information could be sent or received. All sending of information goes through only one fibre called the axon. Through the other fibres, called dendrites, the individual neuron receives information. The length of the dendrites is tens to hundreds micrometers. The length of the axons in the human body, on the other hand, can differ a lot. Some neurons have axons which are up to one meter long and some axons have a length comparable to the length of the dendrites. The axon divides into a number of small fibers. In these ends there are terminals that make the connection with one other cell through the synapse.

Depending of the type of neuron, the axon could make different connections. The axon from the motor neuron, as showed in Fig. 2.1a, creates connections with muscle cells. The pyramidal cells, as showed in Fig. 2.1b, are the main work horses in the cortex, which is a part of the brain. They do not connect with muscle cells. Instead connections to other neurons are made, and they can connect with both dendrites and the cell body of the actual neuron.



(a) Motor neuron **(b)** Pyramidal neuron

Figure 2.1: Schematic view of different types of neurons and its parts [Kandel et al., 1991, Fig. 2-4 D]. Figure 2.1a is a motor neuron, which sends signals to muscles. Figure 2.1b is a pyramidal neuron. They are the work horses in cortex, which is a part of the brain.

Other types of neurons represented in the brain can make the same types of connections.

Fig. 2.1 gives a schematic view of a motor neuron and a pyramidal neuron. Both figures show the cell body, the axon and the dendrites. The motor neuron shows a good schematic view of neurons, if only one type should be presented.

The largest collection of neurons in humans is found in the brain. The neurons are important parts for the brains function. It is thought that the brain consists of as many as a trillion (10^{12}) individual neurons. Each of these neurons is a separate cell and is connected to several thousands of other neurons. It could therefore be around 10^{15} connections in the brain. Thompson [2000] states that in general the number of *possible* connections in the brain is larger then the total number of atomic particles that makes up the known universe.

In the cell membrane of the neuron there are ion channels that allow ions to move into and out of the cell. The dominant ions are sodium (Na^+), potassium (K^+), calcium (Ca^{2+}) and chloride (Cl^-). The flow of ions through the membrane is controlled by the opening and closing of ion channels. This is done in response to voltage changes and external and internal signals.

Fig. 2.2 shows ion channels in the axon membrane and how these ions are

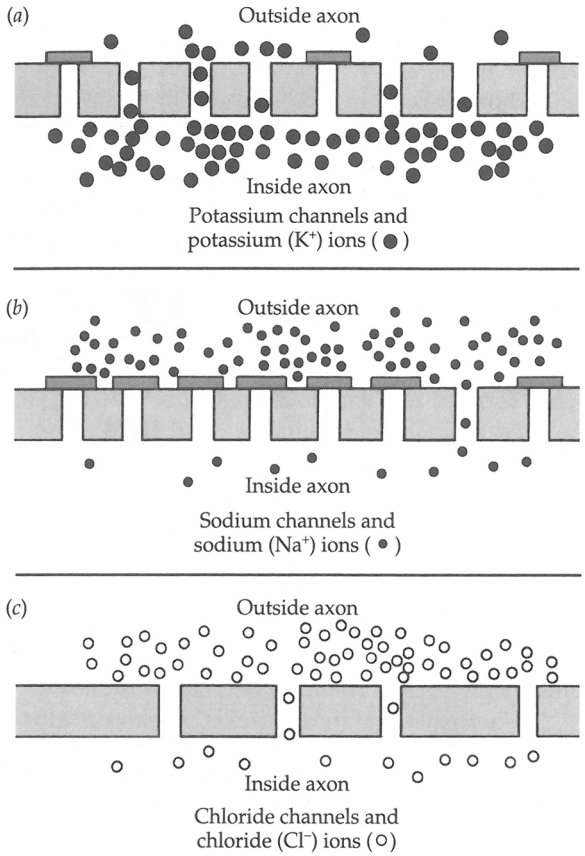


Figure 2.2: Ion channels and ion concentration in the axon membrane of the neuron [Thompson, 2000, Fig. 3.5]. Figure a): Most potassium ions are open and most of the potassium ions are inside. Figure b): Most of the sodium ions are outside and the sodium channels are mostly closed. Figure c): The chloride channels are open and the chloride ions are outside. There are not as many chloride channels as there are channels for potassium and sodium.

distributed between inside and outside of the neuron. The potassium ions are mostly inside and just some of its channels are closed. Most of the sodium channels are closed and the concentration of sodium ions is highest outside the axon. The number of chloride channels is not as large as the number of channels for potassium or sodium, but they are all open. The chloride ions are mostly at the outside.

The membrane potential is the difference between the voltage inside the cell and in the surroundings. It is, among other things, determined by the relative number of open ion channels. The voltage in the surroundings is conveniently set to 0 mV (millivolt). The resting potential of a neuron is

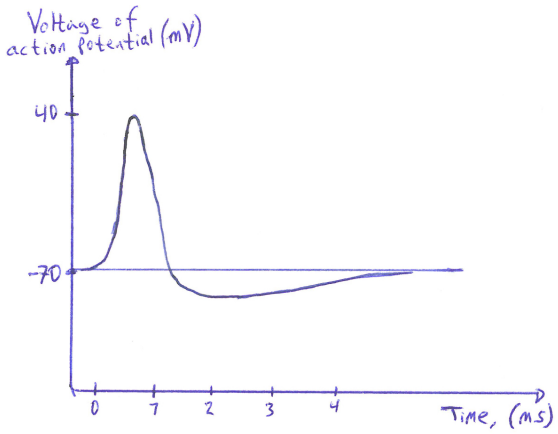


Figure 2.3: Schematic view of an action potential/a spike. Based on Thompson [2000, Fig. 3.12]

about -70mV.

A special property of neurons is their ability to propagate signals rapidly over large distances. The electric pulses that travel down the axon are called action potentials, or simply spikes. Depending on the pattern and frequency of the firing spike sequences, different information is sent down the axon to other neurons and cells.

These action potentials are fluctuations in the membrane potentials of roughly 100 mV that lasts for about 1 ms (millisecond). If the membrane potential gets over a threshold the neuron will send out a spike. The typical threshold value of the membrane potential, V_{th}, is about -55 to -50 mV. Fig. 2.3 shows a schematic view of a spike.

After a spike the neuron becomes refractory, first in the absolute refractory period and second in the relative refractory period. In the first one it is almost impossible for the neuron to create a new spike. This period does not last long, only a few milliseconds. The relative refractory period, on the other hand, lasts for up to tens of milliseconds. Here it is more difficult than normal to create a spike.

Neurons communicate with other neurons or cells through spikes. Because all spikes look the same, the content of the message which a neuron sends out is determined by the pattern and the frequency of the spikes. The frequency of the spikes tells the mean value of spikes per time. It could change over time, and then be time dependent, or it can be constant over time, and then be independent of time. The pattern of the spikes tells how the spikes are

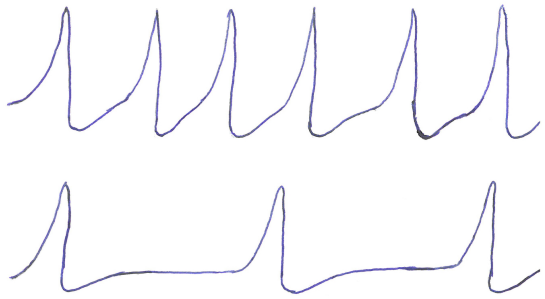


Figure 2.4: Neuron firing rate. Top figure shows high firing rate and the lower one shows low firing rate

distributed over time.

The frequency of the spikes are in many cases taken as the important parameter to describe a spike train. It is therefore important to measure it. The parameter which tells the frequency of the spikes is called the firing rate of a neuron. Fig. 2.4 shows examples of both high and low firing rate.

Although the frequency could influence the message from the neuron more than the pattern of spikes, the pattern is also important. Fig. 2.5 shows different possibilities of spike trains which have the same frequency, but different patterns. In these cases also the pattern influence on the message sent.

The simplest pattern is illustrated as the red crosses in Fig. 2.5. In this case the spikes are separated with the same distance of time. The message is then determined by the frequency of the spikes. One step further in complexity is a spike train where several spikes come with a short distance between themselves and the time distance to the next one is longer. Both the blue crosses (on short, one long) and the green crosses (two short, two long) are examples of this type of pattern. In these cases the frequency could no longer tell the whole truth about the information sent out from the neuron.

2.3 Neuron models

The last section gave an introduction to neurons. It was shown that neurons consist of three main parts, the neuron cell body, the dendrites and the axon. The neurons have a membrane too. Here is ion transport possible and the

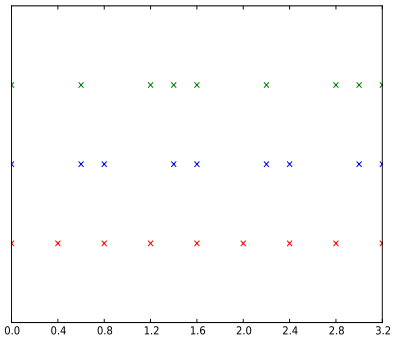


Figure 2.5: Different spike patterns, same frequency

membrane defines the shape of the neuron.

When trying to recreate nature, as done in a simulation, it is useful to make a model of the system we are dealing with. In all cases it is possible to make models which span from simple models, which only cover the basics, to complex models, where all known mechanisms are taken into account. A simple model is easy to use, but it may be so simple that the results not are mirroring what would have happened in the real system. Simulations with a complex model will hopefully show a true picture of the real system. But there are some disadvantages with these types of models. Many mechanisms that are taken into account humans do not understand very well. It is therefore uncertain if the mechanisms are modelled the right way and if the values of the parameters used are representative. If these mechanisms are modelled wrong and incorrect parameter values are used, using a complex model could give a false security about the results of the simulation. Simulations with complex models could also need more resources than necessary.

When modelling neurons it is possible to make the model with all the mechanisms which happen inside the neuron cell body taken into account. This is called a full-compartment model. When modelling networks of neurons the network activity is the main thing of interest. Therefore it is sufficient to use a less complex model. The model used is called a single-compartment model and models the neuron as a point neuron.

Single-compartment models use a single variable V to describe the membrane potential of a neuron. Multi-compartment models can also describe spatial variations in the potential. The basic equation for single-compartment models is [Dayan and Abbott, 2001, Eq. 5.6]

$$c_{\mathrm{m}} \frac{\mathrm{d}V}{\mathrm{d}t} = -i_{\mathrm{m}} + \frac{I_{\mathrm{E}}}{A}. \tag{2.1}$$

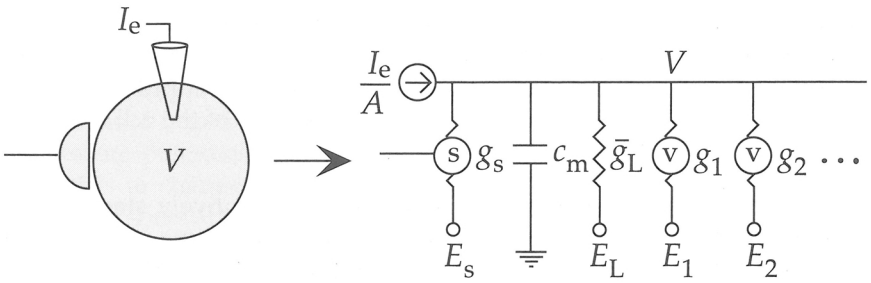


Figure 5.4 The equivalent circuit for a one-compartment neuron model. The neuron is represented, at the left, by a single compartment of surface area A with a synapse and a current-injecting electrode. At right is the equivalent circuit. The circled (s) indicates a synaptic conductance that depends on the activity of a presynaptic neuron. A single synaptic conductance g_s is indicated, but in general there may be several different types. The circled (v) indicates a voltage-dependent conductance, and I_e is the current passing through the electrode. The dots stand for possible additional membrane conductances.

Figure 2.6: Equivalent electric circuit to a single-compartment model. I_e equals I_E. The original figure text is included for full explanation of the figure [Dayan and Abbott, 2001, Fig. 5.4].

The charge in the neuron builds up with a rate equal to the total amount of current entering the neuron. The change of the membrane potential $\frac{dV}{dt}$ multiplied with the total capacitance C_m gives this current. The current is also equal to the membrane current plus eventually an experimentally added current I_E. Since the membrane current usually is given as current per unit area, both total capacitance and experimentally added current must be divided by the neuron area. It gives the specific capacitance c_m and $\frac{I_E}{A}$, where A is the surface of the neuron. Putting all this together gives Eq. (2.1).

A single-compartment model has its equivalent electric circuit. Both the neuron and the equivalent circuit are shown in Fig. 2.6. The neuron has surface area A, one synapse and a current-injecting electrode. The electric circuit is modelled with a conductance dependent on the presynaptic input, a synaptic conductance and several voltage-dependent conductance. The original explanation from Dayan and Abbott [2001] is included in the figure and explains better.

Integrate-and-fire-models (I&F models) consider only the sub-threshold dynamics of the membrane potential. This means that the biophysical mechanisms that are responsible for action potentials are not explicitly taken into account. They are well known and could be modelled, but are excluded to make neuron models easier and simulations faster. In the I&F model a neuron spikes whenever its membrane potential reaches the threshold potential V_{th}. After a spike the membrane potential is reset to a reset potential

$V_{\text{reset}} < V_{\text{th}}$. The neuron is then refractory for some time before it has the opportunity to send out a new spike.

The most used version of the I&F model is called passive or leaky integrate-and-fire model. In this model the "entire membrane conductance is modeled as a single passive leakage term, $i_{\text{m}} = \bar{g}_{\text{L}}(V - E_{\text{L}})$" [Dayan and Abbott, 2001]. The resting potential of the cell is E_{L}. Combining it with Eq. (2.1) gives [Dayan and Abbott, 2001, Eq. 5.7]

$$c_{\text{m}}\frac{\mathrm{d}V}{\mathrm{d}t} = -\bar{g}_{\text{L}}(V - E_{\text{L}}) + \frac{I_{\text{E}}}{A}. \tag{2.2}$$

The specific membrane resistance $r_{\text{m}} = 1/\bar{g}_{\text{L}}$. The membrane time constant τ_{m} is defined as the specific membrane conductance multiplied with the specific membrane resistance, $\tau_{\text{m}} = c_{\text{m}} r_{\text{m}}$. Multiplying Eq. (2.2) with r_{m} and using the connections given over, gives the basic equation for leaky I&F models [Dayan and Abbott, 2001, Eq. 5.8]

$$\tau_{\text{m}}\frac{\mathrm{d}V}{\mathrm{d}t} = E_{\text{L}} - V + I_{\text{E}}R_{\text{m}}. \tag{2.3}$$

This equation does not depend of the surface area A of the cell, as Eq. (2.2) did. Instead the total resistance R_{m} is used. Using Ohms law it can be showed that $r_{\text{m}} = R_{\text{m}}A$.

Eq. (2.4) starts with Ohms law $U = IR$. The total voltage is also equal to specific current multiplied with specific recistance. The specific current is $i_s = \frac{I}{A}$. If the product $i_s r_s$ should be equal U, the specific resistance r_s must be given as $R_s A$.

$$U = IR = i_s r_s = \frac{I}{A}(RA) = IR \tag{2.4}$$

If the current I_{E} is the only input to the neuron, the membrane potential will, with time constant τ_{m}, reach this value as time $t \to \infty$. An analytical solution of Eq. (2.3) could be computed, independent on whether I_{E} is time dependent or not.

In the case of time independent current I_{E} and initial condition $V(t = 0) = E_{\text{L}}$, the analytical solution is

$$V(t) = E_{\text{L}} + \frac{I_{\text{E}}\tau_{\text{m}}}{C_{\text{m}}}(1 - \mathrm{e}^{-t/\tau_{\text{m}}}). \tag{2.5}$$

2.3.1 Current vs conductance based I&F models

There are mainly two different types of leaky integrate-and-fire models, where the difference is how the synaptic input is modelled. In the first model the input is modelled directly as an input of current. The second possibility is that the input could be modelled as a change of the membrane conductance. This change would then result in a current input.

The current I_{E} to a neuron could be described as the sum of external current I_{ext} and current due to synaptic input, I_{syn},

$$I_{\mathrm{E}} = I_{\mathrm{ext}} + I_{\mathrm{syn}}. \tag{2.6}$$

I_{syn} could be described as a sum of current functions $i(t)$ or as a sum of conductance functions $g(t)$ multiplied with a voltage $V - E_{\mathrm{rev}}$. E_{rev} is a reversal potential which is different for each type of ion. The functions are weighted with a factor w. These two cases are shown mathematically as

$$I_{\mathrm{syn}}(t) = -\sum_{t_{\mathrm{sp}}} wi(t - t_{\mathrm{sp}})\Theta(t - t_{\mathrm{sp}}), \tag{2.7}$$

$$I_{\mathrm{syn}}(t) = -\sum_{t_{\mathrm{sp}}} wg(t - t_{\mathrm{sp}})(V - E_{\mathrm{rev}})\Theta(t - t_{\mathrm{sp}}). \tag{2.8}$$

The two equations describe $I_{\mathrm{syn}}(t)$ for a current based I&F model and a conductance based I&F model, respectively.

The sum is over all spike times. Θ is the Heavyside function. It is 0 when its argument is negative and 1 when its argument is positive or zero. When using $\Theta(t - t_{\mathrm{sp}})$ it is ensured that times before spike time t_{sp} should not count.

In NEST the conductance based model and the current based model are called iaf_cond_alpha and iaf_psc_alpha, respectively. There are four equations which describe the synaptic dynamics and one equation which describes the membrane potential. The equations for the synaptic dynamics are linear first orders differential equations.

In the case of iaf_psc_alpha, with current based input, the equation for the membrane potential is linear too. In the case of iaf_cond_alpha, with conductance based input, the equation becomes non-linear. This is caused by the product of conductance and membrane potential, $\bar{g}V$, in Eq. (2.2). The four equations that describe the synaptic input then create a linear subsystem. This gives opportunities for the way these variables are updated

and is used when creating new neuron models based on the iaf_cond_alpha neuron model (see Chapter 4).

The function which describes the conductance and current functions in these two models is an alpha function [iaf_psc_alpha.h, 2010]. Alpha functions, and beta functions, are explained by Rotter and Diesmann [1999]. The second-order differential equation given by Eq. (2.9) is called alpha-function for $a = b$ and beta-function for $a \neq b$.

$$\ddot{\eta} + (a+b)\dot{\eta} + (ab)\eta = 0, \;\; \eta(0) = 0, \;\; \dot{\eta}(0) = 0 \tag{2.9}$$

The explicit form of these equations are

$$\eta(t) = \dot{\eta_0} t \mathrm{e}^{-at}, \tag{2.10}$$

$$\eta(t) = \frac{\dot{\eta_0}}{b-a}\left(\mathrm{e}^{-at} + \mathrm{e}^{-bt}\right), \tag{2.11}$$

for the alpha and the beta function, respectively.

Chapter 3

Numerically integration of ODE's

The previous chapter concerned neurons and explained the biology of neurons and how to model them. These neuron models are necessary tools when trying to recreate nature by simulation. The mathematical description is done with ordinary differential equations.

This chapter concerns solving such equations on a computer. Section 3.1 gives an introduction to numerical integration. Section 3.2 is an overview of different types of methods that can be used for this purpose. The last two sections explain different methods that can be used for numerical integration. Different Runge-Kutta methods is explained in Section 3.3. Exact Integration is explained in Section 3.4.

This chapter has been written based on different sources. Section 3.2 is based on the book of Vandergraft [1983]. The books of Butcher [2008] and Press et al. [2007] are used in Section 3.3. Section 3.4 is written based on the article of Rotter and Diesmann [1999].

3.1 Introduction to numerical integration

Differential equations can be written as

$$y'(x) = f(x, y(x)). \tag{3.1}$$

$y'(x)$ means the derivative $\frac{\mathrm{d}y}{\mathrm{d}x}$. x and y are generally used to denote the time independent and time dependent variable, respectively [Butcher, 2008]. x is then often denoting time and y is the variable which changes over time.

The solution of some differential equations can be found analytically. For those where an analytical solution not is achievable, the differential equation is to be solved numerically with use of computers. The rest of this chapter considers solving this kind of differential equations.

The representation of differential equations in computers is done by difference equations. Instead of using the derivative $\frac{\mathrm{d}y}{\mathrm{d}x}$, the derivative are represented approximately by $\Delta y/\Delta x$. This is used with the assumption that the function is linear between the start point of time interval, x_1, and the end point of the time interval, x_2. In most cases this is not a valid assumption. Decreasing the interval of time, which means decrease Δx, increase the validity of the change of the derivative. However, because it only is an approximation there will always be an error when calculating a value numerically.

Eq. (3.1) shows how a difference Δy can be calculated from a difference Δx and the derivative $f(x, y)$.

$$\Delta y = f(x, y(x))\Delta x. \tag{3.2}$$

The aim of solving differential equations is to find out how y behaves as a function of x. When standing at time x_1 at place y_1, finding the new value y_2 is what is desirable. It is possible to calculate the derivative at y_1 based on future points. Standing at y_1 it is assumed that the $y(x)$ from y_1 to y_2 is linear with derivative equal to the derivative at y_1.

In these cases where x represents time, Δx is the time step for the integration. This is hereafter called the resolution and is denoted h.

Updating of y could look like

$$y_{\mathrm{k}+1} = y_{\mathrm{k}} + \Delta y \tag{3.3}$$
$$= y_{\mathrm{k}} + hf(x, y(x)). \tag{3.4}$$

Δx is here replaced by h. h is denoting the resolution of the integrating methods, or the length of the time steps taken. y_{k} is the value of y after k steps of size h. The initial value, $y_0 = y(x = 0)$, is the starting value of y and has to be given explicit. $f(x, y(x))$ is denoting a derivative, because of the equality in Eq. (3.1). $f(x, y)$ do hereafter represents the derivative.

This way of updating a variable y is called Euler's method and Fig. 3.1 shows it graphically.

Euler's method is not recommended for practical use. The method is not very accurate when compared to other, more advanced methods which run at

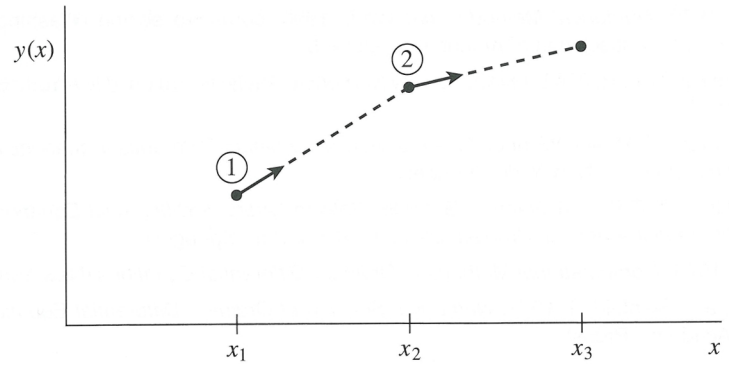


Figure 3.1: Euler's method [Press et al., 2007, Fig. 17.1.1]. The derivative at the starting point (1) of the interval is used to extrapolate the value at the ending point (2).

the equivalent step size. Because the behaviour of the function modelled only depends on the derivative at one point for each step taken, Euler's method is not stable either. Amongst several reasons to avoid Euler's method, these two may be the main reasons.

During the years many different methods have been developed. They are specialized for different purposes and for different types of differential equations. Some methods are fast, but very simple. Others can simulate the behaviour of the actual differential equation with good precision, but are less efficient.

Efficiency is important when solving differential equations numerically. The right hand side of the differential equation has to be evaluated many times during the simulation and, depending on the method used, many evaluations are also necessary during one time step h. Another important criterion is that the difference between the simulated values and the real solution should be as small as possible. Reducing h would in many cases give simulated values which are closer to the real solution.

Numerical solutions only give approximate values. Therefore it will always be a difference between the simulated value and the real value, which means that there will be an error. The real value is accessible in those cases where an analytical solution could be found. The difference between the approximate values from the simulation and the exact values, if an analytical solution is achievable, is an estimate of this error for the actual method for numerical integration. The error term also defines how good the method is.

The error term is denoted $O(h^n)$. Eq. (3.5) is identical with Eq. (3.4), except that the error term $O(h^2)$ is added.

$$y_{k+1} = y_k + hf(x, y(x)) + O(h^2) \tag{3.5}$$

There could be defined two types of errors which are local and global error. The local error is the error that occur in every single computational step. This means the difference between the simulated value and the real value, if accessible. The global error is the total error occured during a simulation. The local error depends on h^n and the global error depends on h^{n-1}.

Every type of problem has different criteria for what is preferred or needed. An error which is as small as possible desired in many cases, but not always. In some cases it is more important that the simulations deliver roughly correct results efficiently, in other cases the time used is nearly irrelevant because the results have to be as correct as possible. Depending on what is emphasized to be important, different methods should be chosen.

3.2 Different types of integrating methods

Implicit and explicit methods

Eq. (3.6) [Vandergraft, 1983, Eq. 8.27] is an example of an equation used in an implicit method. The variable for which we seek a value, in this case y_{k+1}, occurs on both sides of the formula. Because the new value of the variable depends on itself, the updating has to be done by iteration. This means to go through all possible values that y_{k+1} can take and test if this actual value fulfils the equation. If it does, the integration routine could move to step y_{k+1} and start over again.

$$y_{k+1} = y_k + \frac{1}{2}h(f(x_{k+1}, y_{k+1} + f(x_k, y_k)) \tag{3.6}$$

Implicit methods have two main advantages compared to explicit methods. In general they are more accurate than comparable explicit methods and the difference equations created are also often better conditioned than those created by explicit methods.

The main disadvantage of implicit models is their complexity. This is due to the fact that the variables are dependent of themselves when updating. Implicit methods therefore demand a lot of resources, as time and simulating power.

Eq. (3.7) [Vandergraft, 1983, Eq. 8.30] is an example of an explicit method. It is, said in a more specific way, the equation for the Runge-Kutta method of second order. The last term, $O(h^3)$, is an error term.

$$y_{k+1} = y_k + \frac{1}{2}h[f(x_k, y_k) + f(x_{k+1}, y_k + hf(x_k, y_k))] + O(h^3) \tag{3.7}$$

With explicit methods it is, as shown, possible to update variables based only on already known values. y_k is needed to calculate y_{k+1}, but y_{k+1} does not occur on both sides of the equation. Explicit models are, when compared to implicit methods, less complex.

When simulating neural networks it is important to measure the spike activity as correctly as possible. It is therefore not possible to use large time steps, which equals rough resolutions. The reason is that increasing the length of the time step will also increase the possibility of missing a spike. How this could happen is explained in the following.

V denotes the membrane potential for a neuron. After k time steps of length h, the time is $t_k = kh$. At time t_k the membrane potential $V < V_{\text{th}}$, where V_{th} is the threshold potential for spiking. Within the next time step, which means before time $t_{k+1} = (k+1)h = t_k + h$, the membrane potential of the neuron could have exceeded the threshold potential for spiking, V_{th}, and then decreased to a value lower than the threshold. At time t_{k+1} the membrane potentiale $V < V_{\text{th}}$, and no spike is sent out. But during the time step from t_k to t_{k+1} at least one spike should have been sent out. The threshold passing fluctuation of the membrane potential could have been registered if the length of the time step has been smaller. This would mean using a finer resolution h.

Other types of methods

Another way to split different integrating methods is to see whether they are one-step or multistep/n-step methods. A one-step method do only use $y(x_k)$ to obtain the approximate value for $y(x_{k+1})$. For $k > 0$, however, approximate values have already been calculated. In these cases it would be reasonable to use more than one value to create the next one. A method is called a n-step method if it uses n approximate and already calculated values for $y(x)$ to obtain $y(x_{k+1})$. If $n > 1$ the method could also be called a multistep method.

Runge-Kutta methods (see Section 3.3) are not considered as a multistep method, despite the fact that several test steps are taken during a time step h. This is because the value of $V_{\text{k+1}}$ only depends on V_{k} and test steps taken between V_{k} and $V_{\text{k+1}}$. It do no depend of values of V_{i} with $i < k$.

Adaptive solvers do not use the same step size all the time. Adjustments of the step size are done by the step size control. It checks how small steps it is necessary to take, or in other words how large steps that could be taken, to

describe the behaviour of the system well enough compared to some criteria. If the function used in the solver is smooth, big steps could be taken and the simulation could be done more efficiently. The maximal length of the time step is, however, determined by the user of the alghoritm. The adaptive solver can therefore not use as large steps as possible, but can reduce the length of the time step to a minimum.

3.3 The Runge-Kutta method

When using Euler's method the whole interval is taken in just one step. The information about the derivative is only used in the beginning of the step. This can give non-accurate results. A possible way to achieve better accuracy is by using the Runge-Kutta method, which is a modified version of Euler's method.

The Runge-Kutta method takes test steps inside a time step h. Approximate values of y, the time dependent variable, are calculated at the end points of these test steps. The updating of y, from y_{k} to $y_{\mathrm{k}+1}$, is done by giving the values from the test steps different weights. There are several versions of the Runge-Kutta method, and a main difference is how many test steps that are taken.

As for all methods for numerical integration, the Runge-Kutta method has errors. In general, a Runge-Kutta method is called an nth order method if its error term is $O(h^{n+1})$. This can be showed with a Taylor expansion of, for example, Eq. (3.15) around a point $(x_{\mathrm{k}}, y_{\mathrm{k}})$ [Vandergraft, 1983, Ch. 8.2.1]. The error from the Runge-Kutta method does therefore decrease with decreasing step size, which means finer resolution h.

The Runge-Kutta method of second order is the most basic one, and uses one test step in the middle of the interval. Information about the derivative at the midpoint of the interval is then used in addition to information about the derivative at the beginning of the step. Eq. (3.8)–(3.10) describes the way of computing an integrating step with the Runge-Kutta midpoint method. The last part of Eq. (3.10), $O(h^3)$, is an error term. Fig. 3.2 shows graphically the Runge-Kutta midpoint method.

$$k_1 = hf(x_n, y_n) \tag{3.8}$$

$$k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \tag{3.9}$$

$$y_{n+1} = y_n + k_2 + O(h^3) \tag{3.10}$$

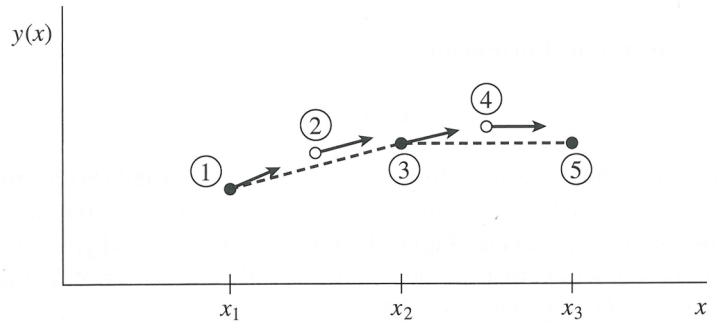


Figure 3.2: Runge-Kutta method of second order, also called the midpoint method. The derivative in the midpoint of the interval (at 2) is, together with the derivative at the starting point (1), used to extrapolate the value at the ending point of the interval (3) [Press et al., 2007, Fig. 17.1.2].

The Runge-Kutta method is, rather than Euler's method, preferred when selecting a method for numerical integration. The Runge-Kutta method of second order uses a test step at $h/2$ to get better results. Euler's method do not use test steps. But if Euler's method is used with step size $h/2$, would not that be equal to use the Runge-Kutta method of second order with step size h?

The actual Runge-Kutta method has an error term which depends on h^3. The error term in Euler's method depends on h^2. With step size $h/2$ it depends on $h^2/4$. This difference in the local error is the reason why the Runge-Kutta method of second order is preferred rather than Euler's method with half the step size.

Higher orders and the Fehlberg variant

Runge-Kutta methods could, as mentioned, be of different orders. Methods of higher orders use more test steps, and the result would be more correct than with use of lower orders‘ methods. More test steps in total means more calculations. The computational resources needed are therefore greater for the methods of higher orders than, for example, for the second order Runge-Kutta method.

The most common of the Runge-Kutta methods of higher orders, is the fourth order method. Eq. (3.11)–(3.15) is from Press et al. [2007] and describes how to compute this method. The last part of Eq. (3.15), $O(h^5)$, is an error term.

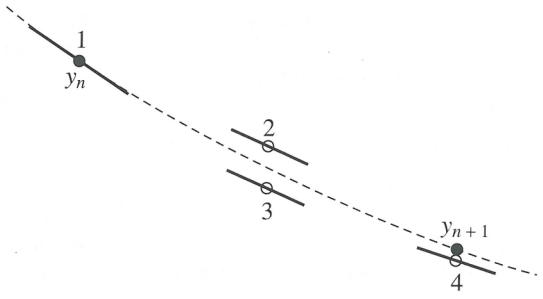


Figure 3.3: Runge-Kutta method of fourth order. The derivative at the starting point of the interval (1), twice at the midpoint of the interval (2 and 3) and at the endpoint of the interval (4), are used to extrapolate the value at the endpoint (4) [Press et al., 2007, Fig. 17.1.3].

$$k_1 = hf(x_n, y_n) \tag{3.11}$$

$$k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \tag{3.12}$$

$$k_3 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \tag{3.13}$$

$$k_4 = hf(x_n + h, y_n + k_3) \tag{3.14}$$

$$y_{n+1} = y_n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \tag{3.15}$$

Different simulations tolerate different errors. The error is depending on the resolution h. If an estimate of the local error had been accessible, this would have given important information. It would then be possible to change the resolution due to the tolerated error. A resolution which gives an error slightly smaller than tolereated will give the best result. It combines highest precision with the lowest computational cost. Using a finer resolution will decrease the error to a level lower than necessary. This will increase the computational costs. A less precise result is achieved with a rougher resolution.

In a multi step method this is possible with just one extra evaluation. In a Runge-Kutta method several extra evaluations are needed. Felberg introduced the idea to use two Runge-Kutta methods where some of the evaluations were used in both methods. It turns out that this is possible if the methods are of different orders. The local error in the lower order Runge-Kutta method is calculated when doing it like this. The most used of the Runge-Kutta-Fehlberg methods is the fourt-fifth method. This gives an estimate of the difference between the fifth and the fourth order method.

3.4 Exact Integration

Rotter and Diesmann [1999] introduced "an efficient method for the exact digital simulation of time-invariant linear systems"[1]. In such time-invariant systems the output, for example the membrane potential, is treated as to not be explicitly independent of time. Time is therefore representing the independent variable.

Eq. (3.16) could describe a time-invariant linear system [Rotter and Diesmann, 1999, Eq. (2)]. As for the other methods, y is the time independent variable and x represents time. Both y and x are vectors of dimension n. A is a quadratic matrix and its numerical constants defines the characteristics of the system.

$$\dot{y} = Ay + x \tag{3.16}$$

There are two variants of this equation, the homogenous one and the inhomogenous one. The homogenous equation has $x = 0$ and describes a system with zero input. The inhomogenous equation is as shown above and the corresponding system has input.

The matrix exponential e^{At} would have given the set of solutions if the equation was homogenous. This matrix is then termed "time-evolution operator or propagator" matrix. If the equation is inhomogenous the matrix exponential is called "the impulse response of the system".

It can be shown that the updating of variable y can be done as

$$y_{\mathrm{k}+1} = \mathrm{e}^{A\Delta} y_{\mathrm{k}} + x_{\mathrm{k}+1}, \tag{3.17}$$

where Δ is the resolution of the updating. This means that it is equal to the variable h used before, $\Delta = h$. This resolution determines the time grid of the updating. A condition for this equation to be valid is that input is restricted to this grid.

Exact Integration proved to deliver more accurate results on both subthreshold dynamics and spike-timing than the other integrating methods tested by Rotter and Diesmann [1999]. The tests were done with integrate-and-fire neurons. The fact that "in theory, deviations from the analytical solution do not occur, independently of the step size of the iteration" is perhaps the main advantage of this method.

The grid restricted input makes this method efficient and precise. On the other hand, this restriction may be the main disadvantage of this method.

[1] Quotes in this section are from Rotter and Diesmann [1999]

Within a time interval h some neurons have spiked. They have not spiked at exactly the same time, but since all the input is restricted to the time grid, it seems as though they have done. This could lead to artificial synchrony. If measuring the correlations between neurons in a network simulation, this synchrony can distort the measurements.

Morrison et al. [2007] pointed at this synchrony and developed a new method which ensures that the spikes are handled with their exact spike times. This method is implemented and used in NEST.

Chapter 4

Methods

NEST supports many different types of neuron models. Among them are implementations of both the current based and the conductance based leaky integrate-and-fire model (see Section 2.3). They are named iaf_psc_apha and iaf_cond_alpha, respectively. These two model types differ in the way that the synaptic input is described. In the iaf_psc_alpha model it is described as a current function and as a conductance function in the iaf_cond_alpha model.

Both models have five variables that need regularly updating. Four variables describe the synaptic input and the last one describes the membrane potential. The variables for the synaptic input are described by ordinary differential equations and create a linear system of equations. The differential equation for the membrane potential, Eq. (2.2), is linear in current based models and non-linear in conductance based models. The non-linearity is caused by the product of $\bar{g}V$ in the previously mentioned equation. The result of this difference is that the system of equations for the iaf_psc_alpha model in total is linear and that the corresponding system of equations for the iaf_cond_alpha model is linear except for the equation describing the membrane potential.

The variation of linearity for the two systems of equations makes it necessary to do the updating in two different ways. The iaf_psc_alpha model can, because of the linearity of the system, be updated with use of the exact integration routine. The iaf_cond_alpha model, on other hand, uses the Runge-Kutta-Fehlberg method of fourth-fifth order to updates all five parameters. This method is implemented in GNU Science Library[1] and has adaptive step size control.

Exact Integration can be implemented with use of just a few functions. Few

[1] http://www.gnu.org/softare/gsl/

functions mean that a small amount of time is used for function calls. The GSL solver is, on the other side, an external part. Many functions must therefore be called before the needed values can be calculated by GSL, and then the result has to be returned the opposite way back to the simulator. The time used for this operation will, compared to time used for the iaf_psc_alpha model, be larger.

The iaf_psc_alpha model is considered to be fast enough when simulating. Simulations with the iaf_cond_alpha model, on the other hand, takes longer time than desirable. The work done during this thesis was aimed to make simulations with the iaf_cond_alpha model faster.

The fact that the different models use different methods for numerical integration to update the variables is important when trying to reduce the simulation time for the iaf_cond_alpha neuron model.

The way to reduce the simulation time is to convert as much as possible of the calculations from use of GSL and over to Exact Integration. This means to use Exact Integration, instead of Runge-Kutta-Fehlberg method of fourth-fifth order from GSL, to update the variables which describe the synaptic input. These equations are linear and do not need to be updated with GSL. Since the equation for the membrane potential is non-linear it can not be updated with Exact Integration. The Runge-Kutta method must be used for updating of this variable.

With this conversion the simulation time would decrease, and therefore increase the computational efficiency of the simulator. This is an always existing goal when developing simulators. To convert other neuron models was another aim for this master thesis, but the time period was not long enough. Instead, two versions of the iaf_cond_alpha model, which are slightly different from each other, have been developed.

Testing of the new neuron models was also an important part of the work done during this master thesis. It is necessary to test both the simulation time and the numerical precision. This precision at the new models should be at least as good as with the original iaf_cond_alpha model.

This chapter presents the work done during this thesis and how it is done. Section 4.1 presents the neuron models in more detail than done in this introduction and Section 4.2 presents the methods used for testing of the neuron models.

The Python interface of NEST, PyNEST, has been used when testing the neuron models. An explanation of the functions used in this master thesis is given in Appendix A. The programming code for the new neuron models and the test scripts used for production of figures and data used in this master thesis are included in Appendix B.

Processor	Intel Core 2 Duo T6600, 2.2 GHz
Memory	4 GB RAM
OS	Linux Ubuntu 9.10, 32 bits
C++ Compiler	gcc version 4.4.1

Table 4.1: Specifications of test computer

This master thesis, including the testing done, has been produced on a computer with specifications as given in Tab. 4.1.

4.1 Creating neuron models

The new model iaf_cond_alpha_ei has been developed. To update the synaptic dynamics Exact Integration is used instead of GSL. This use of Exact Integration is indicated by the adding "_ei" in the model name.

The original plan was to use the GSL-solver to update the membrane potential. This routine takes test steps, inside one interval, at the lengths

$$S_{\text{gsl}} = \left[0, \frac{1}{4}, \frac{3}{8}, \frac{12}{13}, 1, \frac{1}{2}\right],$$

and in this order.

When updating the other variables, Exact Integration with constant step size would be used. The GSL version of Runge-Kutta-Fehlberg algorithm has, on the other hand, adaptive step size control. If combining Exact Integration and GSL there will therefore be possibilities that the approximate values are calculated at different times. All five variables have to be calculated at the same point in time, otherwise it would lead to wrong results. This excludes the combination of Exact Integration and solvers with adaptive step size control.

Instead of using GSL to update the membrane potential, the Runge-Kutta method of fourth order was implemented manually. Constant step size is used and it would therefore be possible to combine it with use of Exact Integration.

The GSL solver uses the Runge-Kutta-Fehlberg solver of fourth-fifth order. In the iaf_cond_alpha_ei model only the Runge-Kutta method of fourth order has been used. This difference led to the development of the iaf_cond_alpha_ei_step model.

The iaf_cond_alpha_ei_step model is based on the iaf_cond_alpha_ei model. The Runge-Kutta-Fehlberg method of the fifth order is used to calculate the

membrane potential. Constant step size is used. The increase from fourth to fifth order Runge-Kutta is indicated by adding "_step" to the model name.

This model has also implemented a calculation of error. This is a copy of how it is done in the GSL. The error is, in this case, the difference between the approximate values from the Runge-Kutta method of fourth order and the Runge-Kutta method of fifth order. To calculate this error the Runge-Kutta-Fehlberg method of fourth order is used in addition to the already used fifth order method. This error could be picked out with use of the PyNEST function `GetStatus` with keyword `'V_error'` (see Appendix A).

Updating the models

In the iaf_cond_alpha_ei model, the Runge-Kutta method of fourth order is used. The test steps are then taken at the start of the interval, twice halfway through the interval and at the end of the interval. The two steps in the middle create the derivative based on different values. Fig. 3.3 shows this method graphically.

The fact that two of the steps are at the middle of the interval makes the calculation a bit easier. Then the simulation only has to be brought forwards in steps of $h/2$. First from the start to the middle, and then from the middle to the end. The values of the synapses have to be calculated at the same points as the derivative. Updating of the synapses does therefore also happen in steps of $h/2$.

When values from all the four test steps have been calculated, the difference of membrane potential ΔV is calculated. The values from the different test steps are weighted differently according to the Runge-Kutta method.

The updating sequence during a time step of length h could in the iaf_cond_alpha_ei neuron model be described as follows. The variables k_x is the value of test step number x.

1. Time t=0 (beginning of time step):
   - Calculate k_1
   - Update synapses $h/2$ forwards in time
2. Time t=$h/2$ (middle of test step):
   - Calculate k_2
   - Calculate k_3
   - Update synapses $h/2$ forwards in time
3. Time t=h (end of time step):

- Calculate k_4
- Calculate ΔV and update V

The different elements of the propagator matrix used in the Exact Integration routine are calculated once. This calculation involves the synaptic time constant. Since this time constant is different for excitatory and inhibitory neurons the calculations are done twice. Because every updating step is of length $h/2$ there is only necessary to create six matrix elements for the exact integration, three for each of excitatory and inhibitory neurons.

In the iaf_cond_alpha_ei_step model the test steps are taken at the lengths of the interval given in S_{gsl} in Eq. (4.1). The updating of the synapses is done once in the end of each time interval, after the test steps have been taken. All the different test steps are taken with the start of the interval as a base. The values are created using corresponding length and elements in the propagator matrix. This could be showed as follows, where the variables k_x denote the value of the different test steps taken:

1. Time t=0 (beginning of time step):
   - Calculate k_1 at the base (t=0)
   - Calculate k_2 at time t=$(1/4)h$
   - Calculate k_3 at time t=$(3/8)h$
   - Calculate k_5 at time t=$(12/13)h$
   - Calculate k_6 at time t=h (end of time step)
   - Calculate k_4 at time t=$(1/2)h$
   - Calculate ΔV and update V
   - Update synapses h forwards in time

The updating in the step model differs a bit from the "_ei" model on this point. The change has been done to minimize numerical error. Since all calculations done in a computer increase the numerical error, it is desirable to make as few calculations as possible. The precision of values depend on the precision of the future calculations. All test steps are therefore taken with the same point as a base. The precision of the fifth and last test step depends therefore only on the numerical precision of the base.

The test steps could have been taken in increasing order and with the value from the previous test step as a base for the next one. The numerical precision of the fifth and last test step would then have been influenced by the precision of all the four previous steps. Taking all the test steps with the use of the same base therefore decrease the numerical error and increase the numerical precision.

The propagator matrix for both excitatory and inhibitory neurons is calculated once in the beginning of the simulation.

4.2 Testing neuron models

Four neuron models have been tested. They are

- Iaf_psc_alpha
- Iaf_cond_alpha
- Iaf_cond_alpha_ei
- Iaf_cond_alpha_ei_step

Four different test regimes have been used to test the precision of these neuron models. The first one is without output spikes and the second one is with extra produced noise from Poisson generators. The third one is with extra produced spikes from spike generators in NEST. The fourth regime is equal to the second, but with noise discretized to intervals of one millisecond. These regimes are in the test scripts respectively called

nsp Non-spiking regime with external current as only input.

wno Noise input from Poisson generators.

esp Extra excitatory spikes.

wfn As wno, but with noise input discretized to intervals of one millisecond.

Fig. 4.1 shows the behaviour of the membrane potential for all these types of regimes. These figures are produced with the iaf_cond_alpha neuron model and with resolution $h = 2^{-6}$. Graphs of membrane potential will look different for each model and with different parameters, but these figures give an impression of how the membrane potential develops through time in the different test regimes.

Exact values are achievable in the first regime, the non-spiking regime. Eq. (2.3) describes the behaviour of this system. Eq. (2.5) is the analytical solution of the mentioned equation. They are repeated here as Eq. (4.1) and (4.2), respectively. Exact values for the membrane potential are therefore achievable for every moment of the simulation. The precision of each neuron model, being the difference between exact and simulated values, is therefore possible to calculate.

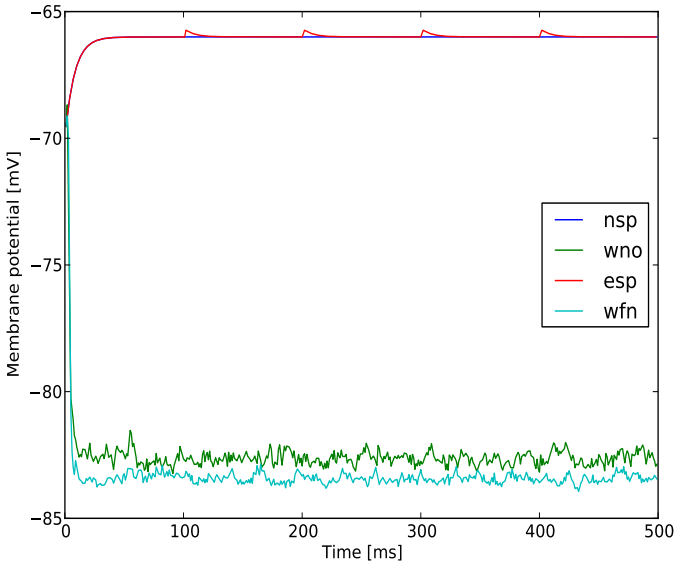


Figure 4.1: The behaviour of the membrane potential for the four different test regimes. The figures are produced with the iaf_cond_alpha neuron model with resolution 2^{-6} ms.

$$\tau_{\mathrm{m}} \frac{\mathrm{d}V}{\mathrm{d}t} = E_{\mathrm{L}} - V + I_{\mathrm{e}} R_{\mathrm{m}} \tag{4.1}$$

$$V(t) = E_{\mathrm{L}} + \frac{I_{\mathrm{E}} \tau_{\mathrm{m}}}{C_{\mathrm{m}}} (1 - \mathrm{e}^{-t/\tau_{\mathrm{m}}}) \tag{4.2}$$

The second test regime with extra produced noise is used to test the precision of the models when the membrane potential is fluctuating a lot. This extra noise comes from the Poisson process, which gives an approximation of neuronal firing. This process has proved "extremely useful" for this purpose [Dayan and Abbott, 2001].

A homogenous Poisson process is used in this master thesis. The firing rate for this process is constant over time and therefore time independent. The events in a Poisson process do not depend on the history of the system either, and every event is then statistically independent.

The noise from the Poisson process is one type of possible input to the neuron. This input comes through the synapse and is called synaptic input. This synaptic input can come from two types of neurons, excitatory neurons and inhibitory neurons. Their spikes influence the membrane potential in opposite directions. Inhibitory spikes decrease the membrane potential

Description	Value
Excitatory rate [Hz]	80 000
Inhibitory rate [Hz]	20 000
Excitatory connection strength	15.0/70.0
Inhibitory connection strength	-4.0
Delay [ms]	1.0

Table 4.2: Standard values for Poisson generators used for production of noise to the with-noise regime (wno).

and excitatory spikes increase it. The possibility of firing a spike therefore decreases when a neuron receives a spike from an inhibitory neuron. Receiving a spike from an excitatory neuron would give the opposite outcome and increase the possibility of firing a spike.

Inhibitory neurons are normally modelled as four times as powerful as excitatory neurons. To make the network simulation so that the inhibitory and the excitatory neurons influence equally, four times as many excitatory neurons as inhibitory neurons are being used. This can be seen, for example, in the network used for testing of speed (see speed_brunel.py in Appendix B).

When using Poisson generators it is possible to adjust both the firing rates, the strength of the connection with the neuron and the delay of the transmission of the signal. The standard values of these parameters are showed in Tab. 4.2. The delay could be zero millisecond, but are of technical reasons set to one millisecond.

When using the rates as described in Tab. 4.2, the frequency of the spike generators is at a level which equals a network of 10^4 neurons each firing at 10 Hz. It has also been done stress tests of the neuron models with total rates from 1 kHz to 750 kHz. This equals, in the same network with 10^4 neurons, a firing rate for each neuron varying from 0.1 Hz to 75 Hz. Stress tests with different synaptic time constants have also been done.

The third test regime was created to test how the different neuron models, and at different resolutions h, manage the sudden changing of the membrane potential which a spike causes. Tests of the esp regime has not been emphasized in this master thesis.

The fourth test regime is based on the with-noise regime, but all the Poisson input are discretized to intervals of one millisecond. Every simulation then receive the same input independent of resolution. The results presented in Section 6.2 are based on simulations done with the with-frozen-noise test regime.

Description	Parameter	Value
Leak reversal potential [mV]	E_{L}	-70
Threshold potential [mV]	V_{th}	-55
Reset potential [mV]	V_{reset}	-60
Leak conductance [mS]	g_{L}	15.0
Input current [pA]	I_{E}	4.0*g_{L}
Membrane capacitance [pF]	C_{m}	120.0
Membrane time constant [ms]	τ_{m}	$C_{\mathrm{m}}/g_{\mathrm{L}}$
Exc. synaptic time constant [ms]	$\tau_{\mathrm{syn,E}}$	0.2
Inh. synaptic time constant [ms]	$\tau_{\mathrm{syn,I}}$	2.0

Table 4.3: Parameter values for the neuron. Descriptions and units are from iaf_cond_alpha.h [2010].

Resolutions ($\log_2$ h)	[-2, -4, -6, -10, -14]
Repetitions per resolution	3

Table 4.4: Resolutions and the number of repetitions used in the speed test. This test measures the simulation times for the neuron models.

The neuron models have been tested in several different ways, and some examples are given. Differences between analytical and simulated values have been investigated. How the error from the iaf_cond_alpha_ei_step model develops with resolutions has given indications of how reliable the new models are compared to for example the original iaf_cond_alpha model.

The parameter values for the neurons used for testing can be found in Tab. 4.3. These parameters have been the same in all the tests, both in the tests of precision and in the speed tests. All handling of values of parameters is done in the file class_Params.py (se Appendix B).

The test script that does the simulation uses functionalities of PyNEST, NEST's Python interface. These functions are therefore showed and explained in Appendix A.

Testing of speed

The work done during this thesis was aimed to decrease the time used when simulating with the iaf_cond_alpha neuron model. The numerical precision should be kept as good as it was with the original model. Decreased simulation time means increased computational efficiency, which is desirable.

Since two new versions of this model have been created, and the updating done with Exact Integration is based on how it is done in the iaf_psc_alpha

Variables	g	eta
Current based	19.05	6.75
Conductance based	2.0	0.0015

Table 4.5: The two variables g and eta are used to adjust the firing rate. In the speed test it is important to keep the firing rate constant. These two sets of values keeps the firing rate at around 30 Hz. Tested with resolution $h = 2^{-4}$ ms.

model, all four models have been tested. These four models are

- Iaf_psc_alpha
- Iaf_cond_alpha
- Iaf_cond_alpha_ei
- Iaf_cond_alpha_ei_step

The testing of speed has been done with the scripts comparision.py and speed_brunel.py (see Appendix B). The script speed_brunel.py is based on the test script brunel-alpha-numpy.py found in the NEST source code. The speed test is done by simulating the network given in brunel-alpha-numpy.py and measure the time used.

The speed test makes three repetitions of the network with different resolutions. Resolutions $h = [\,2^{-2}, 2^{-4}, 2^{-6}, 2^{-10}, 2^{-14}\,]$ are used. Different resolutions are used to get an impression of how the simulation time is influenced by the resolution. Tab. 4.4 shows graphically the repetitions and resolutions used in the two different speed tests.

The firing rate, both excitatory and inhibitory, for all models should be almost equal. If they are not, the simulation is not done with the same network with the same activity. The spike handling is the part of NEST which uses the most time, so it is important to keep the same firing rate if the simulations' times should be comparable.

The firing rates could be adjusted by the parameters g and eta in speed_brunel.py. g is the ratio between the strength of the inhibitory connection and the strength of the excitatory connection. eta is the ratio between the external rate of input and the threshold rate of input which is sufficient to make the neuron spike [Brunel, 2000].

The variables g and eta must be adjusted differently for current based and conductance based integrate-and-fire models. Since the iaf_psc_alpha model is the only current based model, and the three others are conductance based models, just two sets of parameters have to be found. The values used are shown in Tab. 4.5. Tested with resolution $h = 2^{-4}$ ms they adjust the firing rates to around 30 Hz.

Chapter 5

Stable states

During this master thesis a new numerical phenomenon has been discovered. It is seen in the non-spiking test regime. These errors are caused by decreasing precision in the numerical representation of numbers. This chapter will show what has been seen and give the explanation of this phenomenon.

5.1 Resolution dependent differences

The first test regime is a non-spiking regime with an external current as only input. The equation which describes the membrane potential in this case is presented in Dayan and Abbott [2001]. The equation is

$$\dot{V} = -\frac{V - E_{\mathrm{L}}}{\tau_m} + \frac{I_{\mathrm{E}}}{C}. \tag{5.1}$$

The solution of this equation is

$$V(t) = E_{\mathrm{L}} + \frac{I_{\mathrm{E}}\tau_{\mathrm{m}}}{C_{\mathrm{m}}}(1 - \mathrm{e}^{-t/\tau_{\mathrm{m}}}). \tag{5.2}$$

This is the same equation as Eq. (2.5) in Section 2.3.

The membrane potential described by Eq. (5.2) would evolve exponentially towards its asymptotic value. Fig. 4.1 shows this. After the membrane potential has reached a stable value close to its asymptotic value, its derivative $\dot{V} = 0$. This makes it easy to compute the asymptotic value of the membrane potential.

$$V = \frac{I_{\mathrm{E}}\tau_m}{C} + E_{\mathrm{L}} \tag{5.3}$$

In a regime with the current I_{E} as only input, Eq. (5.3) will give the exact value of the asymptotic membrane potential. Exact values for all measuring points could, in the same regime, be calculated with Eq. (5.2).

The fact that analytical values are achievable makes it possible to determine the difference between the simulated values and the analytical ones. This will say something about the precision of the simulated values. Because the simulation tries to recreate nature, the difference between simulated and analytical values should have been zero. It is not.

When the membrane potential evolves towards its asymptotic value given as Eq. (5.3), there are some differences between the analytical and the simulated values. They are caused by too rough resolution. These kinds of differences are treated in Chapter 6.

Differences caused by rough resolution can almost be removed if the resolution chosen is fine enough. But there will always be some unavoidable differences between the exact values of the membrane potential and the simulated values. Such differences increase with increasing fineness of the resolution. Figure 5.1 shows these differences.

The values, which are plotted, are therefore

$$\Delta V = V_{\mathrm{sim}} - V_{\mathrm{exact}}, \tag{5.4}$$

where V_{sim} is the simulated value of the membrane potential and V_{exact} is the exact value of the analytical solution given in Eq. (5.2). The different lines in the graph represent different resolutions h for the simulation.

At some point about $t = 150$ ms the difference starts to oscillate. The first one to start oscillating is the simulation with the finest resolution (smallest time step), and the last one to start is the simulation with the roughest resolution (largest time step). This one is also the one to oscillate the least of all.

When graphs are starting to oscillate, the amplitude grows larger and larger. At some point the difference has found a steady state value. From here on the difference between exact and simulated, or approximated, value is constant. This steady state solution seems to be more stable than the correct solution where the difference between exact and simulated values are zero.

How much the graph oscillates and what the value of this steady state difference becomes, depends on the resolution of the simulation. Increased fineness of the resolution h gives more oscillations and a larger steady state difference.

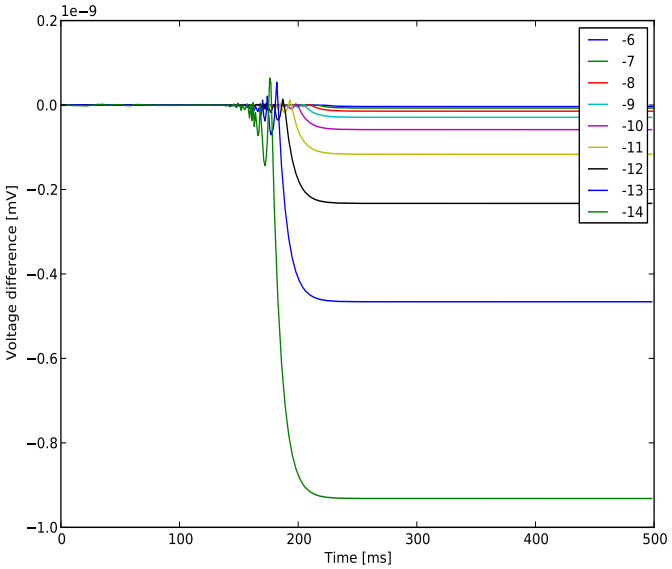


Figure 5.1: Differences between analytical and simulated values do occur. This figure shows schematically how these differences depends on the resolution used in the simulation. Different lines indicate different resolutions and increasing fineness of the resolution give increased difference. Simulations are done with iaf_cond_alpha neuron model with resolutions 2^{-6} to 2^{-14} ms.

The steady state difference will be, as shown in Figure 5.1, from the point where the graph reaches its steady state and to the end of simulation. Simulating in 500 ms or 5000 ms gives the same difference, and the difference is constant for about 500-200= 300 ms or 5000-200 = 4800 ms, respectively.

These oscillations also occur both for neuron models which use Exact Integration and for those who use the Runge-Kutta method to calculate the approximate values. This is an indication that the described behaviour not can be explained based on the algorithm used, and that the error is depending on something numerically more general.

Calculating the error

Eq. (3.11)–(3.15) are from Press et al. [2007] and describe how to calculate the value y_{n+1} by using the Runge-Kutta method of fourth order. Eq. (5.1) was used as the function $f(x, y)$. The resting potential of the cell, E_{L}, was set to zero for simplicity. Eq. (5.1) was written as

$$f(V) = \dot{V} = \lambda V + \beta, \tag{5.5}$$

with

$$\lambda = -\frac{1}{\tau_{\mathrm{m}}} \text{ and } \beta = \frac{I_{\mathrm{E}}}{C}. \tag{5.6}$$

The change in membrane potential, ΔV, is given as

$$\Delta V = \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \tag{5.7}$$

when using the Runge-Kutta method of the fourth order.

The software Mathematica[1] was used to obtain an explicit equation for ΔV. The FullSimplify-command was used and the result was

$$\begin{aligned}\Delta V =& \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \\ =& \frac{1}{24}h(\beta + V\lambda)(24 + h\lambda(12 + h\lambda(4 + h\lambda))) \\ =& (h\beta + \frac{h\lambda}{24}(12 + 4h\lambda + h^2\lambda^2)) \\ &+ V\left(\frac{h\lambda}{24}(24 + 12h\lambda + 4h^2\lambda^2 + h^3\lambda^3)\right)\end{aligned} \tag{5.8}$$

Updating of the membrane potential with the Runge-Kutta method of the fourth order is done as in Eq. (3.15). Combined with Eq. (5.7) and excluded the error term $O(h^5)$ it is

$$V_{n+1} = V_n + \Delta V. \tag{5.9}$$

If combining Eq. (5.8) and (5.9) the new value of V_{n+1} could be written as

$$V_{n+1} = a + bV_n, \tag{5.10}$$

with constants

$$a = h\beta + \frac{h\lambda}{24}(12 + 4h\lambda + h^2\lambda^2), \tag{5.11}$$

[1] http://www.wolfram.com

and

$$b = 1 + \frac{h\lambda}{24}(24 + 12h\lambda + 4h^2\lambda^2 + h^3\lambda^3), \tag{5.12}$$

based on Eq. (5.8).

As $V \to V_\infty$, the membrane potential will reach a fix point. At this point, and further on,

$$y_{n+1} = y_n. \tag{5.13}$$

The equation at this point, which corresponds to Eq. (5.10), is

$$V^\star = a + bV^\star. \tag{5.14}$$

The fix point value of the membrane potential, $V^\star$, could then be written as

$$V^\star = \frac{a}{1-b} \tag{5.15}$$

An exact representation of the constants a and b is possible with, for example, a fraction with integers in both numerator and denominator. However, a and b will be represented in a double data type in a computer. This data type has only 53 bits of space, which means 16 decimals. The result is that a and b not are represented exact any longer. The corresponding constants to a and b are $\bar{a}$ and $\bar{b}$. They are as equal to a and b as possible when cut to fit into a double data type.

The exact representation of the fix point value of the membrane potential depends on a and b, as given in Eq. (5.15). The fix point value achievable in a computer, $\bar{V}^\star$, will therefore depend on the new constants $\bar{a}$ and $\bar{b}$ as

$$\bar{V}^\star = \frac{\bar{a}}{1-\bar{b}}. \tag{5.16}$$

The difference between the exact fix point value and the fix point value possible in a computer could then be defined as

$$\Delta V^\star = V^\star - \bar{V}^\star. \tag{5.17}$$

When $h \to 0$, which equals reducing the length of the time step, it could be shown, based on Eq. (5.6), (5.11) and (5.12), that $a \to 0$ and $b \to 1$. This influences $\Delta V^\star$as well.

h [ms]	$\Delta V^\star = V^\star - \bar{V}^\star$	
2^{-6}	$\frac{57}{374\,933\,703\,564\,416}$	1.52027×10^{-13}
2^{-8}	$\frac{683}{1\,125\,625\,073\,669\,632}$	6.06774×10^{-13}
2^{-10}	$\frac{1\,707}{1\,125\,831\,190\,162\,432}$	1.51621×10^{-12}
2^{-12}	$\frac{5\,467}{1\,125\,882\,727\,149\,568}$	4.85575×10^{-12}
2^{-14}	$\frac{43\,691}{2\,251\,791\,223\,783\,424}$	1.94028×10^{-11}

Table 5.1: Differences between exact and numerically possible fix point value for different resolutions h in the non-spiking regime. The differences are shown both as fractions and decimal numbers.

Values given as

$$\tau_\mathrm{m} = 8,\ C = 120,\ I_\mathrm{E} = 60,\ \text{ and } E_\mathrm{L} = 0,$$

was inserted into Eq. (5.10). Both $V^\star$, $\bar{V}^\star$ and $\Delta V^\star$ were calculated for different resolutions. The results of $\Delta V^\star$, given both as a fraction and decimal number, for different values of the resolution h are shown in Tab. 5.1.

The values in this table show that decreasing step size results in increasing difference between exact and numerically possible fix point value, an effect already seen in Fig. 5.1. The explanation of the phenomenon seen in this figure is therefore as described above.

Chapter 6

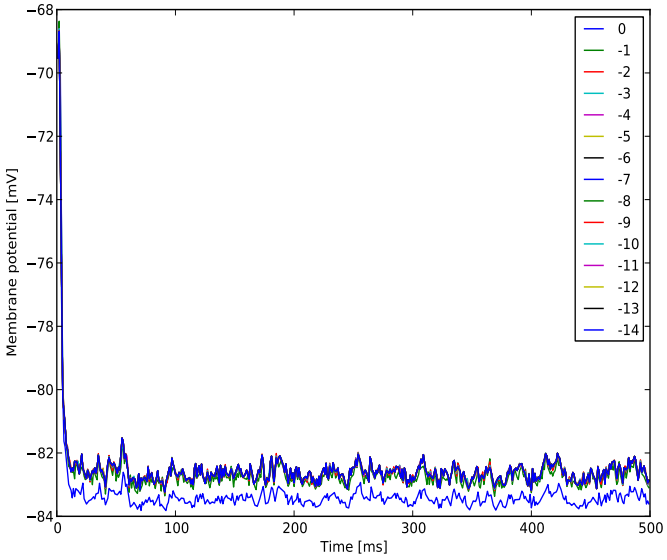
Precision and error

The precision of the neuron models have been tested in different ways. Some general aspects of simulating with large step sizes are dealt with in Section 6.1. Section 6.2 concerns simulations with external input from Poisson generators. This is a realistic system and recommendations for resolutions can therefore be given. The membrane potential of the current based leaky integrate-and-fire (I&F) model iaf_psc_alpha has been updated in two different ways. The difference and its result are explained in Section 6.3. The effects of varying rates from the Poisson generators or varying synaptic time constants are treated in Section 6.4.

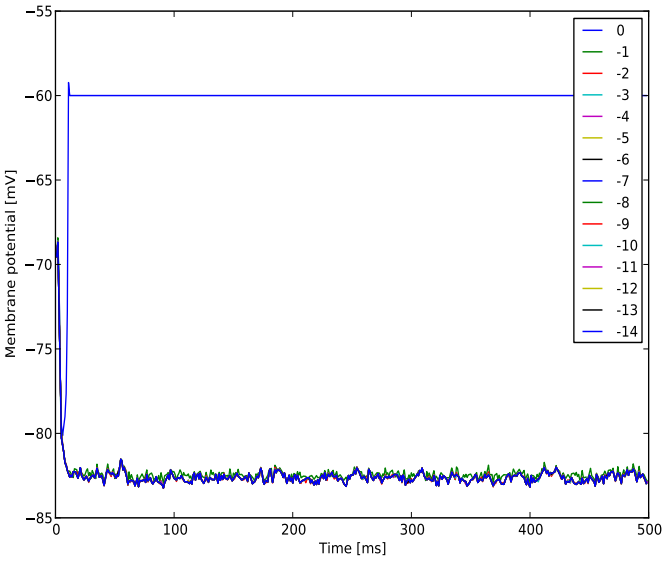
6.1 Rough resolutions

The main difference between the original iaf_cond_alpha neuron model and the new developed models iaf_cond_alpha_ei and iaf_cond_alpha_ei_step is the change of method for numerical integration. The use of adaptive step size control in the iaf_cond_alpha model results in good precision also for large step sizes (rough resolutions h). The new models use constant step size, independent of the fluctuations some eventual synaptic input causes. If there are many variations in the membrane potential, as it is in the with-noise test regime, this constant step size can lead to fatal errors.

Fig. 6.1 shows the membrane potential in the with-noise regime for all three versions of the iaf_cond_alpha neuron model. The line which belongs to resolution $h = 2^0$ ms for the new models should be mentioned. The variation of the membrane potential in these cases is not as it is for finer resolutions. A schematic view of how the membrane potential in the with-noise test regime should vary for a iaf_cond_alpha neuron model is shown in Fig. 4.1.

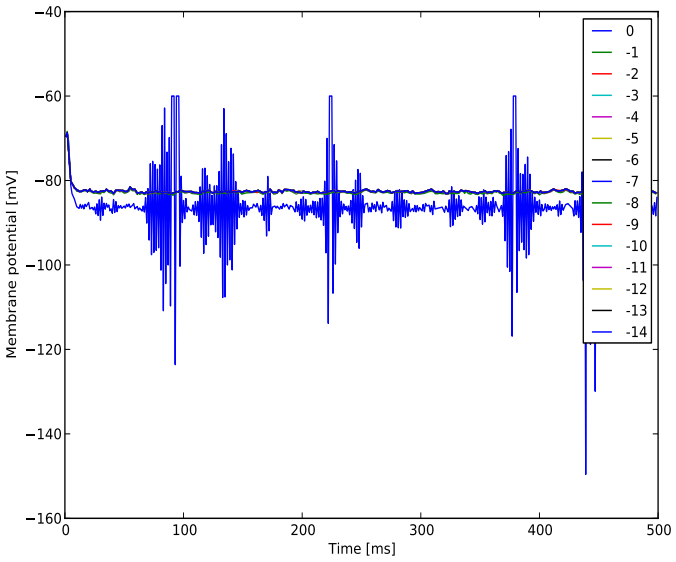


(a) Iaf_cond_alpha



(b) Iaf_cond_alpha_ei

Figure 6.1: Fluctuations of the membrane potential for the three different cond_alpha neuron models in the with-noise regime. The figures show how wrong results the resolution $h = 2^0$ ms gives. All figures show resolution $h = 2^0 - 2^{-14}$ ms. Fig. 6.1c shows the _ei_step model.



(c) Iaf_cond_alpha_ei_step

Figure 6.1: (Continued) Fluctuations in the with-noise regime for the iaf_cond_alpha_ei_step neuron model. All resolutions $h = 2^0 - 2^{-14}$ ms are plotted.

For the iaf_cond_alpha_ei model (Fig. 6.1b) some input in the first seconds of the simulation causes a big jump of the membrane potential. It jumps to $V = -60$ mV, which is the reset potential for the neuron. The membrane potential keeps this value during the rest of the simulation.

The membrane potential in the iaf_cond_alpha_ei_step neuron model (Fig. 6.1c) fluctuates a lot instead of being stable as seen with the iaf_cond_alpha_ei model. The membrane potential drops to about -85 mV during the first seconds of the simulation. Throughout the simulation there are a lot of fluctuations with a lower limit less than -100 mv and an upper limit at the reset potential. The potential reaches this upper limit just a few times during the 500 ms of simulation. This seems to happen in the middle of an interval with many fluctuations.

The iaf_cond_alpha neuron model also shows some unexpected behaviour at the simulation with $h = 2^0$ ms. The line representing this resolution in Fig. 6.1a lies some millivolts lower than the lines for the finer resolutions. This equals decreased length of the time step.

Based on the behaviour of the membrane potential in the with-noise regime, shown in Fig. 6.1, it is recommended not to use resolution $h = 2^0$ ms for the iaf_cond_alpha_ei model and the iaf_cond_alpha_ei_step model.

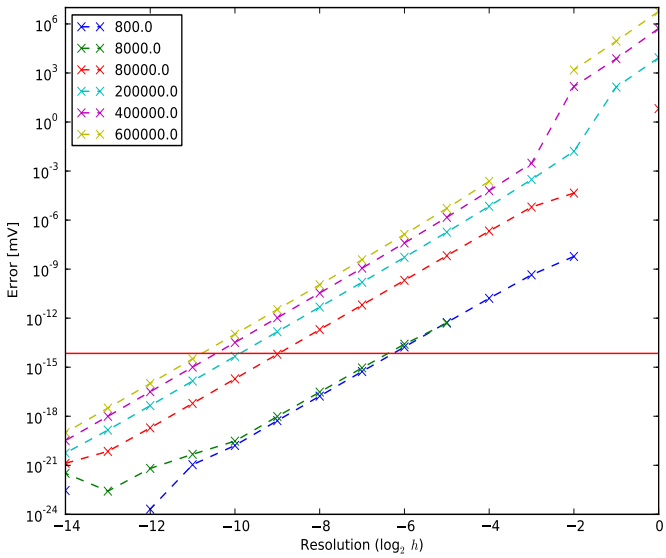


Figure 6.2: The difference between the Runge-Kutta-Fehlberg method of the fifth order and the fourth order is plotted against resolution. The plotted value is the median value for all measuring points. The simulation is done with the with-frozen-noise regime (wfn) and the different lines indicate different rates used for the Poisson generators. The legend gives the excitatory rate and the inhibitory rate is one fourth of the excitatory. The red line indicates when the error, given the values used, has reached machine precision. The line is plotted at $y = ||E_{\mathrm{L}}||10^{-16}$.

Another aspect which makes it unwise to use large time steps is the possibility of missing a spike. This is due to threshold passing fluctuations of membrane potential inside a time step of length h. Different number of spikes registered will mean that the behaviour of the system simulated will be different, and that is not desirable. A more detailed explanation is given in Section 3.2.

6.2 Realistic simulations

In the with-frozen-noise test regime the spike trains from the Poisson generators are discretized to intervals of one millisecond. This ensures that every spike is handled at exactly the same time of the simulation. The same input is therefore given to the system independent of resolution h.

Data from simulations of the with-frozen-noise regime are used to create Fig. 6.2. It shows the local error calculated in the iaf_cond_alpha_ei_step

model. Every cross is the median value of this error over all measuring points for a given resolution. The different lines represent different rates used in the Poisson generators, and the excitatory rate is given in the legend. The inhibitory rate is one fourth of the excitatory rate. Crosses produced with the same rate, but with different resolutions, are connected with dotted lines. The different rates are plotted in different colours.

The red line in this figure indicates when the error, given the values used in the actual simulated neuron, has reached machine precision. The value equals the product $||E_{\mathrm{L}}|| \times 10^{-16}$.

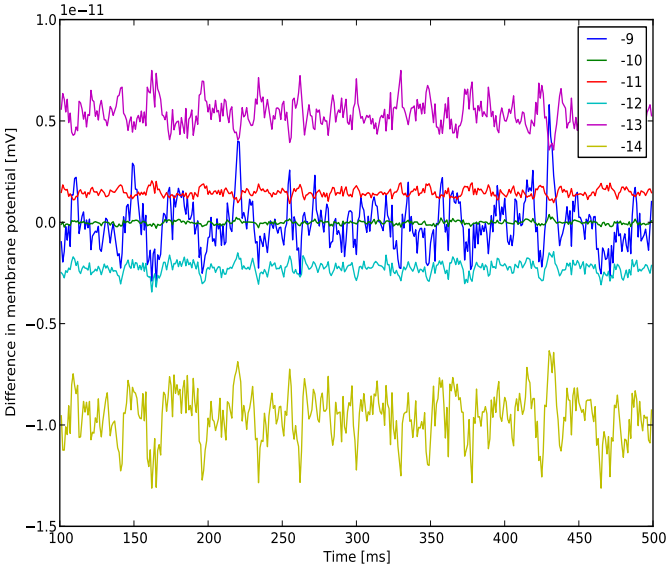
At some resolution the error is equal to the possible machine precision. Decreasing the step size further will only increase the error. The reason is as follows.

There will always be numerical errors caused by numerical calculation. These errors grow linear with the resolution h. The error from the numerical integration routine decreases, in this case, with h^6. This is because a Runge-Kutta-Fehlberg method of fifth order is used to calculate the value of the membrane potential.

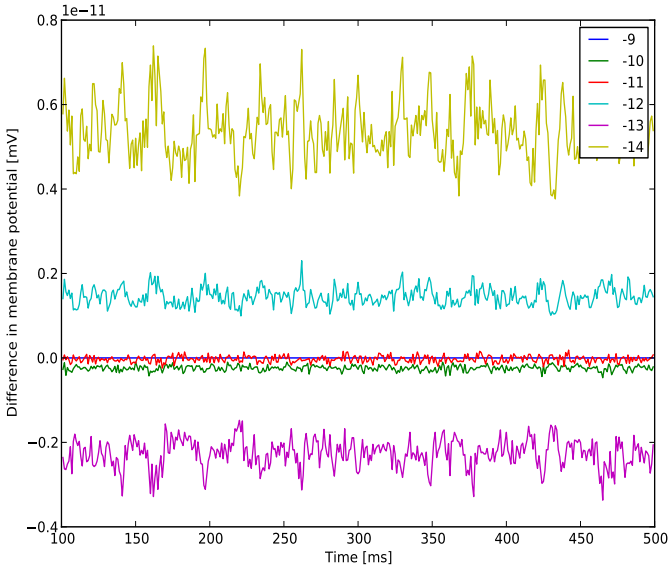
The total error equals the sum of errors from both numerical calculation and numerical integration. When the error from the numerical integration routine has reached the possible machine precision, given the values of the parameters used, it is not possible to reduce this further. Until now the development of the total error has been dominated of the error from numerical integration. Decreased error from the numerical integration has caused decreased total error. Since it is not possible to reduce the error from the numerical integration further, decreasing the step size, which equals using a finer resolution, will increase the total error. This is due to the increase of error from numerical calculations.

The standard rate used in the other simulation is the line marked with red. For these firing rates the optimal resolution is reached at $h = 2^{-9}$ milliseconds. At smaller rates the minimal total error will be reached when simulating with larger time steps, and higher rates need finer resolutions (smaller time steps) to achieve minimal total error.

Values from the iaf_cond_alpha_ei_step neuron model at resolution $h = 2^{-9}$ ms is used as a reference solution in Fig. 6.3. The difference in membrane potential between this reference solution and the three conductance based models are plotted in this figure. The iaf_cond_alpha_ei_step model is included in these three models. The simulations are done in the with-frozen-noise regime. The interesting part of the simulation is when the membrane potential is fluctuating around approximately -83 mV, and not the fast decrease in membrane potential in the first part of the simulation (see Fig. 4.1). Values for $t < 100$ ms are therefore not plotted.

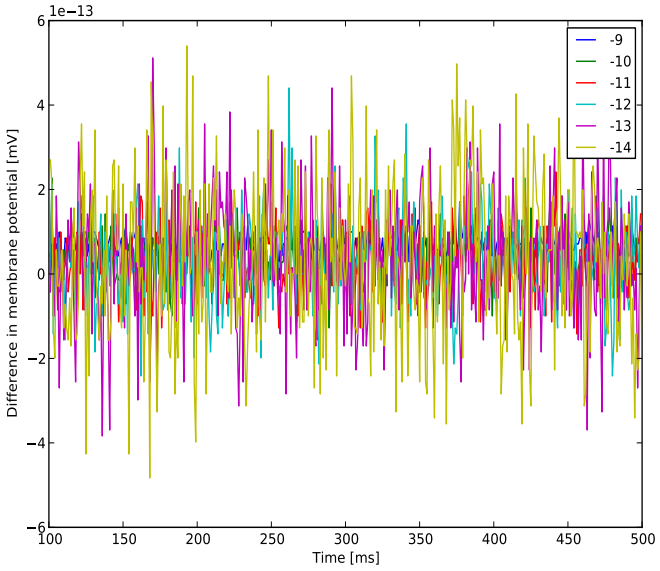


(a) Iaf_cond_alpha_ei, $h = 2^{-9}$–2^{-14} ms



(b) Iaf_cond_alpha_ei_step, $h = 2^{-9}$–2^{-14} ms

Figure 6.3: Iaf_cond_alpha_ei_step at resolution $\log_2\ h = -9$ is considered the most exact simulation. These figures show the difference in membrane potential between this simulation and the three conductance based models for resolutions $h = 2^{-9}$–2^{-14} ms. Simulations are done with the with-frozen-noise test regime.



(c) Iaf_cond_alpha, $h = 2^{-9}$–2^{-14} ms

Figure 6.3: (Continued) Difference in membrane potential between the simulation with iaf_cond_alpha_ei_step at resolution $h = 2^{-9}$ ms and the iaf_cond_alpha model at resolutions $h = 2^{-9}$–2^{-14} ms.

Three interesting phenomena have been observed in Fig. 6.3. These phenomena are commented, but an explanation of them has not been found.

The first phenomenon is the difference in order of magnitude of the differences. The differences for the original iaf_cond_alpha model have order of magnitude 10^{-13} mV. This is about 30 times smaller than the differences for both of the new models.

The difference in amplitude for the different resolutions is a second observation. It could be seen in all three subfigures that the finest resolution, $h = 2^{-14}$ ms, has the largest amplitude. It is, however, easiest to see it in Figure 6.3a and 6.3b. The amplitude decreases when increasing the length of the time steps. This means also a resolution more equal to the resolution used in the reference solution, $h = 2^{-9}$ ms. This might influence. It is also seen that the iaf_cond_alpha model fluctuate around zero for all resolutions plotted (Fig. 6.3c). For the other models, lines for different resolutions fluctuate around different values (Fig. 6.3a and 6.3b).

The third phenomenon could be seen in Fig. 6.3a and 6.3b in connection. These figures show the new models, respectively the iaf_cond_alpha_ei model and the iaf_cond_alpha_ei_step model. In both figures the mean value

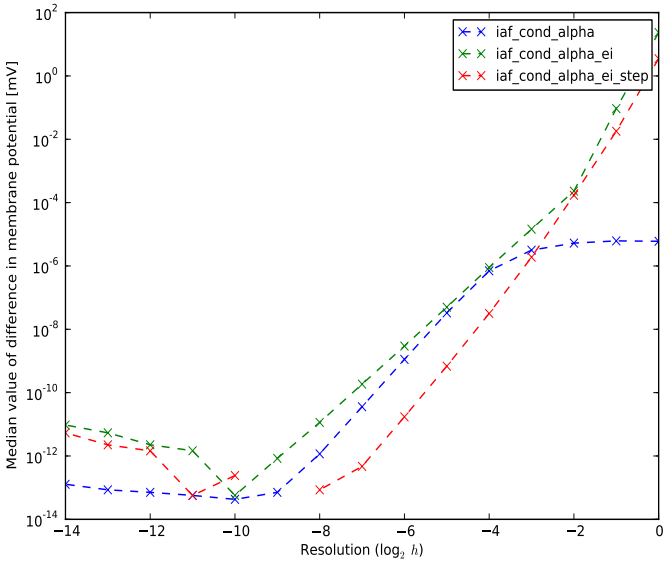


Figure 6.4: Differences between the membrane potential at $h = 2^{-9}$ ms for iaf_cond_alpha_ei_step model and different resolutions at the three conductance based models. The values plotted are the median value of the absolute value of the difference, taken over all measuring points.

of difference changes sign when resolution changes from one to the next. However, the mean value of difference has the opposite sign for these two models when comparing the same resolution. The line for $h = 2^{-14}$ ms is negative in Fig.6.3a and positive in Fig. 6.3b. For resolution $h = 2^{-13}$ ms it is opposite. This behaviour could be seen for at least the finest resolutions $h = 2^{-12}$–2^{-14} ms.

Fig. 6.4 is based on the same simulations as Fig. 6.3. The rates from the Poisson generators are 80 000 Hz excitatory and 20 000 Hz inhibitory.

This figure shows the difference between the reference solution (iaf_cond_-alpha_ei_step at resolution $h = 2^{-9}$ ms) and all three conductance based models for resolution $h = 2^{0}$–2^{-14} ms. For every model the median value over all measuring points is computed for each resolution. These values for the three neuron models are plotted.

Fig. 6.4 shows that, for time steps larger than $h = 2^{-4}$ ms and compared to the new models, the difference for the original iaf_cond_alpha model is quite small. The difference between having adaptive step size control and using constant step size influences here. For large time steps this control reduces the length of the time steps. This gives better precision.

For time steps smaller than $h = 2^{-9}$ ms the original model has smaller differences too. The step size control can not increase the length of the step size further than the step size given by the user. The reason why the original model is better also for small time steps is therefore unknown.

Fig.6.4 also shows something which could be seen in Fig. 6.3a and 6.3b as well. For the new models, the general development for time steps smaller than $h = 2^{-9}$ ms is an increase of the difference. The decision to choose $h = 2^{-9}$ ms as a reference solution is strengthened by this.

The iaf_cond_alpha_ei_step model at resolution $h = 2^{-9}$ ms was chosen as the reference solution in this testing. Simulations with other neuron models and different resolution could of course have been taken as a reference solution. They could have been used instead of the chosen solution, in addition to the chosen solution or perhaps both. The results would hopefully, and probably, have shown almost the same as with the used reference solution.

6.3 Updating algorithms

In contrast to the three versions of the iaf_cond_alpha neuron model, the numerical integration of the iaf_psc_alpha model is most precise at the largest time steps. The fact that large time steps increase the possibility of missing spikes is not taken into account. In the conductance based neuron models the updating of the membrane potential has to be done with the use of, for example, a Runge-Kutta method. This is because of the non-linearity of its equation. In the current based model iaf_psc_alpha, the membrane potential is described by a linear equation. Exact Integration could therefore be used for updating.

Fig. 6.5 shows differences between exact and simulated values for the iaf_psc_alpha model for resolutions $h = 2^0$–2^{-14} ms. The difference between the two subfigures lies in the way the updating of the membrane potential is done.

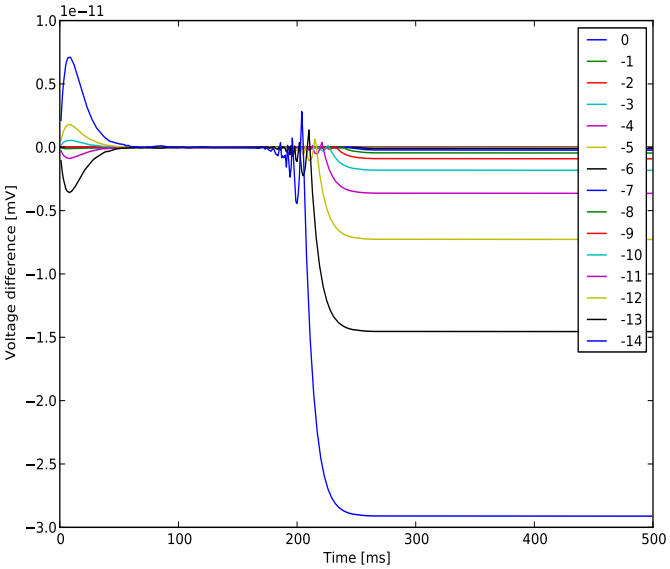
In the upper figure (Fig. 6.5a) the updating is done like this,

$$V_{\mathrm{k}+1} = I_{\mathrm{E}} + \mathrm{e}^{-Ah} V_{\mathrm{k}}, \tag{6.1}$$

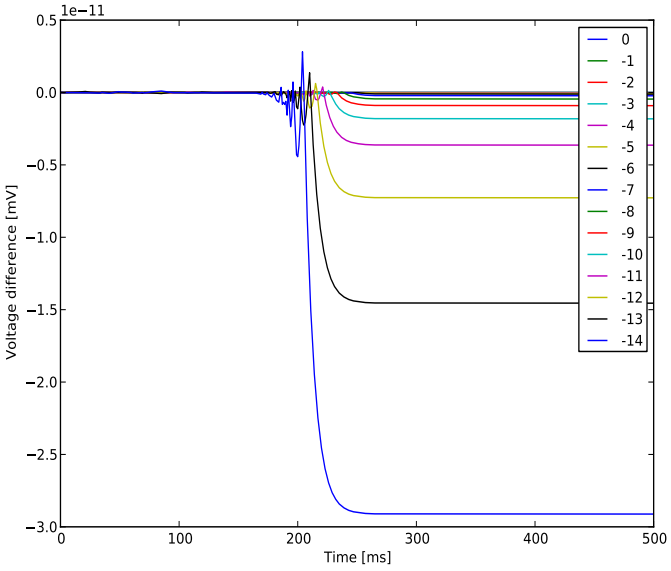
where I_{E} is the external input of current to the neuron.

In the lower figure (Fig. 6.5b) the same updating is done, but it is written a bit different,

$$V_{\mathrm{k}+1} = I_{\mathrm{E}} - (1 - \mathrm{e}^{-Ah}) V_{\mathrm{k}} + V_{\mathrm{k}}. \tag{6.2}$$



(a) Initial updating method



(b) Improved updating method

Figure 6.5: Differences between exact and simulated value of the membrane potential for the iaf_psc_alpha model. Different updating methods for the membrane potential are used in the two figures. For the improved method (Fig. 6.5b) the unavoidable errors caused by numerical representation dominate. Resolution $h = 2^0$–2^{-14} ms are plotted in both figures. The legend connects the resolutions and the different coloured lines.

These two equations (Eq. (6.1) and (6.2)) are mathematically equal but not computationally equal. When simulating, the same variable is just updated. The equations could therefore be written as

$$V = I_{\mathrm{E}} + \mathrm{e}^{-Ah}V, \tag{6.3}$$

$$V = I_{\mathrm{E}} - (1 - \mathrm{e}^{-Ah})V + V, \tag{6.4}$$

respectively Eq. (6.1) and (6.2).

The presence of V at both sides of the = is as it should be. The reason is because the = sign does not mean "equal" when programming. It rather means "set to". Eq. (6.3) is therefore not used to test whether V is equal $(I_{\mathrm{E}} + \mathrm{e}^{-Ah}V)$, which it surely not is. It is rather the way of giving the command "update V and set its value equal to $(I_{\mathrm{E}} + \mathrm{e}^{-Ah}V)$".The right hand side of the equation, $I_{\mathrm{E}} + \mathrm{e}^{-Ah}V$, is calculated first, and after that V is updated with the result of this calculation.

The precision of the updating is higher when using Eq.(6.4) than Eq. (6.3). This is because the properties of e^{-Ah}.

For small values of h, for example $h = 2^{-14}$ ms, e^{-Ah} is close to one. It is, however, a bit smaller, or more negative. Larger values of h will give less positive values.

In the beginning of e^{-Ah} there are a varying number of 9s, depending on the resolution h. The finest resolution gives most 9s in the beginning. All the 9s are not interesting for the development of the system. The information about the change in the system lies in the other decimals. But the 9s influence on the precision of the different updating methods.

If using the finest resolution it is normal to have five 9s in the beginning. When updating with Eq. (6.3) the maximal precision is therefore reduced from 10^{-16} to 10^{-11}. The errors seen in the beginning of Fig. 6.5a are therefore of order 10^{-11}.

For rougher resolutions, for example $h = 2^{-4}$ ms, the number of 9s in the beginning have decreased to three. This will give an error of order 10^{-13}, if Eq. (6.3) is used. This is too small to be seen in Fig. 6.5a. Rougher resolutions will in general decrease the value of e^{-Ah}. This is the reason why the finest resolution gives the biggest error.

If using Eq. (6.4) instead, the number of 9s is irrelevant. The difference $1 - \mathrm{e}^{-Ah}$ gives a number with some zeros in the beginning, but they can be removed. The result is a number where all decimals influence on the

behaviour of the system. The precision of the updating is therefore independent of the number of 9s in the beginning of e^{-Ah}. In other words, the precision is independent of the resolution h.

The updating of the membrane potential is in Fig. 6.5b done like in Eq. (6.2)/(6.4). Due to this, the errors seen in the beginning of Fig. 6.5a are removed.

6.4 Variations in firing rates and time constants

The different neuron models have been tested in the with-noise regime with varying firing rates from the Poisson generators or varying synaptic time constant. Fig. 6.6 and 6.7 show the changed behaviour of the membrane potential due to these variations.

In Fig. 6.6a and 6.6b the membrane potentials, which occured due to all the rates used, are plotted for the iaf_cond_alpha_ei_step model and iaf_psc_alpha model, respectively. Resolution $h = 2^{-4}$ ms is used in both cases. The legend is the excitatory rate used, and the inhibitory rate is one fourth of the excitatory rate.

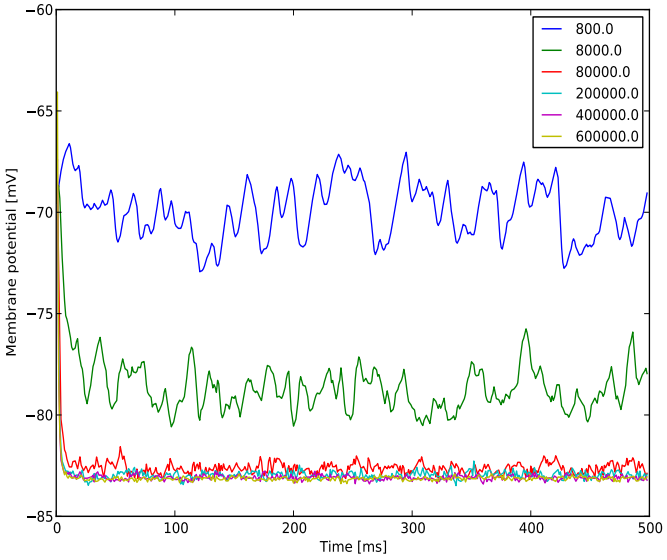
As shown in Fig. 4.1, the membrane potential in the with-noise regime will fluctuate around some value V_{fl}. But with varying rates the membrane potential for the two models, the iaf_cond_alpha_ei_step model and the iaf_psc_alpha model, will fluctuate around different values, and in different ways.

The conductance based model in Fig. 6.6a start at $V_{\text{fl}} \approx$ -70 mV for the lowest rates. When increasing the rates V_{fl} will decrease towards $V_{\text{fl}} \approx$ -82 mV. For excitatory rates from 80 000 Hz and up to the maximal rate used at 600 000 Hz, V_{fl} will practically be constant at -82 mV.

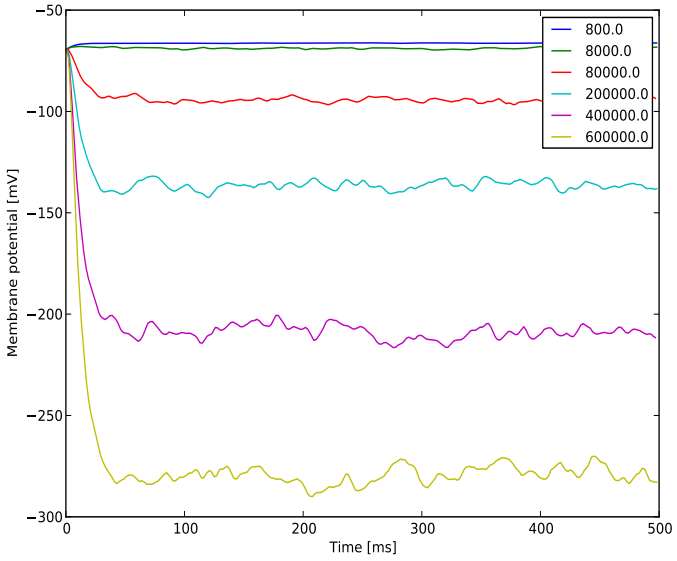
The behaviour of the current based model in Fig. 6.6b is the opposite of what was seen with the conductance based model in Fig. 6.6a. At the lowest rate of 800 Hz excitatory, this model will fluctuate around $V_{\text{fl}} \approx$ -70 mV. When increasing the rates from the Poisson generators, the value of the fluctuation potential V_{fl} will decrease. At the highest rates used, which means excitatory rate equal 600 000 Hz, $V_{\text{fl}} \approx$ -270 mV.

The same behaviour as seen with the iaf_cond_alpha_ei_step model also occur for the other two conductance based neuron models, the original iaf_cond_alpha model and the iaf_cond_alpha_ei model.

This opposite behaviour could be explained of the different way of modelling synaptic input. In the conductance based leaky integrate-and-fire models (iaf_cond_alpha and the new versions of it) the inhibitory synaptic input is

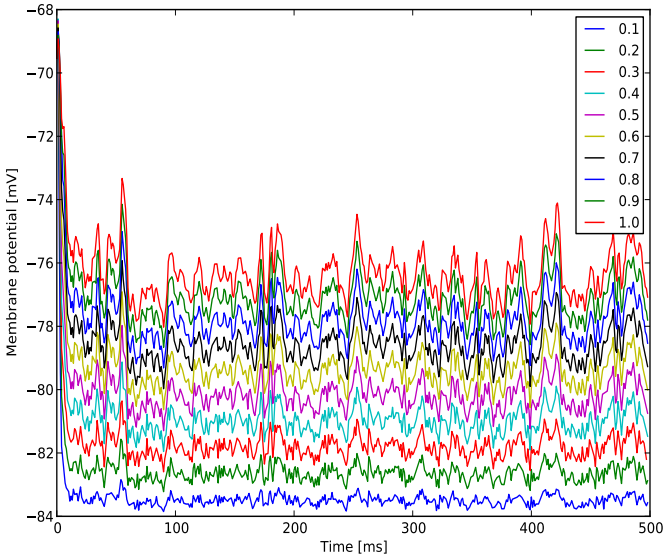


(a) Iaf_cond_alpha_ei_step, $h = 2^{-4}$ ms

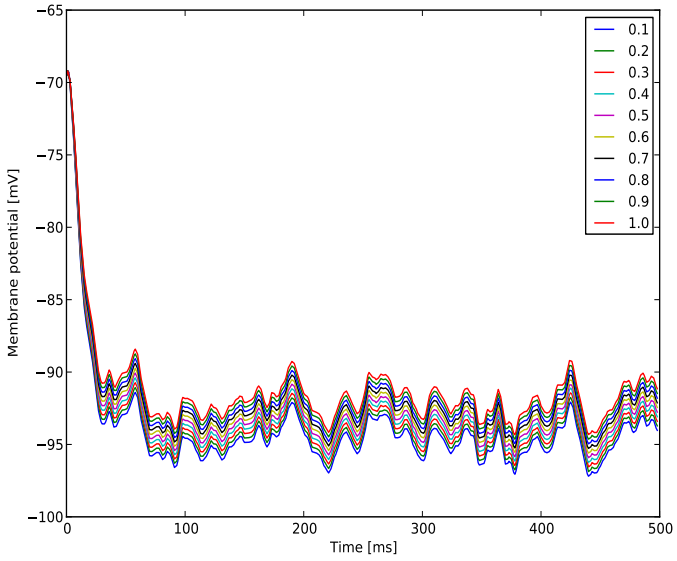


(b) Iaf_psc_alpha, $h = 2^{-4}$ ms

Figure 6.6: Fluctuations in membrane potential caused by rate variations. Different lines are different sets of rates. The excitatory rate is as in the legend and the inhibitory rate is one fourth of this.



(a) Iaf_cond_alpha_ei_step, $h = 2^{-4}$ ms



(b) Iaf_psc_alpha, $h = 2^{-4}$ ms

Figure 6.7: Fluctuations in membrane potential caused by variation of synaptic time constant. Different lines indicate different synaptic time constants. Values are given in the legend.

a current modelled as a change of conductance. The inhibitory input is the important one in this case because it reduces the membrane potential. It could be written as

$$I_{\text{in}} = -g(V - E_{\text{in}}). \tag{6.5}$$

E_{in} is the inhibitory reversal potential for the inhibitory synapse. This is determined by what type of ion which influences the most. The value of E_{in} is approximately -85 mV. g is the membrane conductance and V is actual membrane potential.

Increasing the rate of the Poisson generators will increase the current I_{in}. This happens due to an increase of g. The stabilizing is caused by the lower boundary of V at E_{in}.

The membrane potential for the conductance based models will decrease towards E_{in}. At some input rate the membrane potential will have reached this threshold. Further increase of rate of the Poisson generators will not lower the membrane potential more, it will just stabilize at E_{in}.

Eq. (6.5) is not valid for the current based leaky integrate-and-fire model iaf_psc_alpha. Synaptic input is modelled directly as a current input. All increase of the rate of the Poisson generators would therefore decrease the membrane potential.

The development of the membrane potential with increasing rate from Poisson generators is very different for the iaf_cond_alpha_ei_step model and the iaf_psc_alpha model. This is due to the different ways of modelling synaptic input, as described above.

The results of varying the excitatory synaptic time constants are shown in Fig. 6.7. Both of the same two models, the iaf_cond_alpha_ei_step model and the iaf_psc_alpha model, and the same resolution, $h = 2^{-4}$ ms, are used. The excitatory synaptic time constant varies from 0.1 ms to 1.0 ms in intervals of 0.1 ms.

The excitatory synaptic time constant influences how long the ion channels are open. At a large time constants the ion channels are open a long time. This results in a lot of synaptic signals which either enter or leave the neuron. For small time constants it is the other way around. The ion channels are open a short time and few signals cross the membrane.

The synaptic input in NEST is modelled in a way that an increased synaptic time constant will give more input in total. It takes longer time before it reaches its maximum, but it lasts longer. The effect of this is seen in Fig. 6.7a. For low values of the synaptic time constant each signal will last short and influence less. For high values of the time constant the individual signal

influences a lot because the signal itself is more powerful. The effect is almost not seen in Fig. 6.7b for the iaf_psc_alpha model. The differences in the neuron models may be the reason for this.

Because the time of the maximum of the signal is dependent of the value of the synaptic time constant, the lines for the different time constants should also have been shifted compared to each other. This is not seen in Fig. 6.7. The little variation of the synaptic time constant, only from 0.1 to 1.0, is probably the reason. However, it is not likely that it would have been possible to see the effect, neither for the iaf_cond_alpha_ei_step model nor the other conductance based models. The variation of signal strength, as described above, would probably have dominated. On the other hand, this shifting could perhaps be possible to see for the iaf_psc_alpha model. This is due to the fact that the signals in this model seem to influence almost equally, independent of the value of the synaptic time constant. In any case, a greater variation of the synaptic time constants would then be needed.

Chapter 7

Effectiveness

It is desirable with a neuron model which efficiently delivers results with good precision. Chapter 6 concerned the numerical precision and this chapter will be about the simulation time. This chapter concerns the total simulation times. The new models are proved to be more efficient than the original iaf_cond_alpha model. The simulation time per step is also calculated. At the end of this chapter both an error that was found during speed tests and its solution are described.

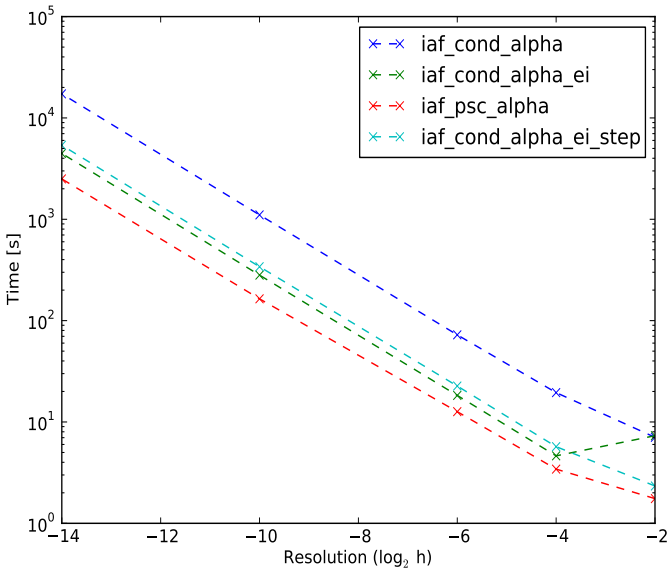
7.1 Simulation times

The work done during this master thesis was aimed to develop a faster variant of the iaf_cond_alpha neuron model. Tests of the simulation times have therefore been done.

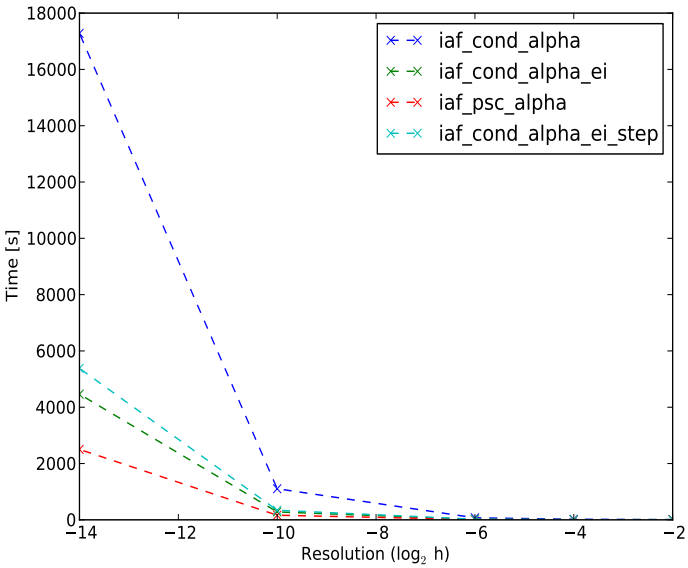
Testing of the simulation times have been done for the different neuron models. It is done as described in Section 4.2 and with resolutions $h = [2^{-2}, 2^{-4}, 2^{-6}, 2^{-10}, 2^{-14}]$. Tab. 7.1 graphically shows the achieved simulation times. The values for each model and resolution are the median value over three repetitions. For all models and resolutions, except the iaf_cond_alpha model at resolution $\log_2 h = -14$, three repetitions have been made. For this simulation only one repetition has been made.

Fig. 7.1 shows how the simulation times increase with decreasing length of the time step. The resolution is on a logarithmic scale and the time is both logarithmic and non-logarithmic, respectively in Fig. 7.1a and 7.1b.

The results from the test of simulation time show that the new neuron models, iaf_cond_alpha_ei and iaf_cond_alpha_ei_step, are much faster than their origin, the iaf_cond_alpha model. The new models need only 25% and



(a) Double logarithmic plot



(b) Semilogarithmic plot

Figure 7.1: Simulation times for the four different neuron models. The value for iaf_cond_alpha at resolution $\log_2 h = -14$ is based on one repetition in contrast to the other which are based on three repetitions.

Resolution ($\log_2 h$)	**-2**	**-4**	**-6**	**-10**	**-14**
Iaf_cond_alpha	7.1	19.5	72.4	1 104	17 281
Iaf_cond_alpha_ei	7.3	4.6	18.3	281	4 464
Iaf_cond_alpha_ei_step	2.3	5.7	22.6	340	5 381
Iaf_psc_alpha	1.8	3.4	12.6	164	2 502

Table 7.1: Simulation times for all four neuron models. Simulated time is 1000 ms. Simulations times are median values and the unit is seconds. At resolution $h = 2^{-2}$ ms the firing rates for iaf_cond_alpha_ei and _step are respectively 215 Hz (excitatory and inhibitory) and 275 Hz (excitatory)/318 Hz (inhibitory). The simulation times for these models are therefore not comparable with the other models at the same resolution. Three repetitions of every model at every resolution is done except for the iaf_cond_alpha model at resolution $h = 2^{-14}$ ms. Only one repetition is done there.

30%, for the iaf_cond_alpha_ei model and the iaf_cond_alpha_ei_step model respectively, of the time necessary when using the iaf_cond_alpha model. This statement is based on the simulation times for all resolutions except $h = 2^{-2}$ ms.

The simulation times from the new neuron models are now also comparable with the simulation times from the iaf_psc_alpha model. At resolution $h = 2^{-4}$ ms, the simulation times for both of the new models are almost equal to the simulation time used when simulating with the iaf_psc_alpha model. The simulation times of the new models are still comparable with the iaf_psc_alpha model when decreasing the time step to $h = 2^{-6}$ ms, but they are 1.5–2 times bigger. At finer resolutions the iaf_psc_alpha model is approximately two times faster. The iaf_cond_alpha model is approximately six times slower than the iaf_psc_alpha model and, compared to this, the new models have made great progress in reducing the simulation time.

When comparing simulation times it is important that the activity in the network is equal for all models and resolutions used. If it is not, different networks are used for testing of simulation times. Wrong conclusions could therefore be made.

The firing rates for both the iaf_cond_alpha_ei model and the iaf_cond_alpha_ei_step model were much higher than the expected 30 Hz when using resolution $h = 2^{-2}$ ms. The firing rates were 216 Hz (both excitatory and inhibitory) for the iaf_cond_alpha_ei_step model, and 276 Hz (excitatory)/318 Hz (inhibitory) for the iaf_cond_alpha_ei model. The simulation times for these models at this resolution is therefore not comparable with simulation times from the two other models, the iaf_cond_alpha model and the iaf_psc_alpha model. This could explain the unexpected long simulation time for iaf_cond_alpha_ei model at resolution $h = 2^{-2}$ ms.

Simulation time per step

The focus in this section has until now been on the total simulation time. Another interesting parameter to measure is the time used per step of the simulation. This could also give an estimate of how efficient the different neuron models are compared to each other.

The total simulation time T_{tot} is the sum of the time used for doing the numerical integration with all neurons, T_{step}, and the time used for other things, B. Time used for spike handling is included in B. The total simulation time is therefore

$$T_{\text{tot}} = T_{\text{step}} + B. \tag{7.1}$$

The time used for numerical integration is equal to the product of the time used per steps (t_{step}) the number of steps taken (n) and the number of neurons simulated (N). The number of steps taken could also be written as the product of the time simulated (T_{sim}) and the number of steps taken per simulated time ($1/h$). Combining this gives the following equation which describes the total simulation time T_{tot},

$$T_{\text{tot}} = N t_{\text{step}} T_{\text{sim}} \frac{1}{h} + B. \tag{7.2}$$

When simulating with very small time steps, for example $h = 2^{-14}$ ms, the constant time B is very small compared to the amount of time used. It is therefore plausible to assume $B = 0$. This gives a simple connection with only t_{step} as unknown.

$$t_{\text{step}} = h \frac{T_{\text{tot}}}{N T_{\text{sim}}} \tag{7.3}$$

If using this equation, t_{step} for the different neuron models could be calculated. Using simulation times for $h = 2^{-14}$ ms as given in Tab. 7.1, $T_{\text{sim}} = 10^3$ ms and $N = 1250$, gives simulation times per step as given in Tab. 7.2. The table shows that, as shown earlier, the new models simulates about 3.5 times faster than the original iaf_cond_alpha neuron model and about two times slower than the iaf_psc_alpha model.

Initialization problems

During the testing of speed a problem occured. When using the original iaf_cond_alpha neuron model, the simulation just stopped after having

Model	**Time**
iaf_cond_alpha	840
iaf_cond_alpha_ei	220
iaf_cond_alpha_ei_step	260
iaf_psc_alpha	120

Table 7.2: Simulation time per step for four neuron models. Times given in 10^{-9} seconds.

done varying number of simulations. The simulated network is described in speed_brunel.py (see Appendix B). The problem did not happen during a simulation, but when one of the simulations was to be started. There were problems with running the first simulation too.

One idea was that this problem could be explained with the use of GSL in the iaf_cond_alpha neuron model. It was tested that after the progress had stopped, over 58% of the time was spent in the GSL. It was also thought that the bit version of the operative system might influence. This hypothesis was weakened after the error was reproduced at both 32 and 64-bits versions of Linux Ubuntu.

An uninitialized internal variable caused the problem. This could result in very high values of input current. GSL would then stop. Whether GSL stopped or not depended on the value in the memory at the actual position where this internal variable was placed. The error therefore seemed to happen at random. After the variable was initialized correct, the simulations were carried out without any errors.

Chapter 8

Summary

The iaf_cond_alpha model in NEST (Neural Simulation Toolbox)[1] is a current based model of a leaky integrate-and-fire neuron. Two new models based on this have been developed during this master thesis. They are named iaf_cond_alpha_ei and iaf_cond_alpha_ei_step. The aim when developing these new models was to reduce the simulation time of the original iaf_cond_alpha model. The numerical precision and error should also be kept at least as good as they were.

The neuron models differ in how the numerical integration is done. For all models there are five equations that need to be updated. Four describe the synaptic dynamics and the last describes the membrane potential.

The original model uses a Runge-Kutta-Fehlberg method of the fourth-fifth order with adaptive step size control from the GNU Science Library. The new models use Exact Integration for updating of the synaptic dynamics. The membrane potential is updated with use of a manually implemented Runge-Kutta method. In the iaf_cond_alpha_ei model a fourth order method is used and in the iaf_cond_alpha_ei model a fifth order method is used.

The new models have been tested both on numerical precision and error, and simulation time. Test regimes with only current input or with extra produced noise from Poisson generators have been used to test precision and error. Network simulations have been used to measure the simulation times.

Both of the new models keep the numerical precision of the original model iaf_cond_alpha. The simulation time decreased to approximately 25% for the iaf_cond_alpha_ei model and 30% for the iaf_cond_alpha_ei_step model, when compared to the simulation time for the iaf_cond_alpha neuron model.

[1] http://www.nest-initiative.org

The convertion from use of GSL to use of Exact Integration and manually implemented Runge-Kutta methods was therefore successful.

Further work

The new neuron models have proved to deliver as precise results as the original iaf_cond_alpha model more efficiently. They should therefore be included in NEST. Before this could happen it is necessary, and desirable, with some more testing of the new models. The source code has to be controlled and adjusted, and it has to be well commented too.

This master thesis has shown that the combination of Exact Integration and manually implemented Runge-Kutta method gives good results. Other neuron models can therefore also be converted from use of GSL.

Appendix A

An introduction to PyNEST

PyNEST gives easy access to the models, devices and other things necessary when simulating neural networks. This chapter will show and explain the functions in PyNEST used in this master thesis. Hopefully, the basic functions needed for not to advanced use would then be covered. All functions are showed as `nest.function`, but are only referred to as `function`.

Further information about PyNEST can be found in the article of Eppler et al. [2009].

Creating objects

The first essential thing to do is to import nest to your Python script with

```
import nest.
```

Objects could then be created with the command

```
nest.Create('object').
```

Creating an iaf_cond_alpha neuron could be done like

```
cond = nest.Create('iaf_cond_alpha').
```

Several objects could be created at the same time. The wanted number of objects is then passed as an argument to `Create`. Creating two iaf_cond_alpha neurons could look like

```
cond2 = nest.Create('iaf_cond_alpha', 2).
```

Since just the name of the object, and the number of objects wanted made, is passed to `Create`, `cond` and `cond2` are created with standard values. For neurons with non-standard values a dictionary of parameters could be

passed to `Create` as an argument. This dictionary should contain names of parameters that exist for the actual neuron and the value this variable should get. Creating an iaf_cond_alpha-neuron, change some of the parameters and store it in the variable `neuron`, is done like this

```
neuron = nest.Create('iaf_cond_alpha', params=
    {"C_m":  200.0, "tau_syn_ex":  0.5}).
```

In addition to different types of neurons it is possible to create measuring instruments and generators. Voltmeters, multimeters, spike generators and Poisson generators have been used in this thesis.

Both the voltmeter and the multimeter could measure the membrane potential. The multimeter could also record all other variables that it is possible to record from the actual neuron type. The spike generator produces extra spikes and the Poisson generator produces an approximation of spike input from other neurons.

In the same way as neurons, measuring instruments and generators could be created with standard and non-standard values or settings. The next two lines show how to create a voltmeter with recording interval of 0.1 ms and a multimeter which records the membrane potential V_{m} and the excitatory conductance g_{ex}. The argument to `record_from` has to be enclosed by square brackets (`[ ]`). This is also true if the argument is just one variable.

```
voltmeter = nest.Create("voltmeter", params =
            {"interval":  0.1})
multimeter = nest.Create('multimeter',params =
     {'record_from':  ['V_m', 'g_ex'] })
```

How to create Poisson and spike generators is shown in the next two lines of code. The first line creates two Poisson generators, one excitatory and one inhibitory generator. Their firing rates are set to 80000 Hz and 20 000 Hz. The Poisson generators created are of the type "poisson_generator_ps" because these give more precise results than the "poisson_generator". The second line shows how to create a spike generator which should produces spikes at 100, 200, 300 and 400 ms.

```
noise = nest.Create("poisson_generator_ps", 2, params =
       [{'rate':  80000.0}, {'rate':  20000.0}] )
spg = nest.Create('spike_generator', 1, {'spike_times':
            [100., 200., 300.  , 400.]} )
```

When passing more than one parameter to "`params =`",as done for the rates with `noise`, it is necessary to enclose the arguments with square brackets (`[ ]`). But the "`params =`" is not necessary itself. In the last of the code lines, where the `spg` is created, the "`params =`" is excluded. Then it is necessary to

pass the number of objects wanted made as the second argument to `Create`. This is also valid if only one object is wanted, like for `spg`.

Values or parameters that are set in the creation could be changed with the `nest.SetStatus`. The following two lines of code show examples of how this function is used. The first one changes to membrane capacitance in `cond` and the next changes multimeter so that it only records the excitatory conductance. It is possible to use `[]` instead of "`params =` " here too.

```
    nest.SetStatus(cond, params = {'C_m':  300.0})
nest.SetStatus(multimeter, [{'record_from':  ['g_ex']}] )
```

Simulating

After the wanted objects have been created and before it is possible to simulate, the objects have to be connected. There are two possible ways of connecting objects, depending on the number of objects that are to be connected. `nest.Connect` connects one sending object to one receiving object. `nest.ConvergentConnect` connects every single sending object to one receiving object.

```
nest.Connect(multimeter, neuron)
```

connects `multimeter` and `neuron` with use of `Connect`

An example of how to use `ConvergentConnect` is shown next. The two Poisson generators `noise` are being connected to the iaf_cond_alpa neuron `cond`. The third argument describes the strength of the connections, where 0.50 and -4.0 is the strength of the excitatory and inhibitory connections, respectively. The fourth argument describes the delay of the transmission of signals. Because of technical reasons, the delay should be one millisecond, but it is possible for the Poisson generator to have a delay which equals zero millisecond.

```
nest.ConvergentConnect(noise, cond, [0.50, -4.0], 1.0)
```

The simulation is done with the command `nest.Simulate(simulation_time)`. The data type of the argument has to be a double.

After the simulation is done, recorded values can be picked out with `nest.GetStatus`. The next lines of code show how to get the excitatory conductance, membrane potential and recording times from multimeter, voltmeter and multimeter, respectively. It does not matter whether `GetStatus(multimeter, "events")[0]` or `GetStatus(multimeter)[0]["events"]` is being used. This is valid also for voltmeter.

```
  g_ex = nest.GetStatus(multimeter)[0]['events']['g_ex']
v_m = nest.GetStatus(voltmeter, "events")[0]['potentials']
   T = nest.GetStatus(multimeter, "events")[0]['times']
```

`GetStatus` could also be used to show, for example, which values that could be recorded with a `multimeter`. The command is given as the first code line, and the result as the second one.

```
       nest.GetStatus(cond)[0]['recordables']
['V_m', 'g_ex', 'g_in', 'integration_step', 't_ref_remaining']
```

Appendix B

Programming code

Programming code for the new neuron models iaf_cond_alpha_ei and iaf_cond_alpha_ei_step, declaration file (.h) and definition file (.cpp) for both, can be found on pages 72–92.

Test scripts used to create figures included or data used in this master thesis can be found on pages 93–104.

Neuron models

iaf_cond_alpha_ei.h

```
/*
 *  iaf_cond_alpha.h
 *
 *  This file is part of NEST
 *
 *  Copyright (C) 2005-2009 by
 *  The NEST Initiative
 *
 *  See the file AUTHORS for details.
 *
 *  Permission is granted to compile and modify
 *  this file for non-commercial use.
 *  See the file LICENSE for details.
 *
 */

#ifndef IAF_COND_ALPHA_EI_H
#define IAF_COND_ALPHA_EI_H

#include "config.h"

#include "nest.h"
#include "event.h"
#include "archiving_node.h"
#include "ring_buffer.h"
#include "connection.h"
#include "universal_data_logger.h"
#include "recordables_map.h"

// next line will disappear when all models support multimeter
#include "analog_data_logger.h"


/* BeginDocumentation
Name: iaf_cond_alpha_ei - Simple conductance based leaky integrate
    -and-fire neuron model.

Description:
iaf_cond_alpha_ei is an implementation of a spiking neuron using
    IAF dynamics with conductance-based synapses. Incoming spike
    events induce a post-synaptic change of conductance modelled
    by an alpha function. The alpha function is normalised such
    that an event of weight 1.0 results in a peak current of 1 nS
    at t = tau_syn.

Parameters:
The following parameters can be set in the status dictionary.

V_m         double - Membrane potential in mV
E_L         double - Leak reversal potential in mV.
C_m         double - Capacity of the membrane in pF
t_ref       double - Duration of refractory period in ms.
V_th        double - Spike threshold in mV.
V_reset     double - Reset potential of the membrane in mV.
E_ex        double - Excitatory reversal potential in mV.
E_in        double - Inhibitory reversal potential in mV.
g_L         double - Leak conductance in nS;
tau_ex      double - Rise time of the excitatory synaptic alpha
    function in ms.
tau_in      double - Rise time of the inhibitory synaptic alpha
    function in ms.
I_e         double - Constant input current in pA.

Sends: SpikeEvent

Receives: SpikeEvent, CurrentEvent, PotentialRequest,
    SynapticConductanceRequest

Author: Schrader, Plesser

SeeAlso: iaf_cond_exp, iaf_cond_alpha_mc
*/

namespace nest
{
   /**
   * Integrate-and-fire neuron model with two conductance-based
       synapses.
   *
   * @note Per 2009-04-17, this class has been revised to our
       newest
   *       insights into class design. Please use THIS CLASS as a
       reference
   *       when designing your own models with nonlinear dynamics.
   *       One weakness of this class is that it distinguishes
       between
   *       inputs to the two synapses by the sign of the synaptic
       weight.
   *       It would be better to use receptor_types, cf
       iaf_cond_alpha_mc.
   */
  class iaf_cond_alpha_ei : public Archiving_Node
  {
```

```
// Boilerplate function declarations
    --------------------------------

public:

  iaf_cond_alpha_ei();
  iaf_cond_alpha_ei(const iaf_cond_alpha_ei&);
  ~iaf_cond_alpha_ei();

  /*
   * Import all overloaded virtual functions that we
   * override in this class.  For background information,
   * see http://www.gotw.ca/gotw/005.htm.
   */

  using Node::connect_sender;
  using Node::handle;

  port check_connection(Connection&, port);

  port connect_sender(SpikeEvent &, port);
  port connect_sender(CurrentEvent &, port);
  port connect_sender(DataLoggingRequest &, port);

  // The next two lines will disappear once all models have been
  // equipped for use with multimeter. Do not include them in
  // new models!
  port connect_sender(PotentialRequest &, port);
  port connect_sender(SynapticConductanceRequest &, port);

  void handle(SpikeEvent &);
  void handle(CurrentEvent &);
  void handle(DataLoggingRequest &);
  void handle(PotentialRequest &);            // will disappear
      (multimeter)
  void handle(SynapticConductanceRequest &);  // will disappear
      (multimeter)

  void get_status(DictionaryDatum &) const;
  void set_status(const DictionaryDatum &);

private:
  void init_node_(const Node& proto);
  void init_state_(const Node& proto);
  void init_buffers_();
  void calibrate();
  void update(Time const &, const long_t, const long_t);

  // END Boilerplate function declarations
      ----------------------------

  // Friends
      --------------------------------------------------------

  // The next two classes need to be friends to access the
       State_ class/member
  friend class RecordablesMap<iaf_cond_alpha_ei>;
  friend class UniversalDataLogger<iaf_cond_alpha_ei>;

private:

  // Parameters class
      --------------------------------------------

  //! Model parameters
  struct Parameters_ {
    double_t V_th;         //!< Threshold Potential in mV
    double_t V_reset;      //!< Reset Potential in mV
    double_t t_ref;        //!< Refractory period in ms
    double_t g_L;          //!< Leak Conductance in nS
    double_t C_m;          //!< Membrane Capacitance in pF
    double_t E_ex;         //!< Excitatory reversal Potential in
        mV
    double_t E_in;         //!< Inhibitory reversal Potential in
        mV
    double_t E_L;          //!< Leak reversal Potential (aka
         resting potential) in mV
    double_t tau_synE;     //!< Synaptic Time Constant Excitatory
         Synapse in ms
    double_t tau_synI;     //!< Synaptic Time Constant for
        Inhibitory Synapse in ms
    double_t I_e;          //!< Constant Current in pA


    Parameters_();         //!< Set default parameter values

    void get(DictionaryDatum&) const;  //!< Store current values
         in dictionary
    void set(const DictionaryDatum&);  //!< Set values from
        dicitonary
  };

  // State variables class
      --------------------------------------

  /**
   * State variables of the model.
   *
```

```
   * State variables consist of the state vector for the
       subthreshold
   * dynamics and the refractory count. The state vector must be
       a
   * C-style array to be compatible with GSL ODE solvers.
   *
   * @note Copy constructor and assignment operator are required
       because
   *       of the C-style array.
   */
public:
  struct State_ {

    double_t V_; //!< Membrane potential

    double_t y1_ex_; //!< Excitatory conductance, derivative
    double_t y2_ex_; //!< Excitatory conductance

    double_t y1_in_; //!< Inhibitory conductance, derivative
    double_t y2_in_; //!< Inhibitory conductance



    //!< number of refractory steps remaining
    int_t    r;

    State_(const Parameters_&);  //!< Default initialization
    State_(const State_&);
    State_& operator=(const State_&);

    void get(DictionaryDatum&) const;  //!< Store current values
         in dictionary

    /**
     * Set state from values in dictionary.
     * Requires Parameters_ as argument to, eg, check bounds.'
     */
    void set(const DictionaryDatum&, const Parameters_&);


  };


private:

  // Buffers class
      -------------------------------------------------------

  /**
   * Buffers of the model.
   * Buffers are on par with state variables in terms of
       persistence,
   * i.e., initalized only upon first Simulate call after
       ResetKernel
   * or ResetNetwork, but are implementation details hidden from
       the user.
   */
  struct Buffers_ {
    Buffers_(iaf_cond_alpha_ei&); //!<Sets buffer pointers to 0
    Buffers_(const Buffers_&, iaf_cond_alpha_ei&); //!<Sets
        buffer pointers to 0

    //! Logger for all analog data
    UniversalDataLogger<iaf_cond_alpha_ei> logger_;

    /** buffers and sums up incoming spikes/currents */
    RingBuffer spike_exc_;
    RingBuffer spike_inh_;
    RingBuffer currents_;

    // The next two lines will disappear once all models have
        been
    // equipped for use with multimeter. Do not include them in
    // new models!
    AnalogDataLogger<PotentialRequest>           potentials_;
    AnalogDataLogger<SynapticConductanceRequest> conductances_;

    // IntergrationStep_ should be reset with the neuron on
        ResetNetwork,
    // but remain unchanged during calibration. Since it is
        initialized with
    // step_, and the resolution cannot change after nodes have
        been created,
    // it is safe to place both here.
    double_t step_;           //!< step size in ms
    double   IntegrationStep_;//!< current integration time step
        , updated by GSL
  };

  // Variables class
      -------------------------------------------------------

  /**
   * Internal variables of the model.
   * Variables are re-initialized upon each call to Simulate.
   */
  struct Variables_ {
    /**
     * Impulse to add to DG_EXC on spike arrival to evoke unit-
         amplitude
```

```
   * conductance excursion.
   */
  double_t PSConInit_E;

  /**
   * Impulse to add to DG_INH on spike arrival to evoke unit-
       amplitude
   * conductance excursion.
   */
  double_t PSConInit_I;

  //! refractory time in steps
  int_t    RefractoryCounts;

  //! make external input current available to dynamics
       function
  double_t I_stim;


  //! Variables for the propagator matrices, both excitatory
       and inhibitory
  double_t P11_ex_;
  double_t P21_ex_;
  double_t P22_ex_;

  double_t P11_in_;
  double_t P21_in_;
  double_t P22_in_;

};

// Access functions for UniversalDataLogger
    ------------------------------

double_t get_V_() const {return S_.V_;} //!< Returns membrane
     potential
double_t get_g_ex_() const {return S_.y2_ex_;} //!< Returns
     excitatory conductance
double_t get_g_in_() const {return S_.y2_in_;} //!< Returns
     inhibitory conductance

//! Read out remaining refractory time, used by
     UniversalDataLogger
double_t get_r_() const { return Time::get_resolution().get_ms
     () * S_.r; }

//! Read out length of most recent integration time step, used
     by UniversalDataLogger
double_t get_integration_step_() const { return B_.
     IntegrationStep_; }

  //! Updates the membrane potential so the Runge-Kutta method
       gets the rigt values.
  double_t dVdt (double_t V_m) const;

  //! Updates the conductances a half computational time step
       further
  void synapse_half_step();


  // Data members
     ---------------------------------------------------------


  // keep the order of these lines, seems to give best
       performance
  Parameters_ P_;
  State_      S_;
  Variables_  V_;
  Buffers_    B_;

  //! Mapping of recordables names to access functions
  static RecordablesMap<iaf_cond_alpha_ei> recordablesMap_;
};


// Boilerplate inline function definitions
    ----------------------------------

inline
port iaf_cond_alpha_ei::check_connection(Connection& c, port
     receptor_type)
{
  SpikeEvent e;
  e.set_sender(*this);
  c.check_event(e);
  return c.get_target()->connect_sender(e, receptor_type);
}

inline
port iaf_cond_alpha_ei::connect_sender(SpikeEvent&, port
     receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  return 0;
}

inline
```

```
port iaf_cond_alpha_ei::connect_sender(CurrentEvent&, port
    receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  return 0;
}

inline
port iaf_cond_alpha_ei::connect_sender(DataLoggingRequest& dlr,
                                       port receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  B_.logger_.connect_logging_device(dlr, recordablesMap_);
  return 0;
}

inline
void iaf_cond_alpha_ei::get_status(DictionaryDatum &d) const
{
  P_.get(d);
  S_.get(d);
  Archiving_Node::get_status(d);

  (*d)[names::recordables] = recordablesMap_.get_list();
}

inline
void iaf_cond_alpha_ei::set_status(const DictionaryDatum &d)
{
  Parameters_ ptmp = P_;  // temporary copy in case of errors
  ptmp.set(d);                       // throws if BadProperty
  State_      stmp = S_;  // temporary copy in case of errors
  stmp.set(d, ptmp);                 // throws if BadProperty

  // We now know that (ptmp, stmp) are consistent. We do not
  // write them back to (P_, S_) before we are also sure that
  // the properties to be set in the parent class are internally
  // consistent.
  Archiving_Node::set_status(d);

  // if we get here, temporaries contain consistent set of
      properties
  P_ = ptmp;
  S_ = stmp;
}

// The next two functions will disappear once all models have
    been
  // equipped for use with multimeter. Do not include them in
  // new models!
  inline
  port iaf_cond_alpha_ei::connect_sender(PotentialRequest& pr,
      port receptor_type)
  {
    if (receptor_type != 0)
      throw UnknownReceptorType(receptor_type, get_name());
    B_.potentials_.connect_logging_device(pr);
    return 0;
  }

  inline
  port iaf_cond_alpha_ei::connect_sender(
      SynapticConductanceRequest& scr, port receptor_type)
  {
    if (receptor_type != 0)
      throw UnknownReceptorType(receptor_type, get_name());
    B_.conductances_.connect_logging_device(scr);
    return 0;
  }


} // namespace


#endif //IAF_COND_ALPHA_EI_H
```

iaf_cond_alpha_ei.cpp

```
/*
 *  iaf_cond_alpha_ei.cpp
 *
 *  This file is part of NEST
 *
 *  Copyright (C) 2005-2009 by
 *  The NEST Initiative
 *
 *  See the file AUTHORS for details.
 *
 *  Permission is granted to compile and modify
 *  this file for non-commercial use.
 *  See the file LICENSE for details.
 *
 */
```

```
#include "iaf_cond_alpha_ei.h"

#include "exceptions.h"
#include "network.h"
#include "dict.h"
#include "integerdatum.h"
#include "doubledatum.h"
#include "dictutils.h"
#include "numerics.h"
#include "analog_data_logger_impl.h"
#include "universal_data_logger_impl.h"
#include <limits>

#include <iomanip>
#include <iostream>
#include <cstdio>

/* ----------------------------------------------------------------

 * Recordables map
 * ----------------------------------------------------------------
 */

nest::RecordablesMap<nest::iaf_cond_alpha_ei> nest::
    iaf_cond_alpha_ei::recordablesMap_;

namespace nest   // template specialization must be placed in
    namespace
{
  /*
   * Override the create() method with one call to RecordablesMap
      ::insert_()
   * for each quantity to be recorded.
   */
  template <>
  void RecordablesMap<iaf_cond_alpha_ei>::create()
  {
    // use standard names whereever you can for consistency!
    insert_(names::V_m,
            &iaf_cond_alpha_ei::get_V_);
    insert_(names::g_ex,
            &iaf_cond_alpha_ei::get_g_ex_);
    insert_(names::g_in,
            &iaf_cond_alpha_ei::get_g_in_);

    insert_(names::t_ref_remaining,
            &iaf_cond_alpha_ei::get_r_);
    insert_(names::integration_step,
            &iaf_cond_alpha_ei::get_integration_step_);
  }
}

/* ----------------------------------------------------------------

 * Default constructors defining default parameters and state
 * ----------------------------------------------------------------
 */

nest::iaf_cond_alpha_ei::Parameters_::Parameters_()
  : V_th    (-55.0    ),  // mV
    V_reset (-60.0    ),  // mV
    t_ref   (  2.0    ),  // ms
    g_L     ( 16.6667 ),  // nS
    C_m     (250.0    ),  // pF
    E_ex    (  0.0    ),  // mV
    E_in    (-85.0    ),  // mV
    E_L     (-70.0    ),  // mV
    tau_synE(  0.2    ),  // ms
    tau_synI(  2.0    ),  // ms
    I_e     (  0.0    )   // pA
{
  recordablesMap_.create();
}

nest::iaf_cond_alpha_ei::State_::State_(const Parameters_& p)
  : V_(p.E_L),
    y1_ex_(0),
    y2_ex_(0),
    y1_in_(0),
    y2_in_(0),
    r(0)
{
}

nest::iaf_cond_alpha_ei::State_::State_(const State_& s)
  : V_(s.V_),
    y1_ex_(s.y1_ex_),
    y2_ex_(s.y2_ex_),
    y1_in_(s.y1_in_),
    y2_in_(s.y2_in_),
    r(s.r)
{
}
```

```
nest::iaf_cond_alpha_ei::State_& nest::iaf_cond_alpha_ei::State_::
    operator=(const State_& s)
{
  if ( this == &s )  // avoid assignment to self
    return *this;

  V_ = s.V_;
  y1_ex_ = s.y1_ex_;
  y2_ex_ = s.y2_ex_;
  y1_in_ = s.y1_in_;
  y2_in_ = s.y2_in_;
  r = s.r;

  return *this;
}

nest::iaf_cond_alpha_ei::Buffers_::Buffers_(iaf_cond_alpha_ei& n)
  : logger_(n)
  {
   // The other member variables are left uninitialised or are
   // automatically initialised by their default constructor.
  }

nest::iaf_cond_alpha_ei::Buffers_::Buffers_(const Buffers_&,
    iaf_cond_alpha_ei& n)
  : logger_(n)
  {
  // The other member variables are left uninitialised or are
  // automatically initialised by their default constructor.
  }

/*
    ----------------------------------------------------------

 * Parameter and state extractions and manipulation functions
 *
    ----------------------------------------------------------
     */

void nest::iaf_cond_alpha_ei::Parameters_::get(DictionaryDatum &d)
    const
{
  def<double>(d,names::V_th,          V_th);
  def<double>(d,names::V_reset,       V_reset);
  def<double>(d,names::t_ref,         t_ref);
  def<double>(d,names::g_L,           g_L);
  def<double>(d,names::E_L,           E_L);
  def<double>(d,names::E_ex,          E_ex);
  def<double>(d,names::E_in,          E_in);
  def<double>(d,names::C_m,           C_m);
  def<double>(d,names::tau_syn_ex,    tau_synE);
  def<double>(d,names::tau_syn_in,    tau_synI);
  def<double>(d,names::I_e,           I_e);
}

void nest::iaf_cond_alpha_ei::Parameters_::set(const
    DictionaryDatum& d)
{
  // allow setting the membrane potential
  updateValue<double>(d,names::V_th,     V_th);
  updateValue<double>(d,names::V_reset,  V_reset);
  updateValue<double>(d,names::t_ref,    t_ref);
  updateValue<double>(d,names::E_L,      E_L);

  updateValue<double>(d,names::E_ex,     E_ex);
  updateValue<double>(d,names::E_in,     E_in);

  updateValue<double>(d,names::C_m,      C_m);
  updateValue<double>(d,names::g_L,      g_L);

  updateValue<double>(d,names::tau_syn_ex, tau_synE);
  updateValue<double>(d,names::tau_syn_in, tau_synI);

  updateValue<double>(d,names::I_e,      I_e);

  if ( V_reset >= V_th )
    throw BadProperty("Reset potential must be smaller than
        threshold.");

  if ( C_m <= 0 )
    throw BadProperty("Capacitance must be strictly positive.");

  if ( t_ref < 0 )
    throw BadProperty("Refractory time cannot be negative.");

  if ( tau_synE <= 0 || tau_synI <= 0 )
    throw BadProperty("All time constants must be strictly
        positive.");
}

void nest::iaf_cond_alpha_ei::State_::get(DictionaryDatum &d)
    const
{
  def<double>(d, names::V_m, V_); // Membrane potential
}

void nest::iaf_cond_alpha_ei::State_::set(const DictionaryDatum& d
    , const Parameters_&)
{
  updateValue<double>(d, names::V_m, V_);
```

```
}


/* ----------------------------------------------------------------

 * Default and copy constructor for node, and destructor
 *
 * ---------------------------------------------------------------- */

nest::iaf_cond_alpha_ei::iaf_cond_alpha_ei()
  : Archiving_Node(),
    P_(),
    S_(P_),
    B_(*this)
{
}

nest::iaf_cond_alpha_ei::iaf_cond_alpha_ei(const iaf_cond_alpha_ei
    & n)
  : Archiving_Node(n),
    P_(n.P_),
    S_(n.S_),
    B_(n.B_, *this)
{
}

 nest::iaf_cond_alpha_ei::~iaf_cond_alpha_ei()
 {
 }

/* ----------------------------------------------------------------

 * Node initialization functions
 *
 * ---------------------------------------------------------------- */

void nest::iaf_cond_alpha_ei::init_node_(const Node& proto)
{
  const iaf_cond_alpha_ei& pr = downcast<iaf_cond_alpha_ei>(proto)
      ;
  P_ = pr.P_;
  S_ = pr.S_;
}

void nest::iaf_cond_alpha_ei::init_state_(const Node& proto)
{
  const iaf_cond_alpha_ei& pr = downcast<iaf_cond_alpha_ei>(proto)
      ;
  S_ = pr.S_;
}

void nest::iaf_cond_alpha_ei::init_buffers_()
{
  Archiving_Node::clear_history();

  B_.spike_exc_.clear();        // includes resize
  B_.spike_inh_.clear();        // includes resize
  B_.currents_.clear();         // includes resize

  B_.logger_.init();

  B_.potentials_.clear_data();   // includes resize --- will
      disappear
  B_.conductances_.clear_data(); // includes resize --- will
      disappear

  B_.step_ = Time::get_resolution().get_ms();
  B_.IntegrationStep_ = B_.step_;

  V_.I_stim = 0.0;
}


void nest::iaf_cond_alpha_ei::calibrate()
{
  V_.PSConInit_E  = 1.0 * numerics::e / P_.tau_synE;
  V_.PSConInit_I  = 1.0 * numerics::e / P_.tau_synI;
  V_.RefractoryCounts = Time(Time::ms(P_.t_ref)).get_steps();

  assert(V_.RefractoryCounts >= 0);  // since t_ref >= 0, this can
      only fail in error

  // ---------------- from iaf_psc_alpha.calibrate() START
      -----------------

  // The resolution is divided by two because the updating, inside
      a time step,
  // is done at 0 and h/2. Then at the beginning of the next time
      step.
  const double_t h = Time::get_resolution().get_ms()/2;

  // these P are independent
  V_.P11_ex_ = V_.P22_ex_ = std::exp(-h/P_.tau_synE);
  V_.P11_in_ = V_.P22_in_ = std::exp(-h/P_.tau_synI);

  // these depend on the above. Please do not change the order.
```

```
  V_.P21_ex_ = h * V_.P11_ex_;
  V_.P21_in_ = h * V_.P11_in_;

  // ---------------- from iaf_psc_alpha.calibrate() STOP
       ----------------
}



inline
nest::double_t nest::iaf_cond_alpha_ei::dVdt (double_t V_m) const
{
  const double_t I_syn_exc = S_.y2_ex_ * ( V_m - P_.E_ex );
  const double_t I_syn_inh = S_.y2_in_ * ( V_m - P_.E_in );
  const double_t I_leak    = P_.g_L    * ( V_m - P_.E_L  );

  return (( - I_leak - I_syn_exc - I_syn_inh + V_.I_stim + P_.I_e
       ) / P_.C_m);
}


inline
void nest::iaf_cond_alpha_ei::synapse_half_step()
{ // Lines of code are from iaf_psc_alpha::update()

  //Updating excitatory conductances
  S_.y2_ex_  = V_.P21_ex_ * S_.y1_ex_ + V_.P22_ex_ * S_.y2_ex_;
  S_.y1_ex_ *= V_.P11_ex_;

  //Updating inhibitory conductances
  S_.y2_in_  = V_.P21_in_ * S_.y1_in_ + V_.P22_in_ * S_.y2_in_;
  S_.y1_in_ *= V_.P11_in_;
}



/*
     ----------------------------------------------------------------

 * Update and spike handling functions
 *
      ----------------------------------------------------------------
       */

void nest::iaf_cond_alpha_ei::update(Time const & origin, const
     long_t from, const long_t to)
{

  assert(to >= 0 && (delay) from < Scheduler::get_min_delay());
  assert(from < to);
```

```
const double_t h = B_.step_;

for ( long_t lag = from ; lag < to ; ++lag )
{
  const double_t k1 = h * dVdt(S_.V_ );

  // Updating the synapses h/2 forwards
  synapse_half_step();

  const double_t k2 = h * dVdt(S_.V_ + k1/2);
  const double_t k3 = h * dVdt(S_.V_ + k2/2);

  // Updating the synapses h/2 forwards
  synapse_half_step();

  const double_t k4 = h * dVdt(S_.V_ + k3);

  // Estimating the change in membrane potential
  const double_t deltaV_ = k1/6 + k2/3 + k3/3 + k4/6;
  S_.V_ = S_.V_ + deltaV_;

  // ---- From iaf_cond_alpha.cpp START -----

  // refractoriness and spike generation
  if ( S_.r )
  {// neuron is absolute refractory
          --S_.r;
          S_.V_ = P_.V_reset;  // clamp potential
  }
  else
    // neuron is not absolute refractory
    if ( S_.V_ >= P_.V_th )
    {
      S_.r  = V_.RefractoryCounts;
      S_.V_ = P_.V_reset;

      // log spike with Archiving_Node
      set_spiketime(Time::step(origin.get_steps()+lag+1));

      SpikeEvent se;
      network()->send(*this, se, lag);
    }

  // ---- From iaf_cond_alpha.cpp STOP -----


  S_.y1_ex_ += B_.spike_exc_.get_value(lag) * V_.PSConInit_E;
  S_.y1_in_ += B_.spike_inh_.get_value(lag) * V_.PSConInit_I;
```

```
    // set new input current
    V_.I_stim = B_.currents_.get_value(lag);

    // log state data
    B_.logger_.record_data(origin.get_steps() + lag);

    // voltage logging -- next two lines will disappear

    // With use of multimeter the tow following lines could be
        deleted
    B_.potentials_.record_data(origin.get_steps()+lag, S_.V_ );
    B_.conductances_.record_data(origin.get_steps()+lag,
                                 std::pair<double_t, double_t>(S_.
                                     y2_ex_, S_.y2_in_));

    // ----- From iaf_cond_alpha.cpp STOP -----
  }
}


void nest::iaf_cond_alpha_ei::handle(SpikeEvent & e)
{
  assert(e.get_delay() > 0);

  if(e.get_weight() > 0.0)
    B_.spike_exc_.add_value(e.get_rel_delivery_steps(network()->
        get_slice_origin()),
                            e.get_weight() * e.get_multiplicity() );
  else
    B_.spike_inh_.add_value(e.get_rel_delivery_steps(network()->
        get_slice_origin()),
                            -e.get_weight() * e.get_multiplicity() );
                                  // ensure conductance is positive
}

void nest::iaf_cond_alpha_ei::handle(CurrentEvent& e)
{
  assert(e.get_delay() > 0);

  // add weighted current; HEP 2002-10-04
  B_.currents_.add_value(e.get_rel_delivery_steps(network()->
      get_slice_origin()),
                            e.get_weight() * e.get_current());
}

void nest::iaf_cond_alpha_ei::handle(PotentialRequest& e)
{
  B_.potentials_.handle(*this, e);
}
```

```
void nest::iaf_cond_alpha_ei::handle(SynapticConductanceRequest& e
    )
{
  B_.conductances_.handle(*this, e);
}

void nest::iaf_cond_alpha_ei::handle(DataLoggingRequest& e)
{
  B_.logger_.handle(e);
}
```

iaf_cond_alpha_ei_step.h

```
/*
 *  iaf_cond_alpha_ei_step.h
 *
 *  This file is part of NEST
 *
 *  Copyright (C) 2005-2009 by
 *  The NEST Initiative
 *
 *  See the file AUTHORS for details.
 *
 *  Permission is granted to compile and modify
 *  this file for non-commercial use.
 *  See the file LICENSE for details.
 *
 */

#ifndef iaf_cond_alpha_ei_step_H
#define iaf_cond_alpha_ei_step_H

#include "config.h"

#include "nest.h"
#include "event.h"
#include "archiving_node.h"
#include "ring_buffer.h"
#include "connection.h"
#include "universal_data_logger.h"
#include "recordables_map.h"

// next line will disappear when all models support multimeter
#include "analog_data_logger.h"


/* BeginDocumentation
```

```
Name: iaf_cond_alpha_ei_step - Simple conductance based leaky
     integrate-and-fire neuron model.

Description:
iaf_cond_alpha_ei_step is an implementation of a spiking neuron
     using IAF dynamics with conductance-based synapses. Incoming
     spike events induce a post-synaptic change of conductance
     modelled by an alpha function. The alpha function is
     normalised such that an event of weight 1.0 results in a peak
     current of 1 nS at t = tau_syn.

Parameters:
The following parameters can be set in the status dictionary.

V_m         double - Membrane potential in mV
E_L         double - Leak reversal potential in mV.
C_m         double - Capacity of the membrane in pF
t_ref       double - Duration of refractory period in ms.
V_th        double - Spike threshold in mV.
V_reset     double - Reset potential of the membrane in mV.
E_ex        double - Excitatory reversal potential in mV.
E_in        double - Inhibitory reversal potential in mV.
g_L         double - Leak conductance in nS;
tau_ex      double - Rise time of the excitatory synaptic alpha
     function in ms.
tau_in      double - Rise time of the inhibitory synaptic alpha
     function in ms.
I_e         double - Constant input current in pA.

Sends: SpikeEvent

Receives: SpikeEvent, CurrentEvent, PotentialRequest,
     SynapticConductanceRequest

Author: Schrader, Plesser

SeeAlso: iaf_cond_exp, iaf_cond_alpha_mc
*/

namespace nest
{
   /**
   * Integrate-and-fire neuron model with two conductance-based
       synapses.
   *
   * @note Per 2009-04-17, this class has been revised to our
       newest
   *       insights into class design. Please use THIS CLASS as a
       reference
   *       when designing your own models with nonlinear dynamics.
   *       One weakness of this class is that it distinguishes
       between
   *       inputs to the two synapses by the sign of the synaptic
       weight.
   *       It would be better to use receptor_types, cf
       iaf_cond_alpha_mc.
   */
  class iaf_cond_alpha_ei_step : public Archiving_Node
  {

  // Boilerplate function declarations
     --------------------------------

  public:

    iaf_cond_alpha_ei_step();
    iaf_cond_alpha_ei_step(const iaf_cond_alpha_ei_step&);
    ~iaf_cond_alpha_ei_step();

    /*
     * Import all overloaded virtual functions that we
     * override in this class.  For background information,
     * see http://www.gotw.ca/gotw/005.htm.
     */

    using Node::connect_sender;
    using Node::handle;

    port check_connection(Connection&, port);

    port connect_sender(SpikeEvent &, port);
    port connect_sender(CurrentEvent &, port);
    port connect_sender(DataLoggingRequest &, port);

    // The next two lines will disappear once all models have been
    // equipped for use with multimeter. Do not include them in
    // new models!
    port connect_sender(PotentialRequest &, port);
    port connect_sender(SynapticConductanceRequest &, port);

    void handle(SpikeEvent &);
    void handle(CurrentEvent &);
    void handle(DataLoggingRequest &);
    void handle(PotentialRequest &);            // will disappear
       (multimeter)
    void handle(SynapticConductanceRequest &);  // will disappear
       (multimeter)

    void get_status(DictionaryDatum &) const;
    void set_status(const DictionaryDatum &);
```

```
private:
  void init_node_(const Node& proto);
  void init_state_(const Node& proto);
  void init_buffers_();
  void calibrate();
  void update(Time const &, const long_t, const long_t);


  // END Boilerplate function declarations
     ----------------------------

  // Friends
     --------------------------------------------------------

  // The next two classes need to be friends to access the
      State_ class/member
  friend class RecordablesMap<iaf_cond_alpha_ei_step>;
  friend class UniversalDataLogger<iaf_cond_alpha_ei_step>;

private:

  // Parameters class
     -----------------------------------------------

  //! Model parameters
  struct Parameters_ {
    double_t V_th;        //!< Threshold Potential in mV
    double_t V_reset;     //!< Reset Potential in mV
    double_t t_ref;       //!< Refractory period in ms
    double_t g_L;         //!< Leak Conductance in nS
    double_t C_m;         //!< Membrane Capacitance in pF
    double_t E_ex;        //!< Excitatory reversal Potential in
        mV
    double_t E_in;        //!< Inhibitory reversal Potential in
        mV
    double_t E_L;         //!< Leak reversal Potential (aka
        resting potential) in mV
    double_t tau_synE;    //!< Synaptic Time Constant Excitatory
         Synapse in ms
    double_t tau_synI;    //!< Synaptic Time Constant for
        Inhibitory Synapse in ms
    double_t I_e;         //!< Constant Current in pA


    Parameters_();        //!< Set default parameter values

    void get(DictionaryDatum&) const;  //!< Store current values
         in dictionary

    void set(const DictionaryDatum&);  //!< Set values from
        dicitonary
  };

  // State variables class
     --------------------------------------------

  /**
   * State variables of the model.
   *
   * State variables consist of the state vector for the
       subthreshold
   * dynamics and the refractory count. The state vector must be
       a
   * C-style array to be compatible with GSL ODE solvers.
   *
   * @note Copy constructor and assignment operator are required
       because
   *       of the C-style array.
   */
public:
  struct State_ {

    double_t V_; //!< Membrane potential

    double_t y1_ex_; //!< Excitatory conductance, derivative
    double_t y2_ex_; //!< Excitatory conductance

    double_t y1_in_; //!< Inhibitory conductance, derivative
    double_t y2_in_; //!< Inhibitory conductance


    //!< number of refractory steps remaining
    int_t     r;

    State_(const Parameters_&);  //!< Default initialization
    State_(const State_&);
    State_& operator=(const State_&);

    void get(DictionaryDatum&) const;  //!< Store current values
         in dictionary

    /**
     * Set state from values in dictionary.
     * Requires Parameters_ as argument to, eg, check bounds.'
     */
    void set(const DictionaryDatum&, const Parameters_&);


  };
```

```
private:

  // Buffers class
     ------------------------------------------------

  /**
   * Buffers of the model.
   * Buffers are on par with state variables in terms of
       persistence,
   * i.e., initalized only upon first Simulate call after
       ResetKernel
   * or ResetNetwork, but are implementation details hidden from
       the user.
   */
  struct Buffers_ {
    Buffers_(iaf_cond_alpha_ei_step&); //!<Sets buffer pointers
        to 0
    Buffers_(const Buffers_&, iaf_cond_alpha_ei_step&); //!<Sets
        buffer pointers to 0

    //! Logger for all analog data
    UniversalDataLogger<iaf_cond_alpha_ei_step> logger_;

    /** buffers and sums up incoming spikes/currents */
    RingBuffer spike_exc_;
    RingBuffer spike_inh_;
    RingBuffer currents_;

    // The next two lines will disappear once all models have
        been
    // equipped for use with multimeter. Do not include them in
    // new models!
    AnalogDataLogger<PotentialRequest>           potentials_;
    AnalogDataLogger<SynapticConductanceRequest> conductances_;

    // IntergrationStep_ should be reset with the neuron on
        ResetNetwork,
    // but remain unchanged during calibration. Since it is
        initialized with
    // step_, and the resolution cannot change after nodes have
        been created,
    // it is safe to place both here.
    double_t step_;           //!< step size in ms
    double   IntegrationStep_;//!< current integration time step
        , updated by GSL
  };

  // Variables class
     ------------------------------------------------

  /**
   * Internal variables of the model.
   * Variables are re-initialized upon each call to Simulate.
   */
  struct Variables_ {
    /**
     * Impulse to add to DG_EXC on spike arrival to evoke unit-
         amplitude
     * conductance excursion.
     */
    double_t PSConInit_E;

    /**
     * Impulse to add to DG_INH on spike arrival to evoke unit-
         amplitude
     * conductance excursion.
     */
    double_t PSConInit_I;

    //! refractory time in steps
    int_t    RefractoryCounts;

    //! make external input current available to dynamics
        function
    double_t I_stim;

    //! used for temporary saving of the values calculated by
        the
    //! Runge-Kutta method during a computational step
    double_t k1;
    double_t k2;
    double_t k3;
    double_t k4;
    double_t k5;
    double_t k6;

    //! Used to calculate the difference between fourth and
        fifth order Runge-Kutta method
    std::vector< double_t > ec;

    //! Propagator matrix used in exact integration (updating of
        synaptic dynamics)
    struct Propagator
    {
      double_t P21;
      double_t P11;
    };
```

```
  std::vector<Propagator> P_ex_; //!< Excitatory propagator
      matrix
  std::vector<Propagator> P_in_; //!< Inhibitory propagator
      matrix

  std::vector<double_t> k_delta_h; //!< Contains the test
      steps during a computational time step

  double_t y2_mid_ex;
  double_t y2_mid_in;

};

// Access functions for UniversalDataLogger
    ------------------------------

double_t get_V_() const {return S_.V_;} //<! Returns membrane
    potential
double_t get_g_ex_() const {return S_.y2_ex_;} //<! Returns
    excitatory conductance
double_t get_g_in_() const {return S_.y2_in_;} //<! Returns
    inhibitory conductance

//! Read out remaining refractory time, used by
    UniversalDataLogger
double_t get_r_() const { return Time::get_resolution().get_ms
    () * S_.r; }

//! Calculate and read out the difference betwee fourth and
    fifth order RK. Used by UniversalDataLogger
double_t get_V_error_() const{
  return V_.ec[1]*V_.k1 + V_.ec[3]*V_.k3 + V_.ec[4]*V_.k4 + V_
      .ec[5]*V_.k5 + V_.ec[6]*V_.k6;}

//! Updates the membrane potential so the Runge-Kutta method
    gets the rigt values.
double_t dVdt (int_t i, double_t V_m);

//! Updates the synapses, both the inhibitory and the
    excitatory
void updating_synapses();

//! Creates the synaptic values in the propagator matrices.
    One call for each step taken by the Runge-Kutta method.
void set_synapses_matrix(Variables_::Propagator & Pr, const
    double_t & deltah, const double_t &tau_syn);


// Data members
    ---------------------------------------------------------

  // keep the order of these lines, seems to give best
      performance
  Parameters_ P_;
  State_      S_;
  Variables_  V_;
  Buffers_    B_;

  //! Mapping of recordables names to access functions
  static RecordablesMap<iaf_cond_alpha_ei_step> recordablesMap_;
};


// Boilerplate inline function definitions
    ---------------------------------

inline
port iaf_cond_alpha_ei_step::check_connection(Connection& c,
    port receptor_type)
{
  SpikeEvent e;
  e.set_sender(*this);
  c.check_event(e);
  return c.get_target()->connect_sender(e, receptor_type);
}

inline
port iaf_cond_alpha_ei_step::connect_sender(SpikeEvent&, port
    receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  return 0;
}

inline
port iaf_cond_alpha_ei_step::connect_sender(CurrentEvent&, port
    receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  return 0;
}

inline
port iaf_cond_alpha_ei_step::connect_sender(DataLoggingRequest&
    dlr,
                                            port receptor_type)
{
```

```
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  B_.logger_.connect_logging_device(dlr, recordablesMap_);
  return 0;
}

inline
void iaf_cond_alpha_ei_step::get_status(DictionaryDatum &d)
    const
{
  P_.get(d);
  S_.get(d);
  Archiving_Node::get_status(d);

  (*d)[names::recordables] = recordablesMap_.get_list();
}

inline
void iaf_cond_alpha_ei_step::set_status(const DictionaryDatum &d
    )
{
  Parameters_ ptmp = P_;  // temporary copy in case of errors
  ptmp.set(d);                       // throws if BadProperty
  State_      stmp = S_;  // temporary copy in case of errors
  stmp.set(d, ptmp);                 // throws if BadProperty

  // We now know that (ptmp, stmp) are consistent. We do not
  // write them back to (P_, S_) before we are also sure that
  // the properties to be set in the parent class are internally
  // consistent.
  Archiving_Node::set_status(d);

  // if we get here, temporaries contain consistent set of
      properties
  P_ = ptmp;
  S_ = stmp;
}

// The next two functions will disappear once all models have
    been
// equipped for use with multimeter. Do not include them in
// new models!
inline
port iaf_cond_alpha_ei_step::connect_sender(PotentialRequest& pr
    , port receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  B_.potentials_.connect_logging_device(pr);
  return 0;
}

inline
port iaf_cond_alpha_ei_step::connect_sender(
    SynapticConductanceRequest& scr, port receptor_type)
{
  if (receptor_type != 0)
    throw UnknownReceptorType(receptor_type, get_name());
  B_.conductances_.connect_logging_device(scr);
  return 0;
}

} // namespace


#endif //iaf_cond_alpha_ei_step_H
```

iaf_cond_alpha_ei_step.cpp

```
/*
 *  iaf_cond_alpha_ei_step.cpp
 *
 *  This file is part of NEST
 *
 *  Copyright (C) 2005-2009 by
 *  The NEST Initiative
 *
 *  See the file AUTHORS for details.
 *
 *  Permission is granted to compile and modify
 *  this file for non-commercial use.
 *  See the file LICENSE for details.
 *
 */


#include "iaf_cond_alpha_ei_step.h"

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

```
#include "universal_data_logger_impl.h"
#include <limits>

#include <iomanip>
#include <iostream>
#include <cstdio>

/*
    ----------------------------------------------------------------

 * Recordables map
 *
     ----------------------------------------------------------------
      */

nest::RecordablesMap<nest::iaf_cond_alpha_ei_step> nest::
    iaf_cond_alpha_ei_step::recordablesMap_;

namespace nest   // template specialization must be placed in
    namespace
{
  /*
   * Override the create() method with one call to RecordablesMap
       ::insert_()
   * for each quantity to be recorded.
   */
  template <>
  void RecordablesMap<iaf_cond_alpha_ei_step >::create()
  {
    // use standard names whereever you can for consistency!
    insert_(names::V_m,
            &iaf_cond_alpha_ei_step::get_V_);
    insert_(names::g_ex,
            &iaf_cond_alpha_ei_step::get_g_ex_);
    insert_(names::g_in,
            &iaf_cond_alpha_ei_step::get_g_in_);

    insert_(names::t_ref_remaining,
            &iaf_cond_alpha_ei_step::get_r_);

    insert_("V_error",
            &iaf_cond_alpha_ei_step::get_V_error_);
  }
}

/*
    ----------------------------------------------------------------

 * Default constructors defining default parameters and state
 *
     ----------------------------------------------------------------
      */

nest::iaf_cond_alpha_ei_step::Parameters_::Parameters_()
  : V_th     (-55.0    ),  // mV
    V_reset  (-60.0    ),  // mV
    t_ref    (  2.0    ),  // ms
    g_L      ( 16.6667 ),  // nS
    C_m      (250.0    ),  // pF
    E_ex     (  0.0    ),  // mV
    E_in     (-85.0    ),  // mV
    E_L      (-70.0    ),  // mV
    tau_synE(  0.2    ),  // ms
    tau_synI(  2.0    ),  // ms
    I_e      (  0.0    )   // pA
{
  recordablesMap_.create();
}

nest::iaf_cond_alpha_ei_step::State_::State_(const Parameters_& p)
  : V_(p.E_L),
    y1_ex_(0),
    y2_ex_(0),
    y1_in_(0),
    y2_in_(0),
    r(0)
{
}

nest::iaf_cond_alpha_ei_step::State_::State_(const State_& s)
  : V_(s.V_),
    y1_ex_(s.y1_ex_),
    y2_ex_(s.y2_ex_),
    y1_in_(s.y1_in_),
    y2_in_(s.y2_in_),
    r(s.r)
{
}

nest::iaf_cond_alpha_ei_step::State_& nest::iaf_cond_alpha_ei_step
    ::State_::operator=(const State_& s)
{
  if ( this == &s )  // avoid assignment to self
    return *this;

  V_ = s.V_;
  y1_ex_ = s.y1_ex_;
  y2_ex_ = s.y2_ex_;
  y1_in_ = s.y1_in_;
```

```
  y2_in_ = s.y2_in_;
  r = s.r;

  return *this;
}

nest::iaf_cond_alpha_ei_step::Buffers_::Buffers_(
    iaf_cond_alpha_ei_step& n)
  : logger_(n)
{
  // The other member variables are left uninitialised or are
  // automatically initialised by their default constructor.
}

nest::iaf_cond_alpha_ei_step::Buffers_::Buffers_(const Buffers_&,
    iaf_cond_alpha_ei_step& n)
  : logger_(n)
  {
  // The other member variables are left uninitialised or are
  // automatically initialised by their default constructor.
  }

/*
    ----------------------------------------------------------------

 * Parameter and state extractions and manipulation functions
 *
    ----------------------------------------------------------------
     */

void nest::iaf_cond_alpha_ei_step::Parameters_::get(
    DictionaryDatum &d) const
{
  def<double>(d,names::V_th,          V_th);
  def<double>(d,names::V_reset,       V_reset);
  def<double>(d,names::t_ref,         t_ref);
  def<double>(d,names::g_L,           g_L);
  def<double>(d,names::E_L,           E_L);
  def<double>(d,names::E_ex,          E_ex);
  def<double>(d,names::E_in,          E_in);
  def<double>(d,names::C_m,           C_m);
  def<double>(d,names::tau_syn_ex,    tau_synE);
  def<double>(d,names::tau_syn_in,    tau_synI);
  def<double>(d,names::I_e,           I_e);
}

void nest::iaf_cond_alpha_ei_step::Parameters_::set(const
    DictionaryDatum& d)
{
  // allow setting the membrane potential
  updateValue<double>(d,names::V_th,     V_th);
  updateValue<double>(d,names::V_reset,  V_reset);
  updateValue<double>(d,names::t_ref,    t_ref);
  updateValue<double>(d,names::E_L,      E_L);

  updateValue<double>(d,names::E_ex,     E_ex);
  updateValue<double>(d,names::E_in,     E_in);

  updateValue<double>(d,names::C_m,      C_m);
  updateValue<double>(d,names::g_L,      g_L);

  updateValue<double>(d,names::tau_syn_ex, tau_synE);
  updateValue<double>(d,names::tau_syn_in, tau_synI);

  updateValue<double>(d,names::I_e,      I_e);

  if ( V_reset >= V_th )
    throw BadProperty("Reset potential must be smaller than
        threshold.");

  if ( C_m <= 0 )
    throw BadProperty("Capacitance must be strictly positive.");

  if ( t_ref < 0 )
    throw BadProperty("Refractory time cannot be negative.");

  if ( tau_synE <= 0 || tau_synI <= 0 )
    throw BadProperty("All time constants must be strictly
        positive.");

  const double_t ratio_res_tau = 10.0; // A global variable could
      be used instead
  const double_t res = Time::get_resolution().get_ms();
  if ( (res*ratio_res_tau > tau_synE)  || (res*ratio_res_tau >
      tau_synI) )
    throw BadProperty("Resolution too rough for the chosen time
        constants.");
}


void nest::iaf_cond_alpha_ei_step::State_::get(DictionaryDatum &d)
     const
{
  def<double>(d, names::V_m, V_); // Membrane potential
}

void nest::iaf_cond_alpha_ei_step::State_::set(const
    DictionaryDatum& d, const Parameters_&)
{
  updateValue<double>(d, names::V_m, V_);
```

```
}


/*
    ----------------------------------------------------------------

 * Default and copy constructor for node, and destructor
 *
    ----------------------------------------------------------------
      */

nest::iaf_cond_alpha_ei_step::iaf_cond_alpha_ei_step()
  : Archiving_Node(),
    P_(),
    S_(P_),
    B_(*this)
{
}

nest::iaf_cond_alpha_ei_step::iaf_cond_alpha_ei_step(const
    iaf_cond_alpha_ei_step& n)
  : Archiving_Node(n),
    P_(n.P_),
    S_(n.S_),
    B_(n.B_, *this)
{
}

 nest::iaf_cond_alpha_ei_step::~iaf_cond_alpha_ei_step()
 {
 }

/*
    ----------------------------------------------------------------

 * Node initialization functions
 *
    ----------------------------------------------------------------
      */

void nest::iaf_cond_alpha_ei_step::init_node_(const Node& proto)
{
  const iaf_cond_alpha_ei_step& pr = downcast<
      iaf_cond_alpha_ei_step>(proto);
  P_ = pr.P_;
  S_ = pr.S_;
}

void nest::iaf_cond_alpha_ei_step::init_state_(const Node& proto)
{
  const iaf_cond_alpha_ei_step& pr = downcast<
      iaf_cond_alpha_ei_step>(proto);
  S_ = pr.S_;
}

void nest::iaf_cond_alpha_ei_step::init_buffers_()
{
  Archiving_Node::clear_history();

  B_.spike_exc_.clear();        // includes resize
  B_.spike_inh_.clear();        // includes resize
  B_.currents_.clear();         // includes resize

  B_.logger_.init();

  B_.potentials_.clear_data();   // includes resize --- will
      disappear
  B_.conductances_.clear_data(); // includes resize --- will
      disappear

  B_.step_ = Time::get_resolution().get_ms();
  B_.IntegrationStep_ = B_.step_;

  V_.I_stim = 0.0;
}


void nest::iaf_cond_alpha_ei_step::calibrate()
{
  V_.PSConInit_E  = 1.0 * numerics::e / P_.tau_synE;
  V_.PSConInit_I  = 1.0 * numerics::e / P_.tau_synI;
  V_.RefractoryCounts = Time(Time::ms(P_.t_ref)).get_steps();

  assert(V_.RefractoryCounts >= 0);  // since t_ref >= 0, this can
       only fail in error

  const int_t steps = 6;

  // Values for test steps taken inside one computational time
      steps of length h
  V_.k_delta_h.resize(steps);
  V_.k_delta_h[0] = 0;
  V_.k_delta_h[1] = 1.0/4.0;
  V_.k_delta_h[2] = 3.0/8.0;
  V_.k_delta_h[3] = 1.0/2.0;
  V_.k_delta_h[4] = 12.0/13.0;
  V_.k_delta_h[5] = 1.0;

  V_.P_ex_.resize(steps);
  V_.P_in_.resize(steps);
```

```
  for(int_t i = 0; i< steps; ++i)
  {
    set_synapses_matrix(V_.P_ex_[i], V_.k_delta_h[i], P_.
        tau_synE);
    set_synapses_matrix(V_.P_in_[i], V_.k_delta_h[i], P_.
        tau_synI);
  }

  const double_t ratio_res_tau = 10.0;
  const double_t res = Time::get_resolution().get_ms();
  if ( (res*ratio_res_tau > P_.tau_synE)  || (res*ratio_res_tau >
      P_.tau_synI) )
    throw BadProperty("Resolution too rough for the chosen time
        constants.");

  V_.ec.resize(7);
  // Updating V_.ec with values from GSL 1.12.
  // Used to calculate difference between RK4 and RK5
  V_.ec[0]=  0.0;
  V_.ec[1]=  1.0/  360.0;
  V_.ec[2]=  0.0;
  V_.ec[3]= -128.0/  4275.0;
  V_.ec[4]= -2197.0/  75240.0;
  V_.ec[5]=  1.0/  50.0;
  V_.ec[6]=  2.0 /  55.0;
}

inline
void nest::iaf_cond_alpha_ei_step::set_synapses_matrix(Variables_
    ::Propagator & Pr, const double_t & deltah,
                                                       const
                                                           double_t
                                                           &
                                                           tau_syn
                                                           )
{
  const double_t h = Time::get_resolution().get_ms()*deltah;
  Pr.P11 = std::exp(-h/tau_syn);
  Pr.P21 = h * Pr.P11;
}


inline
nest::double_t nest::iaf_cond_alpha_ei_step::dVdt (int_t i,
    double_t V_m)
{
  V_.y2_mid_ex = V_.P_ex_[i].P21 * S_.y1_ex_ + V_.P_ex_[i].P11 *
      S_.y2_ex_;
  V_.y2_mid_in = V_.P_in_[i].P21 * S_.y1_in_ + V_.P_in_[i].P11 *
      S_.y2_in_;

  const double_t I_syn_exc = V_.y2_mid_ex * ( V_m - P_.E_ex );
  const double_t I_syn_inh = V_.y2_mid_in * ( V_m - P_.E_in );
  const double_t I_leak    = P_.g_L    * ( V_m - P_.E_L  );

  return ((- I_leak - I_syn_exc - I_syn_inh + V_.I_stim + P_.I_e
      ) / P_.C_m);
}


inline
void nest::iaf_cond_alpha_ei_step::updating_synapses()
{// Lines of code are from iaf_psc_alpha::update()
  //Updating excitatory conductances
  S_.y2_ex_  = V_.P_ex_[5].P21 * S_.y1_ex_ + V_.P_ex_[5].P11 * S_.
      y2_ex_;
  S_.y1_ex_ *= V_.P_ex_[5].P11;

  //Updating inhibitory conductances
  S_.y2_in_ = V_.P_in_[5].P21 * S_.y1_in_ + V_.P_in_[5].P11 * S_.
      y2_in_;
  S_.y1_in_ *= V_.P_in_[5].P11;
}

/*
 ----------------------------------------------------------------
 * Update and spike handling functions
 *
 ----------------------------------------------------------------
    */

void nest::iaf_cond_alpha_ei_step::update(Time const & origin,
    const long_t from, const long_t to)
{
  assert(to >= 0 && (delay) from < Scheduler::get_min_delay());
  assert(from < to);

  const double_t h = B_.step_;

  // ___ From gsl-1.12/ode-initval/rkf45.c
  static const double c1 = 902880.0/7618050.0;
  static const double c3 = 3953664.0/7618050.0;
  static const double c4 = 3855735.0/7618050.0;
```

```
static const double c5 = -1371249.0/7618050.0;
static const double c6 = 277020.0/7618050.0;

static const double ah[] = { 1.0/4.0, 3.0/8.0, 12.0/13.0, 1.0,
    1.0/2.0 };
static const double b3[] = { 3.0/32.0, 9.0/32.0 };
static const double b4[] = { 1932.0/2197.0, -7200.0/2197.0,
    7296.0/2197.0};
static const double b5[] = { 8341.0/4104.0, -32832.0/4104.0,
    29440.0/4104.0, -845.0/4104.0};
static const double b6[] = { -6080.0/20520.0, 41040.0/20520.0,
    -28352.0/20520.0, 9295.0/20520.0, -5643.0/20520.0};
// --- END

for ( long_t lag = from ; lag < to ; ++lag )
{
  // taking the test steps. All with base at the beginning of
      the computational time step.
  V_.k1 = h * dVdt(0, S_.V_); // 0
  V_.k2 = h * dVdt(1, S_.V_ + ah[0]*V_.k1); // 1/4
  V_.k3 = h * dVdt(2, S_.V_ + b3[0]*V_.k1 + b3[1]*V_.k2); // 3/8
  V_.k4 = h * dVdt(4, S_.V_ + b4[0]*V_.k1 + b4[1]*V_.k2 + b4[2]*
      V_.k3); // 12/13
  V_.k5 = h * dVdt(5, S_.V_ + b5[0]*V_.k1 + b5[1]*V_.k2 + b5[2]*
      V_.k3 + b5[3]*V_.k4); // 1
  V_.k6 = h * dVdt(3, S_.V_ + b6[0]*V_.k1 + b6[1]*V_.k2 + b6[2]*
      V_.k3 + b6[3]*V_.k4 + b6[4]*V_.k5); // 1/2

  const double_t deltaV_ = c1*V_.k1 + c3*V_.k3 + c4*V_.k4 + c5*
      V_.k5 + c6*V_.k6;
  S_.V_ = S_.V_ + deltaV_;

  // Takes the synapses one computatinal time step forwards.
  updating_synapses();

  // refractoriness and spike generation
  if ( S_.r )
  {// neuron is absolute refractory
      --S_.r;
      S_.V_ = P_.V_reset;  // clamp potential
  }
  else
    // neuron is not absolute refractory
    if ( S_.V_ >= P_.V_th )
    {
      S_.r  = V_.RefractoryCounts;
      S_.V_ = P_.V_reset;

      // log spike with Archiving_Node
      set_spiketime(Time::step(origin.get_steps()+lag+1));
      SpikeEvent se;
      network()->send(*this, se, lag);
    }

  S_.y1_ex_ += B_.spike_exc_.get_value(lag) * V_.PSConInit_E;
  S_.y1_in_ += B_.spike_inh_.get_value(lag) * V_.PSConInit_I;

  // set new input current
  V_.I_stim = B_.currents_.get_value(lag);

  // log state data
  B_.logger_.record_data(origin.get_steps() + lag);

  // voltage logging -- next two lines will disappear

  // With use of multimeter the tow following lines could be
      deleted
  B_.potentials_.record_data(origin.get_steps()+lag, S_.V_ );
  B_.conductances_.record_data(origin.get_steps()+lag,
                               std::pair<double_t, double_t>(S_.
                                   y2_ex_, S_.y2_in_));
  }
}


void nest::iaf_cond_alpha_ei_step::handle(SpikeEvent & e)
{
  assert(e.get_delay() > 0);

  if(e.get_weight() > 0.0)
    B_.spike_exc_.add_value(e.get_rel_delivery_steps(network()->
        get_slice_origin()),
                            e.get_weight() * e.get_multiplicity() );
  else
    B_.spike_inh_.add_value(e.get_rel_delivery_steps(network()->
        get_slice_origin()),
                            -e.get_weight() * e.get_multiplicity() );
                                  // ensure conductance is positive
}

void nest::iaf_cond_alpha_ei_step::handle(CurrentEvent& e)
{
  assert(e.get_delay() > 0);

  // add weighted current; HEP 2002-10-04
  B_.currents_.add_value(e.get_rel_delivery_steps(network()->
      get_slice_origin()),
                         e.get_weight() * e.get_current());
}
```

```
void nest::iaf_cond_alpha_ei_step::handle(PotentialRequest& e)
{
  B_.potentials_.handle(*this, e);
}

void nest::iaf_cond_alpha_ei_step::handle(
    SynapticConductanceRequest& e)
{
  B_.conductances_.handle(*this, e);
}

void nest::iaf_cond_alpha_ei_step::handle(DataLoggingRequest& e)
{
  B_.logger_.handle(e);
}
```

Test scripts

class_Params.py

```
# Class for use in master thesis

import numpy as np

#--------------------------------------------------
#--------------------------------------------------
class Params:
    def __init__(self, mdl='iaf_cond_alpha', mpt=500,
                 simtime=500.0, vec = np.arange(0,-15,-1)):
        self.g_L = 15.
        self.C_m = 120.
        self.I_e = 4*self.g_L
        self.E_L = -70.0
        self.tau_m = self.C_m/self.g_L
        self.tau_synE = 0.2
        self.Tmax = simtime
        self.mpt = mpt
        self.deltaT = self.Tmax / self.mpt
        self.h_log2= vec
        self.model = mdl
        self.recparams = self.recparams_(mdl)

#--------------------------------------------------
    def recparams_(self, mdl):
        if mdl =='iaf_psc_alpha':
            return ['V_m']
        if mdl == 'iaf_cond_alpha' or mdl == 'iaf_cond_alpha_ei':
            return ['V_m', 'g_ex', 'g_in']
        if mdl == 'iaf_cond_alpha_ei_step':
            return ['V_m', 'g_ex', 'g_in', 'V_error']

#--------------------------------------------------
#--------------------------------------------------
```

define_objects.py

```
import numpy as np
import cPickle
import os

import class_Params as P
p = P.Params() # Standard parameter set

#--------------------------------------------------
def define_objects(mpt_ = p.mpt, simtime_ = p.Tmax, vector_= p.
    h_log2):
    models = return_modelnames()
    # Produce four neuron models with values
    p_        = P.Params(mdl=models[0], mpt=mpt_, simtime=
        simtime_, vec=vector_)#, tau_synE)
    p_ei      = P.Params(mdl=models[1], mpt=mpt_, simtime=
        simtime_, vec=vector_)#, tau_synE)
    p_psc     = P.Params(mdl=models[2], mpt=mpt_, simtime=
        simtime_, vec=vector_)#, tau_synE)
    p_ei_step = P.Params(mdl=models[3], mpt=mpt_, simtime=
        simtime_, vec=vector_)#, tau_synE)

    Tmax_mpt = '_%05.0f_%05.0f_' % (p_.Tmax, p_.mpt)

    reg = return_regimetypes()

    nsp = [reg[0]]
      # nsp[0]: Indicates that the simulation is in non-spiking
          regime

    # Standard values for wno and wfn test regime
    rate_ =[80000.0, 20000.0]
    p_gen_ = 2
    conn_str_ = [np.abs(p_.g_L/p_.E_L), -4.0]
    delay_ = 1.0

    wno  = [reg[1], rate_, p_gen_, conn_str_, delay_]
    wfn =  [reg[3], rate_, p_gen_, conn_str_, delay_]
      # wnX[0]: Indicates that noise should be produced with
          Poisson generators
      #         If wnf, the noise is discretized in 1 ms intervals
          .
      # wnX[1]: List of rates for excitatory and inhibitory
          Poisson generators
      # wnX[2]: The number of Poisson generators
      # wnX[3]: The strength of the connections with the neurons,
      #         respectively excitatory and inhibitory
      # wnX[4]: The delay of the transmission of signals

    spt = [100. ,200., 300. , 400.]
    esp = [reg[2], spt, 1]
      # esp[0]: Indicates that extra spikes should be produced (
          spike generators)
```

```
      # esp[1]: Array with spike times
      # esp[2]: The number of spike generators

    return [p_, p_ei, p_psc, p_ei_step],[ nsp, wno, esp, wfn],
        Tmax_mpt

#--------------------------------------------------------
def return_modelnames():
    # The different neuron models used
    return ['iaf_cond_alpha', 'iaf_cond_alpha_ei',
            'iaf_psc_alpha',  'iaf_cond_alpha_ei_step']

#--------------------------------------------------------
def return_regimetypes():
    # The different test regimes used
    return ['nsp', 'wno', 'esp', 'wfn']

#--------------------------------------------------------
def return_mpt_simtime():
    # Standard number of measuring points and simulation time
    return p.mpt, p.Tmax

#--------------------------------------------------------
def return_vector():
    # Standard resolution vector
    return p.h_log2

#--------------------------------------------------------
def return_tau_synE():
    # Standard value of tau_synE
    return p.tau_synE

#--------------------------------------------------------
def return_E_L():
    # Standard value of E_L
    return p.E_L

#--------------------------------------------------------
def return_rates():
    # Firing rates for stress test with different rates
    rex = 8.0
    rin = rex/ 4.0
    rates_10 = np.array([0.1, 1.0, 10, 25, 50, 75])
    rates = 1000 * rates_10
    return rex, rin, rates

#--------------------------------------------------------
def return_tau_syn():
    # Variation of the excitatory synaptic time constant
    return np.arange(0.1, 1.01, 0.1)
```

```
#--------------------------------------------------------
def return_path(name):
    # Return path and creates it if not excisting
    path = "/home/anders/Mastergradsoppgave/NestODE/LaTeX/
        CodeForThesis/Figures/" +name
    path_exists_create(path)
    return path

#--------------------------------------------------------
def return_resvec():
    # Used to plot resolutions exluded the first zero,
    #  the first three and the first six
    return [0, 3, 6]

#--------------------------------------------------------
def return_power2_vec(vec):
    return 2.0**vec # vec is a numpy.arange

#--------------------------------------------------------
def max_recparams(types):
    # Find the maximum number of recordable params in different
        neuron models
    length = 1
    for i in range(len(types)):
        new_len = len(types[i].recparams)
        if new_len > length:
            length = new_len
    return length

#--------------------------------------------------------
def save_binary_file(filename_path, M):
    #From Ch. 4.3.6, "Python Scripting for Computational Science",
         H.P.Langtangen
    outfile = open(filename_path + '.dat', 'wb')
    cPickle.dump(M, outfile)
    outfile.close()

#--------------------------------------------------------
def load_binary_file(filename_path):
    #From Ch. 4.3.6, "Python Scripting for Computational Science",
         H.P.Langtangen
    infile = open(filename_path+ '.dat','rb')
    M = cPickle.load(infile)
    infile.close()
    return M

#--------------------------------------------------------
def path_exists_create(path):
    # Chech if path exists. If it does not it will be created.
```

```
    str = ''
    for i in range(len(path)):
        str += path[i]
        if path[i] == '/' and os.path.exists(str)==False:
            os.mkdir(str)

#-------------------------------------------------------
#-------------------------------------------------------
```

update_sizes.py

```
import matplotlib
from matplotlib import pylab
from matplotlib.pyplot import *

#-------------------------------------------------------
def adjust_axis(axis, limits=[-7,7]):
# Ensures scientific notation of the axis for values larger than
#  10^limits[1] and smaller than 10^limits[0]
        if axis == 'x':
                gca().xaxis.get_major_formatter().set_powerlimits(
                    limits)
        if axis == 'y':
                gca().yaxis.get_major_formatter().set_powerlimits(
                    limits)
        draw()

#-------------------------------------------------------
def update_sizes_tex(legendsize, textsize):
# The following lines adjusts many sizes, e.g. the legend size
# From Jochen Martin Eppler, 16.03.10
        params = {'axes.labelsize':  10,
                  'text.fontsize':   textsize,
                  'legend.fontsize': legendsize,
                  'xtick.labelsize': 10,
                  'ytick.labelsize': 10,
                  'text.usetex':     False}
        pylab.rcParams.update(params)

#-------------------------------------------------------
#-------------------------------------------------------
```

simulate.py

```
import numpy as np

import nest
nest.sli_run("M_WARNING setverbosity")
#Now NEST just print to display error messages and warnings, not
     info messages.

import define_objects as defobj

#-------------------------------------------------------
#-------------------------------------------------------
def get_values(i):
# Returns simulation values for resolutions finer than i
    types, regimes, times = defobj.define_objects(vector_ = defobj
        .return_vector()[i:])
    M, T = simulate_all(types, regimes)
    return M, T, types, regimes

#-------------------------------------------------------
def simulate_all(types, regimes):
# Simulates all neuron models and all test regimes
    mpt = types[0].mpt
    len_recparams =  defobj.max_recparams(types)

    X =  np.zeros((len(types), len(regimes), len_recparams, mpt-1,
         len(types[0].h_log2)))
    for i in range(len(types)):
        for j in range(len(regimes)):
            X[i][j], T = simulate(types[i], regimes[j],
                len_recparams)

    return X, T

#-------------------------------------------------------
#-------------------------------------------------------
def exact_values_nsp(p, T):
# Calculate exact values for the non-spiking test regime
    V_exact=   p.E_L   + ((p.I_e*p.tau_m)/p.C_m)*(- np.expm1(-T/p.
        tau_m))
    return V_exact

#-------------------------------------------------------
def create_noise(M, neuron):
# Produces noise from Poisson gen. to the wno test regime
    # M[1]: Excitatory and inhibitory rates
    # M[2]: Number of poisson generators
    # M[3]: Connection strength, excitatory and inhibitory
    # M[4]: Delay for the transmission of signals
```

```
    noise = nest.Create("poisson_generator_ps", M[2], [{'rate': M
        [1][0]}, {'rate': M[1][1]}] )
    nest.ConvergentConnect(noise, neuron, M[3], M[4])

#-------------------------------------------------------
def create_frozen_noise(M, neuron, Tsim, seed=12345678):
    # Function created by Hans Ekkehard Plesser
    """
    Create Poisson spike trains discretized to 1ms intervals and
        add to
    simulation as excitatory and inhibitory noise.
    M:  M[1] is [ex_rate in_rate] in Hz
    neuron: neuron to connect to
    Tsim: total simulation time
    seed: by default, use fixed seed for RNG
    """
    np.random.seed(seed)

    noise = nest.Create("spike_generator", M[2])
    # M[2]: Number of poisson generators
    for ng, rate in [(noise[0], M[1][0]), (noise[1], M[1][1])]:
        # M[1]: Excitatory and inhibitory rates

        # convert rate from Hz to 1/ms
        rate_ms = rate / 1000.
        # generous estimate of spikes in Tsim
        n_est = 100 + int(1.5 * Tsim * rate_ms)

        # fill array with interspike intervals til we reach Tsim
        # store exact values
        spks = np.array([])
        while sum(spks) < Tsim:
            # add more spike times to array
            nisi = np.random.exponential(1./rate_ms, n_est)
            spks = np.append(spks, nisi)

        # now set spike generator with cumulative sum rounded up
            to ms
        nest.SetStatus([ng], {'spike_times': np.ceil(spks.cumsum()
            )})

    # connect generators to neuron
    nest.ConvergentConnect(noise, neuron, M[3], M[4])
    # M[3]: Connection strength, excitatory and inhibitory
    # M[4]: Delay for the transmission of signals

#-------------------------------------------------------
def create_extra_spikes(M, neuron):
# Creates extra spikes for the esp test regime
    spg = nest.Create('spike_generator', M[2], [{'spike_times': M
        [1] }] )
    # M[1]: Spike times
    # M[2]: Number of spike generators
    nest.ConvergentConnect(spg, neuron)

#-------------------------------------------------------
def simulate(p, M, len_recparams):
# Simulate actual neuron model and test regime
    val = np.zeros([len_recparams, p.mpt -1, len(p.h_log2)])
    for i in range(len(p.h_log2)):
        nest.ResetKernel()
        nest.SetKernelStatus({'tics_per_ms': 2.0**14, '
            resolution': 2.0**(p.h_log2[i])})
        nest.SetDefaults('voltmeter', {'interval': p.deltaT})

        if p.model == 'iaf_psc_alpha' or p.model == '
            iaf_psc_alpha_canon':
            psc_params =  {"tau_m": p.tau_m, "C_m":p.C_m, "I_e":
                p.I_e,
                           "E_L": p.E_L, "tau_syn_ex":p.tau_synE
                               }
            neuron = nest.Create(p.model, params = psc_params)


        if (p.model == 'iaf_cond_alpha' or p.model == '
            iaf_cond_alpha_ei' or p.model=='
            iaf_cond_alpha_ei_step'):
            cond_params = {"g_L": p.g_L, "C_m":p.C_m, "I_e": p.
                I_e,
                           "E_L": p.E_L, "tau_syn_ex":p.tau_synE
                               }
            neuron = nest.Create(p.model, params= cond_params)

        mm_params = {'withtime': True, 'interval': p.deltaT,
                     'record_from': p.recparams}
        multimeter = nest.Create('multimeter',params = mm_params
            )
        nest.Connect(multimeter, neuron)

        if M[0]=='wno':
            create_noise(M, neuron)

        if M[0]=='wfn':
            create_frozen_noise(M, neuron, p.Tmax)

        if M[0]=='esp':
            create_extra_spikes(M, neuron)

        nest.Simulate(p.Tmax)
```

```
            events = nest.GetStatus(multimeter)[0]['events']
            for j in range(len(p.recparams)):
                val[j][:,i] = events[p.recparams[j]]

        T_ = events['times']
        T = resize_T_(p,T_)

        return val,T

#-------------------------------------------------------
def resize_T_(p, T_):
    T = np.zeros([p.mpt-1, 1])
    for i in range(p.mpt -1):
          T[i, 0] = T_[i]
    return T

#-------------------------------------------------------
#-------------------------------------------------------
#-------------------------------------------------------
```

example_figs.py

```
import numpy as np
from matplotlib.pyplot import *

import define_objects as defobj
import simulate as sim
import update_sizes as upd
upd.update_sizes_tex(10,10)

#-------------------------------------------------------
#-------------------------------------------------------
def calculate_plot():
# Simulate and produce all figures possible with script
    M, T, types, regimes = do_simulations()
    path = defobj.return_path('example_figures/')
    make_ex_fig(M, T, types, regimes, path)
    wno_figs(M, T, types, path)
    nsp_diff(M, T, types, path)

#-------------------------------------------------------
def do_simulations():
# Run the simulation with all neuron models and all test regimes
    types, regimes, times = defobj.define_objects()
    M, T = sim.simulate_all(types, regimes)
    return M, T, types, regimes

#-------------------------------------------------------
#-------------------------------------------------------
def make_ex_fig(M, T, types, regimes, path):
# Produces example figure with all four test regimes
    res = 6
    # Iaf_cond_alpha, membrane potential,
    #  resolution 2**(-6)ms
    V = M[0,:,0,:, res]
    res_str = '_res%+03.0f' % res

    figure()
    for i in range(np.shape(V)[0]):
        plot(T, V[i], label = regimes[i][0])
    legend(loc= 'center right')
    xlabel('Time [ms]')
    ylabel('Membrane potential [mV]')
    savefig(path + 'All_regimes' + res_str + '.eps')
    savefig(path + 'All_regimes' + res_str + '.pdf')

#-------------------------------------------------------
#-------------------------------------------------------
def wno_figs(M, T, types, path):
# Produce figures of wno test regime
#  Only conductance based models, all resolutions
    mod_nr=[0,1,3]# Only conductance based models
    # Plot membrane potential in the with-noise test regime
    for i in range(len(mod_nr)):
        figure()
        plot(T, M[mod_nr[i], 1, 0])
        legend(types[0].h_log2)
        xlabel('Time [ms]')
        ylabel('Membrane potential [mV]')
        savefig(path + 'V_M_wno_' + types[mod_nr[i]].model + '.eps
            ')
        savefig(path + 'V_M_wno_' + types[mod_nr[i]].model + '.pdf
            ')

#-------------------------------------------------------
#-------------------------------------------------------
def nsp_diff(M, T, types, path):
# Makes figure of difference between exact and simulated values
#  for all neuron models.
    res = 6
    exact = sim.exact_values_nsp(types[0], T)
    # Difference over all models, nsp-regime, membrane potential,
    #  all measuring points, all resolutions
    diff = M[:,0,0,:,:] - exact
    for i in range(np.shape(diff)[0]):
```

```
        figure()
        plot(T, diff[i, :, res:])
        legend(types[0].h_log2[res:])
        xlabel('Time [ms]')
        ylabel('Voltage difference [mV]')
        savefig(path + 'Diff_Vm_nsp' + types[i].model + '.eps')
        savefig(path + 'Diff_Vm_nsp' + types[i].model + '.pdf')

#--------------------------------------------------------
#--------------------------------------------------------
```

wfn_error.py

```
from matplotlib.pyplot import *
import numpy as np

import define_objects as defobj
import simulate as sim

import update_sizes as update
update.update_sizes_tex(10, 10)

#--------------------------------------------------------
#--------------------------------------------------------
def wfn_all():
    X, T, vec, rate_ex, mcp = wfn_error_rate()
    diff9, V, T, vec, types = wfn_error_types()

#--------------------------------------------------------
def wfn_error_rate():
# Tests how the error from iaf_cond_alpha_ei_step influences
#  of changing firing rates from the Poisson generators

    types, regimes, times = defobj.define_objects()
    regimes_ = [regimes[3]] # wfn-regime  (with frozen noise)
    types_ = [types[3]] # Only iaf_cond_alpha_ei_step

    rex, rin, rates = defobj.return_rates()
    rec_p = len(types_[0].recparams)
    vec = types_[0].h_log2

    x0 = len(rates)
    x1 = types_[0].mpt-1
    x2 = len(vec)

    X= np.zeros([x0, x1, x2])
    for i in range(x0):
        regimes_[0][1] = [rex*rates[i], rin*rates[i]]
        val , T =  sim.simulate(types_[0] , regimes_[0], rec_p)
        X[i] = val[3] # Assumes recparams[3] = 'V_error' for
            ei_step

    mcp = np.abs(types_[0].E_L * 10**(-16))
    rate_ex = rex*rates

    plot_X(X, vec, rate_ex, mcp)

    return X, T, vec, rate_ex, mcp

#--------------------------------------------------------
def plot_X(X, vec, rate_ex, mcp):
    path = defobj.return_path('wfn_error/')
    x0 = np.shape(X)[0]
    figure()
    for i in range(x0):
        semilogy(vec, np.median(X[i], axis=0), 'x--')

    # Red line at machine precision
    plot([vec[-1], vec[0]], [mcp, mcp], 'r-')
    legend(rate_ex, loc ='upper left')
    xlabel('Resolution (log$_2$ $h$)')
    ylabel('Error [mV]')
    savefig(path + 'iaf_cond_alpha_ei_step_Ratetest_wfn_Verror.eps
        ')
    savefig(path + 'iaf_cond_alpha_ei_step_Ratetest_wfn_Verror.pdf
        ')

#--------------------------------------------------------
#--------------------------------------------------------
#--------------------------------------------------------
def wfn_error_types():
# Produces figures of the difference between a reference solution
#  and all other resolutions and models.
#  wfn test regime

    types, regimes, times = defobj.define_objects()
    regimes_ = [regimes[3]] # wfn-regime  (with frozen noise)

    rec_p = len(types[0].recparams)
    vec = types[0].h_log2

    M, T = sim.simulate_all(types, regimes_)
    V = M[:,0,0,:,:] # Only wnf-regime, only membrane potential

    # Uses aaf_cond_alpha_ei_step at resolution log2 h = -9
    #  as a reference solution
    ref9  = V[3, :, 9].reshape([types[0].mpt -1, 1])
```

```
    diff9 = V - ref9

    path = defobj.return_path('wfn_error/')
    plot_error_types(diff9[:, 100:, 6:], T[100:], vec[6:], '9_',
        types, path)
    plot_error_types(diff9[:, 100:, 9:], T[100:], vec[9:], '9_',
        types, path)

    cond = [diff9[0], diff9[1], diff9[3]]   # Excludes
        iaf_psc_alpha
    types_ = [types[0], types[1], types[3]] # Excludes
        iaf_psc_alpha
    med = np.median(np.abs(cond), axis=1)
    plot_loglogdiff(med, types_, vec, path)

    return diff9, V, T, vec, types

#-------------------------------------------------------
def plot_error_types(diff, T, vec, x, types, path):
    vec_str = '_res%+03.0f%+03.0f' % (vec[0], vec[-1])
    for i in range(np.shape(diff)[0]):
        figure()
        plot(T, diff[i])
        legend(vec)
        xlabel('Time [ms]')
        ylabel('Difference in membrane potential [mV]')
        savefig(path + 'Diff' + x + types[i].model + vec_str + '.
            eps')
        savefig(path + 'Diff' + x + types[i].model + vec_str + '.
            pdf')

#-------------------------------------------------------
def plot_loglogdiff(med, types, vec, path):
    figure()
    for i in range(np.shape(med)[0]): # Models
        semilogy(vec, med[i], 'x--',label = types[i].model)
    legend()
    xlabel('Resolution (log$_2$ $h$)')
    ylabel('Median value of difference in membrane potential [mV
        ]')
    vec_str = '_res%+03.0f%+03.0f' % (vec[0], vec[-1])
    savefig(path + 'Loglog_med_diff_all_cond_models' + vec_str +
        '.eps')
    savefig(path + 'Loglog_med_diff_all_cond_models' + vec_str +
        '.pdf')

#-------------------------------------------------------
#-------------------------------------------------------
#-------------------------------------------------------
```

wno_test.py

```
import numpy as np
from matplotlib.pyplot import *

import define_objects as defobj
import simulate as sim

import update_sizes as update
update.update_sizes_tex(10.0, 10.0)

#-------------------------------------------------------
#-------------------------------------------------------
def calculate_plot(type):
    # Possible types: 'rate' or 'tau_syn'
    X, T, types, regimes = wno_stress_test(type)

    vector = types[0].h_log2
    plot_stress_test(X, T, regimes, types, vector, type)

#-------------------------------------------------------
def wno_stress_test(type):
# Calculate the membrane potential for all neuron models
#  for different firing rates.

    types, regimes, times = defobj.define_objects()
    regimes_ = [regimes[1]]
    types_ = types
    rex, rin, rates = defobj.return_rates()
    syn = defobj.return_tau_syn()

    if type=='rate':
        len_t = len(rates)
    elif type=='tau_syn':
        len_t = len(syn)

    X= np.zeros([len(types_), len_t, types_[0].mpt - 1, len(types_
        [0].h_log2)])
    for i in range(len(types_)):
        for j in range(len_t):
            if type=='rate':
                regimes_[0][1] = [ rex*rates[j], rin*rates[j]]
            elif type== 'tau_syn':
                types_[i].tau_synE = syn[j]
            val , T =  sim.simulate(types_[i] , regimes_[0], len(
                types_[i].recparams))
            X[i][j] = val[0] # Assumes recparams[0] = 'V_m' for
                all regimes
```

```
    return X, T, types_, regimes_

#-------------------------------------------------------
def plot_stress_test(X, T, regimes, types, vector, rate_tau):
    labels = ['Time [ms]', 'Membrane potential [mV]']
    path_all = defobj.return_path('stress_test/')

    # All rates for one resolution in a figure. One figure for
        each model.
    res = [4,8]
    for h in range(len(res)):
        for i in range(np.shape(X)[0]):
            all_one_fig(X[i], types[i], path_all, vector, res[h],
                labels, rate_tau)

#-------------------------------------------------------
def all_one_fig(X, types, path, vec, nr, labels, rate_tau):
    # Plots one model; all rates or time constants for
    #  one resolution in one figure
    rex, rin, rates = defobj.return_rates()
    syn = defobj.return_tau_syn()
    figure()
    clf()
    for i in range(np.shape(X)[0]):
        plot(X[i][:, nr])

    res = '%+03.0f' % vec[nr]
    figtitle = types.model + '_wno_' + rate_tau + '_all_res' + res
    if rate_tau == 'rate':
        legend(rex*rates)
    elif rate_tau =='tau_syn':
        legend(syn)
        xlabel(labels[0])
        ylabel(labels[1])
    savefig(path + figtitle + '.eps')
    savefig(path + figtitle + '.pdf')

#-------------------------------------------------------
#-------------------------------------------------------
#-------------------------------------------------------
```

comparison.py

```
import time
import numpy as np
from matplotlib.pyplot import *
import define_objects as defobj
import speed_brunel as sb

#-------------------------------------------------------
#-------------------------------------------------------
def run_all(only_cond = 0, last = 0):
    res_vec_std = [-2, -4]#, -6, -10, -14]
    models_std = defobj.return_modelnames()
    rep = 3 # repetitions per resolution and model

    if only_cond:
        if last ==0:
            # Simulate only iaf_cond_alpha exept finest resolution
            models = [models_std[0]]
            res_vec = res_vec_std[:-1] # All except h=2^(-14)
        if last ==1:
            # Simulate only iaf_cond_alpha at finest resolution
            models = [models_std[0]]
            res_vec = [res_vec_std[-1]] # Only h=2^(-14)
            rep = 1
    else:
        # iaf_cond_alpha_ei, iaf_psc_alpha, iaf_cond_alpha_ei_step
        models = models_std[1:]
        res_vec = res_vec_std

    before = time.time()

    val, M = compare(rep, models, res_vec)

    after = time.time()
    delta = after-before
    print '\n\nTotal simulation time: ', delta/60.0 , 'minutes'

    save_matrices(M, val, res_vec, models)
    print_results(M, res_vec, models, rep)

    return val, M

#-------------------------------------------------------
def vec_to_string(res_vec):
    res_str = '_res'
    for i in range (len(res_vec)):
        str = '%+03.0f' % (res_vec[i])
        res_str += str
    return res_str

#-------------------------------------------------------
def return_comptypes():
    return ['Mean', 'Med', 'Min', 'Max']
```

```
#-------------------------------------------------
def compare(rep, models, res_vec):
    var = 3 # Simulation time, excitatory an inhibitory rate
         should be recorded
    len_ct = len(return_comptypes())
    M = np.zeros([len(models), len(res_vec), var, len_ct])
    val = np.zeros([len(models), len(res_vec), var, rep ])
    s = np.shape(M)
    for i in range(s[0]):
        print '\n\nSimulating ', models[i]
        before = time.time()
        for j in range(s[1]):
            val[i,j], M[i,j] = simulate_compare(rep, var, len_ct,
                models[i], res_vec[j])
            print "Finished resolution ", res_vec[j]
        after = time.time()
        delta = after - before
        print 'Finished simulating ', models[i]
        print 'Simulated in ', delta/60.0 , 'minutes'
    return val, M

#-------------------------------------------------
def simulate_compare(rep, var, comp_len, model, res_log):
    # For resolution log2(h) = -4:
    # psc_alpha:          g=19.05, eta=6.75   gives firing rate at
          28.2 (ex) and 28.4 (in)
    # cond_alpha_ei:      g=2.0,   eta=0.0015 gives firing rate at
          28.7 (ex) and 28.9 (in)
    # cond_alpha_ei_step: g=2.0,   eta=0.0015 gives firing rate at
          28.7 (ex) and 28.4 (in)
    # cond_alpha:         g=2.0,   eta=0.0015 gives firing rate at
          28.7 (ex) and 28.4 (in)

    val = np.zeros([var, rep])
    res = 2**(res_log)

    g_  = [19.05, 2.0]
    eta_= [6.75,  0.0015]

    if model == 'iaf_psc_alpha':
        g = g_[0]
        eta = eta_[0]
    else:
        g = g_[1]
        eta = eta_[1]

    for i in range(rep):
        simt, re, ri = sb.brunel_alpha_numpy(model, res , g, eta)
        print "Finished repetition nr. ", i+1, " of ", rep
        val[0][i] = simt
        val[1][i] = re
        val[2][i] = ri

    M = np.zeros([var, comp_len])
    for i in range(np.shape(val)[0]):
        M[i] =  [np.mean(val[i]), np.median(val[i]), np.min(val[i
            ]), np.max(val[i])]

    return val, M

#-------------------------------------------------
#-------------------------------------------------
def save_matrices(M, val, res_vec, models):
    str_res = vec_to_string(res_vec)
    str_mod =''
    for i in range(len(models)):
        str_mod += models[i][4:]
        if i < len(models)-1:
            str_mod += '-'
    str = str_mod + str_res

    str_val   = 'val-' + str
    str_M     = 'M-'   + str

    defobj.save_binary_file(str_val, val)
    defobj.save_binary_file(str_M, M)

#-------------------------------------------------
def print_results(M, res_vec, models, rep):
    typename = ['Simulation time [s]:','Exc. rate [Hz]:','Inh.
         rate [Hz]:']
    mean_med_min_max = return_comptypes()
    s = np.shape(M)

    for i in range(s[0]): #Models
        filename = models[i] + vec_to_string(res_vec)
        file = open(filename + '.txt', 'w')

        print >> file, "\nSIMULATING RESULTS"
        print >> file, "Model    \t",  models[i]
        print >> file, 'Repetitions\t', rep

        for j in range(s[1]): # Resolutions
            print >> file, '\nResolution [ms]\t', '2^%+1.0f' %
                res_vec[j]
            for k in range(s[2]): # Recorded parameters
                print >> file, '\n', typename[k]
                for l in range(s[3]): # Comp_types
                    print >> file, '\t', mean_med_min_max[l],'\t
                        %.2f' % M[i][j][k][l]
```

```
#-------------------------------------------------
def loglog_fig_file(path):
# This function is written and thought to used when matrices with
    simulation values for all
# four models exists as files in the directory given in "path".
#  One file for iaf_cond_alpha at resolution (-2,-4,-6,-10)
#  One file for iaf_cond_alpha at resolution (-14)
#  One file for the three other models at resolution
    (-2,-4,-6,-10)

    # Upload files from disc
    M_cond = defobj.load_binary_file(path +'M-cond_alpha_res
        -02-04-06-10')
    M_cond_l = defobj.load_binary_file(path +'M-cond_alpha_res
        -14')
    M_ = defobj.load_binary_file(path + 'M-cond_alpha_ei-psc_alpha
        -cond_alpha_ei_step_res-02-04-06-10-14')

    # Combine M_cond and M_cond_l
    s = np.shape(M_cond)
    X = np.zeros([s[0], s[1]+1, s[2], s[3]])
    X[0,:-1] = M_cond[0]
    X[0, -1] = M_cond_l[0,0]

    res_vec = [-2,-4,-6,-10,-14]

    models = defobj.return_modelnames()
    loglog = ['semilog', 'loglog']
    for i in range(len(loglog)):
        figure()

        if loglog[i]=='semilog':
            plot(res_vec, X[0,:,0,1], 'x--')
        else:
            semilogy(res_vec, X[0,:,0,1], 'x--')

        s = np.shape(M_)
        for j in range(s[0]):
            if loglog[i]=='semilog':
                plot(res_vec, M_[j,:,0,1], 'x--')
            else:
                semilogy(res_vec, M_[j,:,0,1], 'x--')

        legend(models)
        xlabel('Resolution (log$_2$ h)')
        ylabel('Time [s]')

        figpath = defobj.return_path('comparision/')
        savefig(figpath + 'Four_models-simtime_' + loglog[i] + '.
            eps')
        savefig(figpath + 'Four_models-simtime_' + loglog[i] + '.
            pdf')

#-------------------------------------------------
#-------------------------------------------------
```

speed_brunel.py

```
# COMMENT:
# This script is based on NEST/src/pynest/examples/brunel-alpha-
    numpy.py
# The main change is that I have made one extra function of it.
# It is then possible to send values to other test scripts.
# A.G. Jahnsen, 16.04.10

#-------------------------------------------------
import numpy
from numpy import exp
import time

import nest
nest.sli_run("M_WARNING setverbosity")
#Now NEST just print to display error messages and warnings, not
    info messages.

import class_Params as P
# Makes it possible to create a neuron with standard values

#-------------------------------------------------
#-------------------------------------------------
def LambertWm1(x):
    nest.sli_push(x); nest.sli_run('LambertWm1'); y=nest.sli_pop()
    return y

#-------------------------------------------------
def ComputePSPnorm(tauMem, CMem, tauSyn):
  """Compute the maximum of postsynaptic potential
     for a synaptic input current of unit amplitude
     (1 pA)"""

  a = (tauMem / tauSyn)
  b = (1.0 / tauSyn - 1.0 / tauMem)

  # time of maximum
  t_max = 1.0/b * ( -LambertWm1(-exp(-1.0/a)/a) - 1.0/a )
```

```
  # maximum of PSP for current of unit amplitude
  return exp(1.0)/(tauSyn*CMem*b) * ((exp(-t_max/tauMem) - exp(-
      t_max/tauSyn)) / b - t_max*exp(-t_max/tauSyn))

#-----------------------------------------------------
def brunel_alpha_numpy(model, resolution, g_, eta_):
    nest.ResetKernel()

    dt      = resolution    # the resolution in ms
    simtime = 1000.0        # Simulation time in ms
    delay   = 1.5           # synaptic delay in ms

    # Parameters for asynchronous irregular firing
    g       = g_
    eta     = eta_
    # Original values: g = 5.0, eta = 2.0
    epsilon = 0.1    # connection probability

    order     = 200
    NE        = 4*order
    NI        = 1*order
    N_neurons = NE+NI
    N_rec     = 50 # record from 50 neurons

    CE    = epsilon*NE   # number of excitatory synapses per
        neuron
    CI    = epsilon*NI   # number of inhibitory synapses per
        neuron
    C_tot = int(CI+CE)  # total number of synapses per neuron

    # Initialize the parameters of the integrate and fire neuron
    tauSyn = 0.5
    CMem = 250.0
    tauMem = 20.0
    theta  = 20.0
    J      = 0.1 # postsynaptic amplitude in mV

    # normalize synaptic current so that amplitude of a PSP is J
    J_ex  = J / ComputePSPnorm(tauMem, CMem, tauSyn)
    J_in  = -g*J_ex

    # threshold rate, equivalent rate of events needed to
    # have mean input current equal to threshold
    nu_th  = (theta * CMem) / (J_ex*CE*numpy.exp(1)*tauMem*tauSyn)
    nu_ex  = eta*nu_th
    p_rate = 1000.0*nu_ex*CE

    nest.SetKernelStatus({'tics_per_ms': 2.0**14, "resolution": dt
        })

# Building the network

# Creates a parameter set with actual model and standard
    values
p = P.Params(mdl = model)

# Picks out necessary parameters dependent of model type
if model == 'iaf_psc_alpha' or model == 'iaf_psc_alpha_canon':
    neuron_params = {"tau_m": p.tau_m, "C_m":p.C_m, "I_e": p.
        I_e,
                     "E_L": p.E_L, "tau_syn_ex":p.tau_synE}

if model == 'iaf_cond_alpha' or model == 'iaf_cond_alpha_ei'
    or model=='iaf_cond_alpha_ei_step':
    neuron_params = {"g_L": p.g_L, "C_m":p.C_m, "I_e": p.I_e,
                     "E_L": p.E_L, "tau_syn_ex": p.tau_synE}

nodes_ex=nest.Create(model,NE, params = neuron_params)
nodes_in=nest.Create(model,NI, params = neuron_params)

nest.SetDefaults("poisson_generator",{"rate": p_rate})
noise=nest.Create("poisson_generator")

espikes=nest.Create("spike_detector")
ispikes=nest.Create("spike_detector")

nest.SetStatus([espikes],[{"label": "brunel-py-ex",
                           "withtime": True,
                           "withgid": True}])

nest.SetStatus([ispikes],[{"label": "brunel-py-in",
                           "withtime": True,
                           "withgid": True}])

# Connecting devices

nest.CopyModel("static_synapse","excitatory",{"weight":J_ex, "
    delay":delay})
nest.CopyModel("static_synapse","inhibitory",{"weight":J_in, "
    delay":delay})

nest.DivergentConnect(noise,nodes_ex,model="excitatory")
nest.DivergentConnect(noise,nodes_in,model="excitatory")

nest.ConvergentConnect(range(1,N_rec+1),espikes,model="
    excitatory")
nest.ConvergentConnect(range(NE+1,NE+1+N_rec),ispikes,model="
    excitatory")

# Connecting network
```

```
    # Here, we create the connections from the excitatory neurons
        to all other
    # neurons. We exploit that the neurons have consecutive IDs,
        running from
    # 1,...,NE           for the excitatory neurons and from
    # (NE+1),...,(NE+NI) for the inhibitory neurons.

    numpy.random.seed(1234)

    sources_ex = numpy.random.random_integers(1,NE,(N_neurons,CE))
    sources_in = numpy.random.random_integers(NE+1,N_neurons,(
        N_neurons,CI))

    # We now iterate over all neuron IDs, and connect the neuron
        to
    # the sources from our array. The first loop connects the
        excitatory neurons
    # and the second loop the inhibitory neurons.

    for n in xrange(N_neurons):
        nest.ConvergentConnect(list(sources_ex[n]),[n+1],model="
            excitatory")

    for n in xrange(N_neurons):
        nest.ConvergentConnect(list(sources_in[n]),[n+1],model="
            inhibitory")

    # Shows the time progress. Makes it possible to see when the
        simulating stops if it do so.
#     nest.SetStatus([0],{'print_time': True})

    startsimulate = time.time()
    nest.Simulate(simtime)
    endsimulate = time.time()

    sim_time   = endsimulate-startsimulate

    events_ex = nest.GetStatus(espikes,"n_events")[0]
    rate_ex   = events_ex/simtime*1000.0/N_rec
    events_in = nest.GetStatus(ispikes,"n_events")[0]
    rate_in   = events_in/simtime*1000.0/N_rec

    return sim_time, rate_ex, rate_in

#-------------------------------------------------
#-------------------------------------------------
#-------------------------------------------------
```

spike_pattern.py

```
import numpy as np
from matplotlib.pyplot import *
import define_objects as defobj

#-------------------------------------------------

# Produces example figure which show that it is possible
#  with equal firing rate and different spike pattern

#-------------------------------------------------

figure()

# Equal distance between spikes
x1 = [0,   0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2]
y1 = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
plot(x1, y1, 'rx')

# One long time interval, then two small
x2 = [0,   0.6, 0.8, 1.4, 1.6, 2.2, 2.4, 3.0, 3.2]
y2 = [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]
plot(x2, y2, 'bx')

# One long time interval, then three small
x3 = [0,   0.6, 1.2, 1.4, 1.6, 2.2, 2.8, 3.0, 3.2]
y3 = [0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]
plot(x3, y3, 'gx')

xticks(np.arange(0, 3.5, 0.4))
yticks([])

xlim([0.0, 3.2])
ylim(0,0.4)

show()

path = defobj.return_path('spike_pattern/')
defobj.path_exists_create(path)

savefig(path + 'different_patterns_same_frequency.eps')
savefig(path + 'different_patterns_same_frequency.pdf')
```

Bibliography

N. Brunel. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of Computational Neuroscience*, 8:183–208, 2000.

J. C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, Ltd, second edition, 2008.

P. Dayan and L. F. Abbott. *Theoretical Neuroscience; Computational and Mathematical Modeling of Neural Systems*. The MIT Press, Massachusetts Institute of Technology, 2001.

J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.O. Gewaltig. PyNEST: A convenient interface to the NEST simulator. *Frontiers in Neuroinformatics*, 2:12, 2009. doi: 10.3389/neuro.11.012.2008.

iaf_cond_alpha.h. Declaration file for the iaf_cond_alpha neuron model in NEST; revision 8594, April 2010. URL `www.nest-initiative.org`.

iaf_psc_alpha.h. Declaration file for the iaf_psc_alpha neuron model in NEST; revision 8607, April 2010. URL `www.nest-initiative.org`.

E. R. Kandel, J. H. Schwartz, and T. M. Jessell. *Principles of Neural Science*. McGraw-Hill, fourth edition, 1991.

A. Morrison, S. Straube, H. E. Plesser, and M. Diesmann. Exact Subthreshold Integration with Continuous Spike Times in Discrete-Time Neural Network Simulations. *Neural Computation*, 19:47–79, 2007.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes, The Art of Scientific Computing*. Cambrigde University Press, third edition, 2007.

S. Rotter and M. Diesmann. Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological Cybernetics*, 81:381–402, 1999.

R. F. Thompson. *The Brain: A Neuroscience Primer*. Worth Publishers, third edition, 2000.

J. S. Vandergraft. *Computer Science and Applied Mathematics: Introduction to Numerical Computations.* Academic Press, second edition, 1983.