SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Firing patterns and multistability in a neuron model

av

Johan Klint

2014 - No 27

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

Firing patterns and multistability in a neuron model

Johan Klint

Självständigt arbete i matematik 15 högskolepoäng, Grundnivå

Handledare: Yishao Zhou

2014

Firing patterns and multistability in a neuron model

Johan Klint
2014

Abstract

Neuron cells are the basic unit of neural systems, and a prerequisite for all animal behavior and human cognition. The complexity of a neural structure, such as the human brain, is truly remarkable and any deeper aspects of its function may therefore appear difficult to grasp. Understanding a single neuron's function, with complex dendrite structure with up to 150 000 branches, connections to up to 10 000 other neurons and axon extensions more than a meter long, is a great challenge for the scientific community. However, with the use of modern computer technology, the electrophysical and mathematical mechanisms behind neuron signal generation can be visualized. In this work, the leech heart interneuron is modelled, and parameter ranges producing autonomous action potentials with different firing patterns were explored. A Java program was developed as a working tool for investigating this Hodgkin-Huxley type of model. The outcome from five explicit, including Euler and Runge-Kutta fourth order, and one semi-implicit numerical method was compared and their stability properties were discussed. The parameter space was explored with respect to the maximum membrane conductance of sodium, calcium and leak ions, with the parameters denoted gNa, gCaS and gleak, respectively. One intriguing feature of this neuron model is the appearance of multistability, where the same parameter settings may generate different signal output from the neuron model depending on the initial conditions of the state variables. This was visualized by simulations and phase portraits.

Contents

INTRODUCTION

The neuron and aim

Neurons are specialized cells for generating, integrating and/or transmitting electric signals in multicellular organisms, and are the fundamental components for e.g. function of the human brain and subsystems in the nervous system such as the control of heart rhythm. Neurons are often connected in highly intricate networks, where transmission of electrical signals between the neuronal cells is obtain by the action potentials, also called spikes. Action potentials are fast changes in the membrane potential, which is the electrical charge separation between inside and outside of the insulating cell membrane, and propagate in the cell membrane along tube-like cell extensions called axons, and work as a means for long-distance communication within the organism.

The changing aspects of electrical potential across the cell membrane of neurons can be mathematically modelled as dynamical systems. Earlier groundbreaking studies of the squid giant axon (Hodgkin and Huxley et al. 1952) generated mathematical models with behaviors highly resembling that of natural neuron cells, and in addition predicted results not included in the process of constructing the equations.

Firing patterns of neuron cells are characterized by the temporal distribution of monitored action potential spikes. Each kind of neuron produces firing patterns characteristic for that neuron cell type. One common firing pattern is tonic spiking, or regular spiking, when the spikes occurs at regular intervals with a certain frequency. Another common pattern is the burst, loosely defined as periods during which spike frequency is relatively high, separated by periods during which frequency is relatively low or spikes are absent altogether.

Making the situation more complex, experimental observations of biological neuron preparations have revealed that the same neuron cell under different conditions may show different action potential firing patterns, such as quiescence, tonic spiking and bursting (Cocatre-Zilgien and Delcomyn, 1992). Mathematical analyses of neuron models have resolved that bifurcation mechanisms are involved in the generation of action potentials, and transitions between different firing patterns when the same neuron cell exhibit more than one firing pattern (Izhikevich, 2000).

By studying neuron models from simpler organisms such as the medical leech (*Hirudo medicinalis*), we may gain deeper insights into the general mechanisms of neuron activity. The understanding obtained from a simpler system may help us understand more complex nervous systems, such as the neurons in the human brain, where also ethical reasons may restrict the opportunity to obtain experimental data. Several models of the leech heart interneuron have been developed (Hill et al., 2001, Malashchenko et al., 2011, Shilnikov et al., 2005 and 2008) using Hodgkin-Huxley type of equations. These models successfully produce the bursting firing pattern seen in nature and in addition, as parameters are varied, reproduce action potentials in a range of spiking patterns.

The aim of this study was to develop a computer software tool for investigating the parameter space of a mathematical neuronal model (Malashchenko et al., 2011). Different numerical methods for simulating neuron activity were analyzed in order to obtain reliable output from the model. The results obtained from simulations using this software, were discussed with an attempt to understand mechanisms behind the emergence of different output signals, and in addition the occurrence of multistability was observed.

The action potential

In neurons, there is an active ion transport over the cell membrane sustaining a concentration difference in specific ions inside the cell compared to outside. This results in a net electrical charge separation over the insulating cell membrane, typically resulting in a voltage difference of approximately -50 mV, i.e. more negative inside the cell compared to outside. This ion and charge separation, together with the voltage-gated ion channels, is the driving force in the generation and propagation of action potentials.

The action potential is characterized by an initial rapid electrical discharge of the membrane potential, where the default negative voltage difference between the inside and outside of the cell suddenly takes on positive values. Soon after this discharge, the membrane potential returns to its negative resting value. This voltage spike in the membrane propagates from the triggering zone, along the axon and reaches synapses where a release of signal molecules functions as a bridge over the synaptic cleft for the signal to the reach and initiate responses in the target cell, which for example could be triggering of an action potential in another neuron cell or contraction in a muscle cell.

The travelling of an action potential along the axon is enabled by specific properties of ionic permeability through the neuronal cell membrane. The main ions involved are Na, K and Ca (Fig. 1) which cross the membrane through voltage gated channels, with different ion specificity. These channels are large molecular structures inserted in the cell membrane, creating a passage for the ions through the hydrophic and otherwise non-ion permeable phospholipid membrane. Also a leak current is usually considered in neuron models. For the leak current, the ions involved are usually not specified, but it represents a flow of ions through non-gated channels, i.e. channels not affected by the membrane potential.

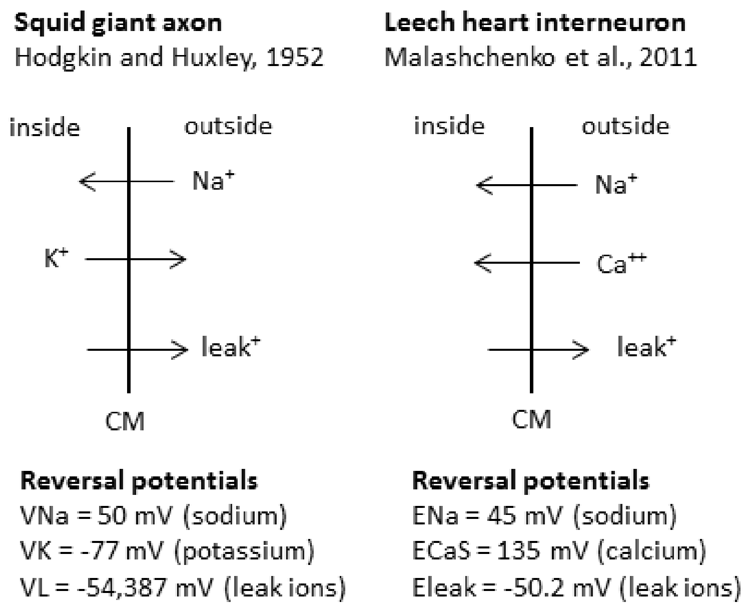

Figure 1. A comparison of the ion currents involved in the action potential of the squid giant axon and the leech heart interneuron models. The ions flow through ion channels inserted in the cell membrane (CM). For the squid giant axon, reversal values from the original paper (Hodgkin and Huxley, 1952) have been adjusted with -65 mV to get a scale with a negative resting value. The leak ions are presented in the figure as a univalent cation, however, they may as well be anions flowing in the opposite direction or be multivalent.

The generation of spikes may be driven by different mechanisms in different neurons. Often a spike is generated as a result of combined signal input from other neurons. Spikes may also be autonomously generated within a neuron cell, without direct input from outer neurons, which is the case for the leech heart interneuron. These autonomously generated spikes may come in different patterns, such as in a regular tonic spiking or in clusters of spikes denoted bursts, as mentioned above.

The leech heart interneuron

The ease by which the neuronal system is prepared from the medical leech explains the extensive use of it as a model organism in neuroscience since the late 19th century. An extensive review of the results obtained from more than 100 years of studies on the leech neuronal network is presented by Kristan et al. 2005. The medical leech has a relatively simple nervous system with approximately 10,000 neurons, most of which are found in the 21 segmental ganglia, which contain 400 neurons each, most of them occurring in pairs.

In biology, a central pattern generator (CPG) is a neural network that produces a rhythmic output signal without sensory feedback. The heart interneurons in a leech form a CPG and are located in mutually inhibitory pairs in the anterior part of the leech, in ganglia 3 and 4. These interneurons act as pacemakers, constituting the core of the leech heartbeat timing network. The pairwise connections of the heart interneuron can pharmacologically be broken by addition of bicuculline. These pharmacologically decoupled interneurons retain endogenous dynamics with bursting activities, showing that being connected in a neuronal network is not required for creating bursting activity, and thereby proves that these neurons are able to autonomously generate the bursting signal pattern (Cymbalyuk et al. 2002).

The bicuculline decoupled neuron is sensitive to disturbance, which was apparent when attempting to intracellularly measure membrane potential. When inserting the electrode into the cell, a small leak current is introduced. This stopped the bursting activity, whereas non-disruptive extracellular recording techniques preserve the autonomous spiking (Cymbalyuk et al. 2002). This indicates that the leak current is of particular interest when investing the bursting activity of these neurons.

Different regimes of neuron bursting/silence

The pacemaker activity of the small network of heart interneurons is obtained by repetitive firing of spikes, or bursting, occurring at regular time intervals. The signal is received by motor neurons as inhibitory signals, relaxing the lateral heart muscles. Concomitantly, the respective heart interneuron in the opposite side of same ganglion, also receives the inhibitory signal, which in turn will inhibit its spiking, and results in activation of the motor neurons on this side of the leech. In this manner, the both sides of the leech heart are timely coordinated, with alternate synchronous and peristaltic contractions and a phase of approximately 10 seconds (Kristan et al. 2005).

A neuron cell can exhibit different regimes of signals: bursting, tonic spiking and subthreshold oscillations and silence. The co-occurrence of several attracting regimes, i.e. multistability, is a common phenomenon in neuron cell models, and has previously been studied in models of the leech heart interneuron (Malashchenko et al., 2011, Cymbalyuk et al., 2005). A requirement for the occurrence of multistability is the underlying complexity in the Hodgkin-Huxley type of electrophysiological mechanisms in neuron cells, i.e. multivariable and non-linearity.

Bursting is characterized by a limited time of rapid oscillatory activity resulting in groups of spikes separated by intervals of quiescence, which is apparent in a leech heart neuron model (Fig. 2a). It is a common phenomenon in CPGs, which for example is involved in motor control. One mechanism allowing for bursting activity is the occurrence is interplaying ionic currents which are voltage-gated on various timescales.

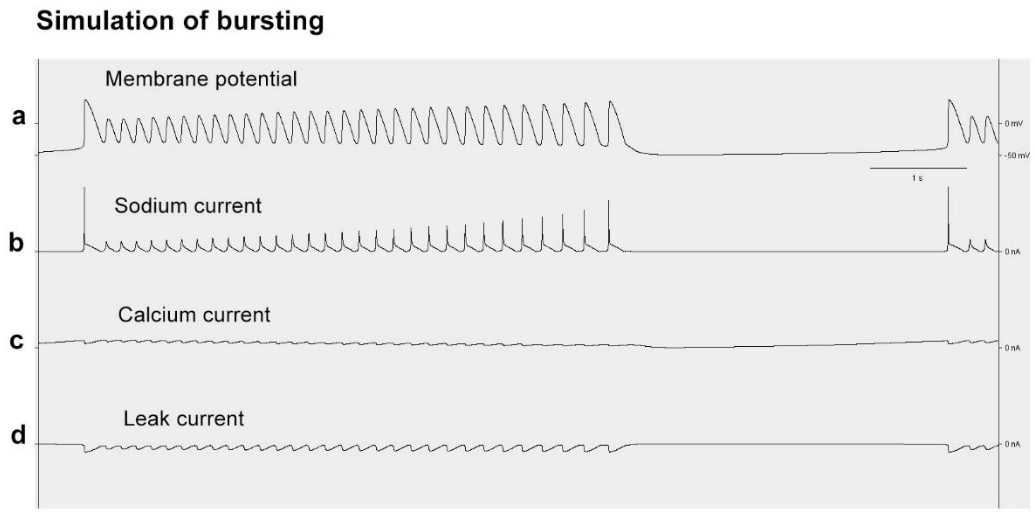

Figure 2. Membrane potential and transmembrane currents occurring during the onset of a neuronal burst. The membrane potential (a) shows a sequence of spikes forming a burst. Here a spike train of 31 action potentials, lasting for approximately 4.5 s, is followed by an interburst interval of approximately 3 s. The sodium currents (b), where sodium ions flows through voltage gated channels into the cell, causes the initial abrupt increase in membrane potential. The calcium current (c) slowly builds up during the interburst intervals, and at the end of the interval, triggers the rapid onset of sodium current. The leak current (d) repolarizes the membrane potential back to its resting potential.

During the burst in a leech heart interneuron model, sodium current (Fig. 2b) peaks result in rapid depolarization of the membrane potential. The initial sodium current peak is triggered by a calcium current (Fig. 2c) slowly building up during the interbursting intervals. Both sodium and calcium currents results from the inflow of positive ions into the cells. In order to repolarize the membrane, back to its resting state, a necessity is the leak current (Fig. 2d), representing the only electric current out from the cell in this model. As above mentioned, this leak current could be positive ions flowing out as well as negative ions flowing into the cell through non-gated ion channels, and is not further specified.

Hodgkin and Huxley equations

The work of Hodgkin and Huxley (1952) is widely conceived as outstanding and lead them to being awarded the Nobel Prize in Physiology or Medicine 1963. They created a mathematical model of the excitable neuron cell as an electric-circuit analog, with a conductive capacitor as an analogue to the cell membrane, and resistors as analogues to the different iron currents through voltage gated ion channels in the cell membrane.

Their neuron model is mathematically expressed as a series of differential equations describing the charge separation between the inside and outside of the cell membrane, and the current derived by flow of ions through gated channels located inside cell membrane. The opening and closing of the ion channels respond dynamically to the membrane voltage.

The Hodgkin-Huxley equations can be expressed as following (modified from Hodgkin and Huxley, 1952, Edelstein-Keshet, 1988):

$$\frac{dV}{dt} = \frac{1}{C_M}\left[I - \overline{gNa}m^3h(V - V_{Na}) - \overline{gK}n^4(V - V_K) - \overline{gl}(V - V_l)\right]$$

$$\frac{dm}{dt} = \alpha_m(1 - m) - \beta_m m$$

$$\frac{dh}{dt} = \alpha_h(1 - h) - \beta_h h$$

$$\frac{dn}{dt} = \alpha_n(1 - n) - \beta_n n$$

The differential equations describe the transmembrane voltage (V) as it is changing, and m, h, n ∈ [0, 1]. The variables m, h, n describe the sodium activation, the sodium inactivation and the potassium activation respectively, affecting the conductivity of the ion channels, selectively permeable for the corresponding ion. C_M is the membrane capacitance.

The constants $\overline{gNa}$, $\overline{gK}$, and $\overline{gl}$ relate to the theoretical maximum of conductance through the corresponding ion, and is therefore an indirect measure of the density of functional channels in the cell membrane. The current expressed as the variable *I* represents an input current that is introduced experimentally, and does not correspond to a current in a non-manipulated neuron. The voltage dependent functions α and β were tailored by Hodgkin and Huxley to fit the experimental data.

$$\alpha_m = \frac{0.1(25 - V)}{e^{\frac{25-V}{10}} - 1}$$

$$\beta_m = 4e^{-V/18}$$

$$\alpha_h = 0.07e^{-V/20}$$

$$\beta_h = 1/e^{\frac{V+30}{10}+1}$$

$$\alpha_n = \frac{0.01(10 - V)}{e^{\frac{10-V}{10}-1}}$$

$$\beta_n = 0.125e^{-V/80}$$

The most prominent property of this model is its ability to generate the all or nothing signal, described above as action potential or spike.

The model is generated form measurements in the squid giant axon, but in modified versions the same type of equations can be fitted to describe membrane potential dynamics in other types of neurons and in other organisms.

The variability of the ionic conductance, represented as constants such as $\overline{gNa}$, $\overline{gK}$, and $\overline{gl}$, in identified neurons with stereotypical function may vary through different developmental stages within an individual or between different individuals in a population. It has previously been shown that this variation may result in variable output from the same type of neurons with different amount of ionic channels in the cytoplasmic membrane, but the signal may as well be similar despite the variability in ionic conductance (Golewash, 2014). Because of the natural variability in the ionic conductance between cells of the same neuron type, it was of interest to monitor the effect of changing the parameter of ionic conductance on the output firing pattern in the neuron model of the current study.

METHODS

The heart interneuron model

The leech heart interneuron has been modelled in using Hodgkin-Huxley type of equations. A canonical model was constructed by Hill et al. 2001, where the ionic currents are modelled in 14 dimensions. This model was simplified to a four-dimensional model by Malaschenko et al. (2011) where the voltage-dependent ionic currents were presented as the fast sodium (*I_{Na}*), slow calcium (*I_{Ca}*) and leak (*I_{leak}*) currents. Although expressed slightly different, and with other variable name, this model is constructed in a similar way as that of Hodgkin and Huxley. This model worked as the basis for the current study:

$$\frac{dV}{dt} = \frac{1}{C_M}[-\overline{g_{Na}} f_\infty^3(-150, 0.028, V) h_{Na}[V - E_{Na}] - \bar{g}_{CaS} m_{CaS}^2 h_{CaS}[V - E_{CaS}] - g_{leak}[V - E_{leak}]]$$

$$\frac{dh_{Na}}{dt} = [f_\infty(500, B_h, V) - h_{Na}]/0.0405$$

$$\frac{dm_{CaS}}{dt} = [f_\infty(-420, 0.0472, V) - m_{CaS}]/\tau_{mCaS}$$

$$\frac{dh_{CaS}}{dt} = [f_\infty(360, B_{hCaS}, V) - h_{CaS}]/\tau_{hCaS}$$

The voltage-dependent activation or inactivation of ion permeability (i.e. opening and closing of ion channels) is modelled with the function $f_\infty(A, B, V)$ given by

$$f_\infty(A, B, V) = 1/[1 + e^{A(V+B)}]$$

A negative value of A implies activation during depolarization, whereas a positive value gives inactivation. The voltage-dependent time 'constants' τ_{mCaS} and τ_{hCaS} determines how quickly the activation or inactivation of ion current occurs. At the depolarized state, it is apparent that τ_{hCaS} will obtain a comparatively large positive value, slowing down the inactivation of the Calcium channels as a result. Hence, h_{CaS} represents the slowest variable acting during the depolarization in the action potential.

$$\tau_{mCaS} = 0.005 + 0.134/[1 + e^{-400(V+0.0487)}]$$

$$\tau_{hCaS} = 0.2 + 5.25/[1 + e^{-250(V+0.043)}]$$

In the model, nine parameters are defined, and have to be given values. These values are mostly obtained from the canonical 14D model, but were modified in the simplified 4D model to maintain a close resemblance to the measured behavior of the natural neuron. In this study, parameters values were selected (Table 1) that showed the output signals of interest, and where multistability was observable.

Table 1. Default values used in when simulating the model unless otherwise specified.

Parameter	Variable	Default value	Unit
Membrane capacitance	C	0.5	nS
Sodium conductance (maximum)	gNa	250	nS
Sodium reversal potential	ENa	0.045	V
Calcium conductance (maximum)	gCaS	80	nS
Calcium reversal potential	ECaS	0.135	V
Leak conductance (maximum)	gleak	15.4	nS
Leak reversal potential	Eleak	-0.0502	V
Sodium half-inactivation potential	Bh	0.031	V
Calcium half-inactivation potential	BhCaS	0.06	V

Simulations and phase portraits

For simulations, a computer software (Fig. 3) was constructed in the Java programming language. The object oriented Java language was chosen since it enabled detailed control over the algorithms, and it has inbuilt classes helpful for creating a user interface. The Java version used was Java SE Development Kit 7 Update 25 (64-bit) and programming environments used were TextPad (https://www.textpad.com/) and JEdit (http://www.jedit.org/). The Java code used for the simulations is presented in Appendix A, B and C.

The main program named HeartNeuron.java contains the algorithms for numerical simulation of the model (Appendix A). As a user interface of the program, a control panel was constructed (Appendix B), where the simulations could be run with the possibility to easily change starting conditions of the state variables and to vary the parameter values. For visualizing the output, a graph plotting function was constructed (Appendix C), which could be modified allowing the output to be visualized in various ways.

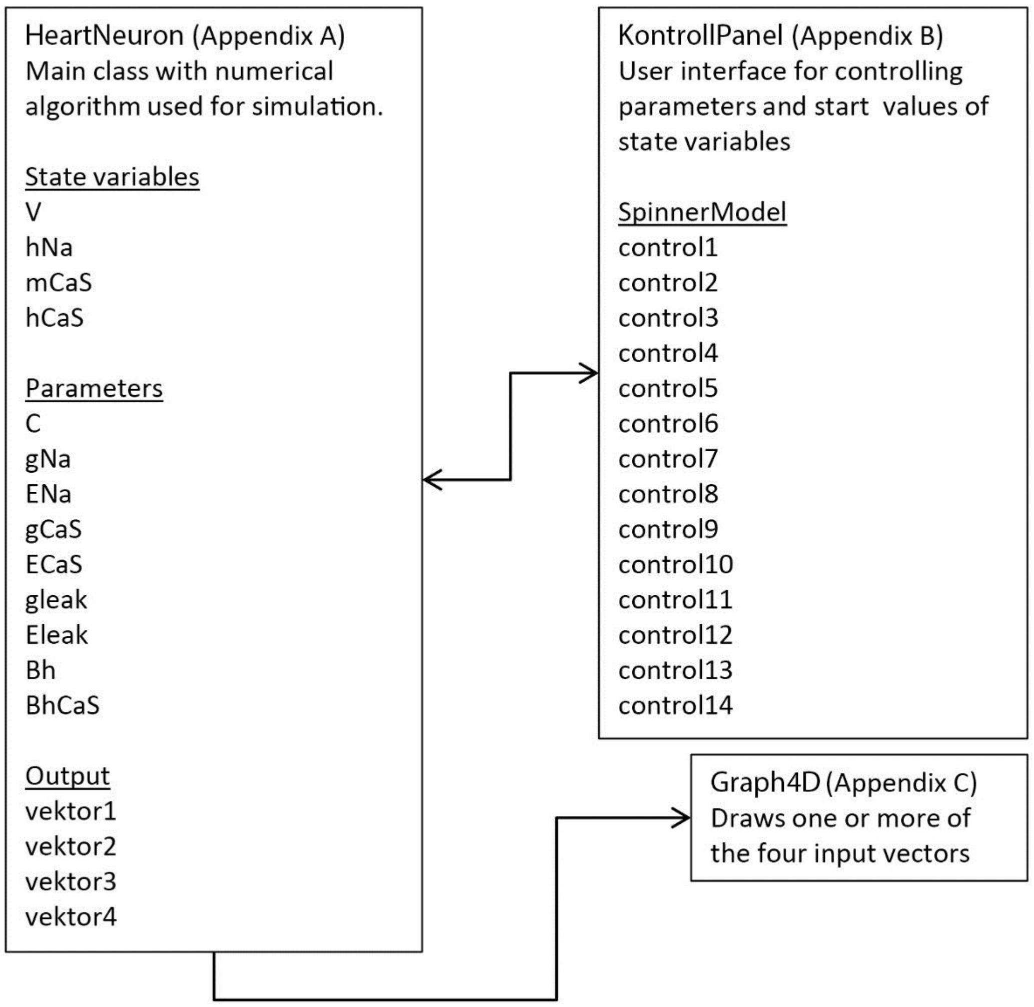

Figure 3. A graphical presentation of the Java program used for simulation, consisting of three classes. The main program, HeartNeuron.java, performs the simulation. The user interface KontrollPanel.java catches active changes of the user in the control panel and sends them to the main program, which then recalculates the simulation. The output from the main program consists of four one-dimensional vectors, visualized in an instance of the Graph4D.java class (Appendix C).

Dynamical systems, such as neuron models, can be visualized by plotting a phase portrait. It gives a geometric representation of the solution trajectories in the dynamical system. The number of state variables determines the number of dimensions required to plot a complete phase portrait of the whole state space, in our case four. However, by only plotting a selected number of state variables, three or less, may still give a useful presentation of the progression of the dynamical system in time. In this study, only two state variables were chosen which allowed the portrait to be visualized on a two dimensional page.

Numerical methods

Initially during the process of developing the software used for simulating the neuron model, the Euler method was chosen for numerically calculating the progress throughout the simulation. The Euler method was chosen because of the few steps in its algorithm, and the intuitive simplicity of its derivation. After running simulations, it became apparent that the Euler method had limitations in reproducing the expected bursting patterns at some of the parameter settings, unless extremely short step size were chosen, which resulted in long simulation time. Therefore, alternative algorithms were searched for and tested. In total, six one-step methods were selected for comparison, with a range of complexity from the explicit Euler method to Runge-Kutta fourth order method. The methods and their implementation were performed as described with derivation in Burden and Faires, 1988.

The selection of methods were chosen based on the similarity in the algorithm construction, and they are all in fact sometimes referred to as Runga-Kutta methods of various orders. The similarity in construction allowed the methods to be readily implemented in the same computer software developed in this study. Except for the semi-implicit Euler method, they are all explicit method, i.e. the successive step in the calculations is expressed in terms of given or previously computed quantities.

The **explicit Euler method** is a simple method performed by stepwise moving forward in time, and calculating the progression of the state variables based on the calculated values of the differential equations in the previous time point. For the scalar, autonomous version of this method, with step size h, the initial value problem

$$\frac{dy}{dt} = f(y), y(0) = y_0$$

can be discretized as

$$y_{n+1} = y_n + hf(y_n)$$

The method generates a local truncation error of order one, O(h). Here the algorithm is presented from the Java code, showing each step in the iterative numerical simulation.

```
Vn = V
V += h*calculateV(Vn, hNa, mCaS, hCaS);
hNa += h*calculatehNa(Vn, hNa, mCaS, hCaS);
mCaS += h*calculatemCaS(Vn, hNa, mCaS, hCaS);
hCaS += h*calculatehCaS(Vn, hNa, mCaS, hCaS);
```

The **semi-implicit Euler method** is highly similar to the Euler method but differs that after the new value of the V is calculated, it is used for the following calculations within the same iteration.

A simplified version of the method with two coupled variables can be expressed as

$$x_{n+1} = x_n + hf(x_n, y_n)$$

$$y_{n+1} = y_n + hg(x_{n+1}, y_n)$$

In practice, when implemented for the neuron model, this means that the algorithm becomes even simpler than the Euler method.

```
V += h*calculateV(V, hNa, mCaS, hCaS);
hNa += h*calculatehNa(V, hNa, mCaS, hCaS);
mCaS += h*calculatemCaS(V, hNa, mCaS, hCaS);
hCaS += h*calculatehCaS(V, hNa, mCaS, hCaS);
```

The **Midpoint method**, by evaluating the functions at two points at every step, a local truncation error of order two, O(h^2).

A scalar, autonomous version can be expressed as

$$y_{n+1} = y_n + hf(y_n + \frac{1}{2}hf(y_n))$$

which was implemented in the code as

```
k1 = calculateV(V, hNa, mCaS, hCaS);
l1 = calculatehNa(V, hNa, mCaS, hCaS);
m1 = calculatemCaS(V, hNa, mCaS, hCaS);
n1 = calculatehCaS(V, hNa, mCaS, hCaS);
k2 = calculateV(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
l2 = calculatehNa(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
m2 = calculatemCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
n2 = calculatehCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);

V += h*k2;
hNa += h*l2;
mCaS += h*m2;
hCaS += h*n2;
```

The **Modified Euler method**, generates a local truncation error of order two.

A scalar, autonomous version can be expressed as

$$y_{n+1} = y_n + \frac{h}{2}[f(y_n) + f\big(y_n + hf(y_n)\big)]$$

which was implemented in the code as

```
k1 = calculateV(V, hNa, mCaS, hCaS);
l1 = calculatehNa(V, hNa, mCaS, hCaS);
m1 = calculatemCaS(V, hNa, mCaS, hCaS);
n1 = calculatehCaS(V, hNa, mCaS, hCaS);
k2 = calculateV(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
l2 = calculatehNa(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
m2 = calculatemCaS(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
n2 = calculatehCaS(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);

V += h*(k1 + k2)/2;
hNa += h*(l1 + l2)/2;
mCaS += h*(m1 + m2)/2;
hCaS += h*(n1 + n2)/2;
```

The **Heun's method**, generates a local truncation error of order two. Both the Modified Euler method and Heun's method are sometimes mentioned as Runge-Kutta methods of order two (Burden and Faires, 1988).

Scalar, autonomous version can be expressed as

$$y_{n+1} = y_n + \frac{h}{4}[f(y_n) + 3f(y_n + \frac{2}{3}hf(y_n))]$$

which in the code was implemented as

```
k1 = calculateV(V, hNa, mCaS, hCaS);
l1 = calculatehNa(V, hNa, mCaS, hCaS);
m1 = calculatemCaS(V, hNa, mCaS, hCaS);
n1 = calculatehCaS(V, hNa, mCaS, hCaS);
k2 = calculateV(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3, hCaS + h*n1*2/3);
l2 = calculatehNa(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3, hCaS + h*n1*2/3);
m2 = calculatemCaS(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3, hCaS + h*n1*2/3);
n2 = calculatehCaS(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3, hCaS + h*n1*2/3);

V += h*(k1 + 3*k2)/4;
hNa += h*(l1 + 3*l2)/4;
mCaS += h*(m1 + 3*m2)/4;
hCaS += h*(n1 + 3*n2)/4;
```

The **Runge-Kutta fourth order method** requires some more steps in the algorithm compared to Euler method, but generally creates more precise estimations of the progress in the state variables.

The scalar, autonomous version of this method can be expressed as

$$y_{n+1} = y_n + \frac{h}{6}[f(y_n) + 2f(y_n + \frac{1}{2}hf(y_n)) + 2f(y_n + \frac{1}{2}hf(y_n + \frac{1}{2}f(y_n))) + f(y_n + hf(y_n + \frac{1}{2}hf(y_n + \frac{1}{2}f(y_n)))]$$

The resulting local truncation error is of order four, O(h^4). Below is an extraction from the Java program showing the part encoding the algorithm.

```
k1 = calculateV(V, hNa, mCaS, hCaS);
l1 = calculatehNa(V, hNa, mCaS, hCaS);
m1 = calculatemCaS(V, hNa, mCaS, hCaS);
n1 = calculatehCaS(V, hNa, mCaS, hCaS);
k2 = calculateV(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
l2 = calculatehNa(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
m2 = calculatemCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
n2 = calculatehCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
k3 = calculateV(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS + h*n2/2);
l3 = calculatehNa(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS + h*n2/2);
m3 = calculatemCaS(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS + h*n2/2);
```

```
n3 = calculatehCaS(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS + h*n2/2);
k4 = calculateV(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);
l4 = calculatehNa(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);
m4 = calculatemCaS(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);
n4 = calculatehCaS(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);

V += h*(k1 + 2*k2 + 2*k3 + k4)/6;
hNa += h*(l1 + 2*l2 + 2*l3 + l4)/6;
mCaS += h*(m1 + 2*m2 + 2*m3 + m4)/6;
hCaS += h*(n1 + 2*n2 + 2*n3 + n4)/6;
```

The Java program was constructed in a way that it is easy to modify the algorithms. There is therefore a possibility to further investigate different algorithms and evaluate with respect to the outcome from the neuron model. All decimal variables were declared as the data type double, which is of the format double-precision 64-bit IEEE 754 floating point, with a precision of approximately 16 digits.

It was apparent that the step size had influence on the result of the simulations. The bursting patterns and multistability properties varied depending on which step size was chosen. Therefore, different step sizes were tested, and the outcome form the simulations evaluated visually with respect to bursting frequency and overall appearance of the output graph.

Temporal measurements

In order to evaluate the outcome form the simulations, a strategy had to be developed for obtaining a quantitative measure of the firing pattern, which can be seen as a qualitative matter. Three temporal quantities were chosen which characterizes the firing patterns: spiking frequency (i.e. the spike firing rate), burst period (i.e. the time between the initiation of two subsequent spike trains) and interburst interval duration (i.e. the time from the last spike in a spike train till the initiation of the following spike train).

The third burst was chosen for the measurements in order to allow the dynamical system to stabilize during the simulations, avoiding the irregularities sometimes apparent in the first burst of the simulations. By counting the number of pixels between the top of the first spike and the last spike in the selected burst, a measurement of the duration of a burst (a spike train) was obtained. In a similar manner, by counting the number of pixels between the first spike of two succeeding bursts, the period of the whole burst was obtained. Spike frequency was estimated by counting the number of spikes in a burst of the simulation and divide by burst duration. Interburst interval was estimated by subtracting the burst duration from the whole burst period.

RESULTS AND DISCUSSION

Algorithm selection and optimization

An optimal numerical algorithm for the simulations would accurately reproduce the pattern of the bursts with the minimal number of calculations required. The requirement of accuracy and stability in the simulations is obvious, and since the time factor is of importance an exaggerated number of calculation steps would result in long waiting times during simulations.

Six numerical methods were chosen for evaluation: Euler’s explicit method, semi-implicit Euler method, Midpoint method, Modified Euler, Heun’s method and Runge-Kutta’s fourth order method (Table 2). The algorithms were tested for different step sizes. As expected, the algorithms generally generate more exact simulations if shorter step sizes are chosen. However, this must be balanced with the longer time required to perform the calculations, and the unavoidable round-off error introduced when the step size is too minute and the decimal averaging in the in the arithmetic calculations performed with finite number of digits on a computer starts to influence. It was apparent during the simulations that the time factor was of greater importance than the rounding-off error, since a test run using the minuscule step size of 1 ns (10^{-9} s) did not generate any visual errors in the bursting pattern, although the test run required more than one hour to complete when using the time-efficient Euler’s method.

Table 2. Comparison of six different numerical methods for simulating the nerve cell model. The properties of the bursting pattern measured were spiking frequency (SF), burst period (BP) and interburst interval (IBI). The shaded regions show where the values were stabilized as the step size was decreased.

Step size	**explicit Euler’s method**			**semi-implicit Euler method**			**Midpoint method**		
	SF	BP	IBI	SF	BP	IBI	SF	BP	IBI
10	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
5	4.88	6.47	3.6	5.59	10.1	3.3	5.42	8.27	3.47
3.33	5	7.13	3.53	5.56	9.27	3.34	5.4	8.47	3.47
1	5.37	8	3.53	5.52	8.53	3.46	5.4	8.47	3.47
0.1	5.57	8.27	3.6	5.48	8.4	3.47	5.4	8.47	3.47
	Modified Euler			**Heun's method**			**Runge-Kutta's fourth order**		
	SF	BP	IBI	SF	BP	IBI	SF	BP	IBI
10	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d	n.d.	n.d.
5	5.4	8.47	3.47	5.42	8.27	3.47	5.4	8.47	3.47
3.33	5.4	8.47	3.47	5.4	8.47	3.47	5.4	8.47	3.47
1	5.4	8.47	3.47	5.4	8.47	3.47	5.4	8.47	3.47
0.1	5.4	8.47	3.47	5.4	8.47	3.47	5.4	8.47	3.47

For the explicit Euler method, a starting step size of 10 ms (Fig. 4a) generated semi-random artefacts and apparent numerical errors in the calculations. This was obvious since irregular spikes were generated with enlarged amplitude, i.e. with a membrane potential range exceeding what was later seen at smaller step sizes generating stable bursting patterns. Already at a step size of 5 ms (Fig. 4b) a regular bursting pattern was produced, indicating that the numerical calculations of Euler method at this step size was close enough to allow the model to exhibit the behavior similar to what is seen in the natural neuron. Decreasing the step size further resulted in a decrease in the number of bursts during the simulation period, ending at 12 bursts during 100 s.

The most pronounced irregularities seen in the Euler's method with the large step size of 10 ms occurred in connection to the first spike in the bursts. This first spike in the bursts is of greater magnitude than the other. This is mainly driven by the rapid influx of sodium, representing the most rapid currents during the simulations (Fig. 2b). It therefor seems that the stability of the Euler method is less when the system is in a rapid change. This stability issue could be put into the context of looking at the magnitudes of the eigenvalues of the linearized system at the states when the spike is generated. This will be further discussed below.

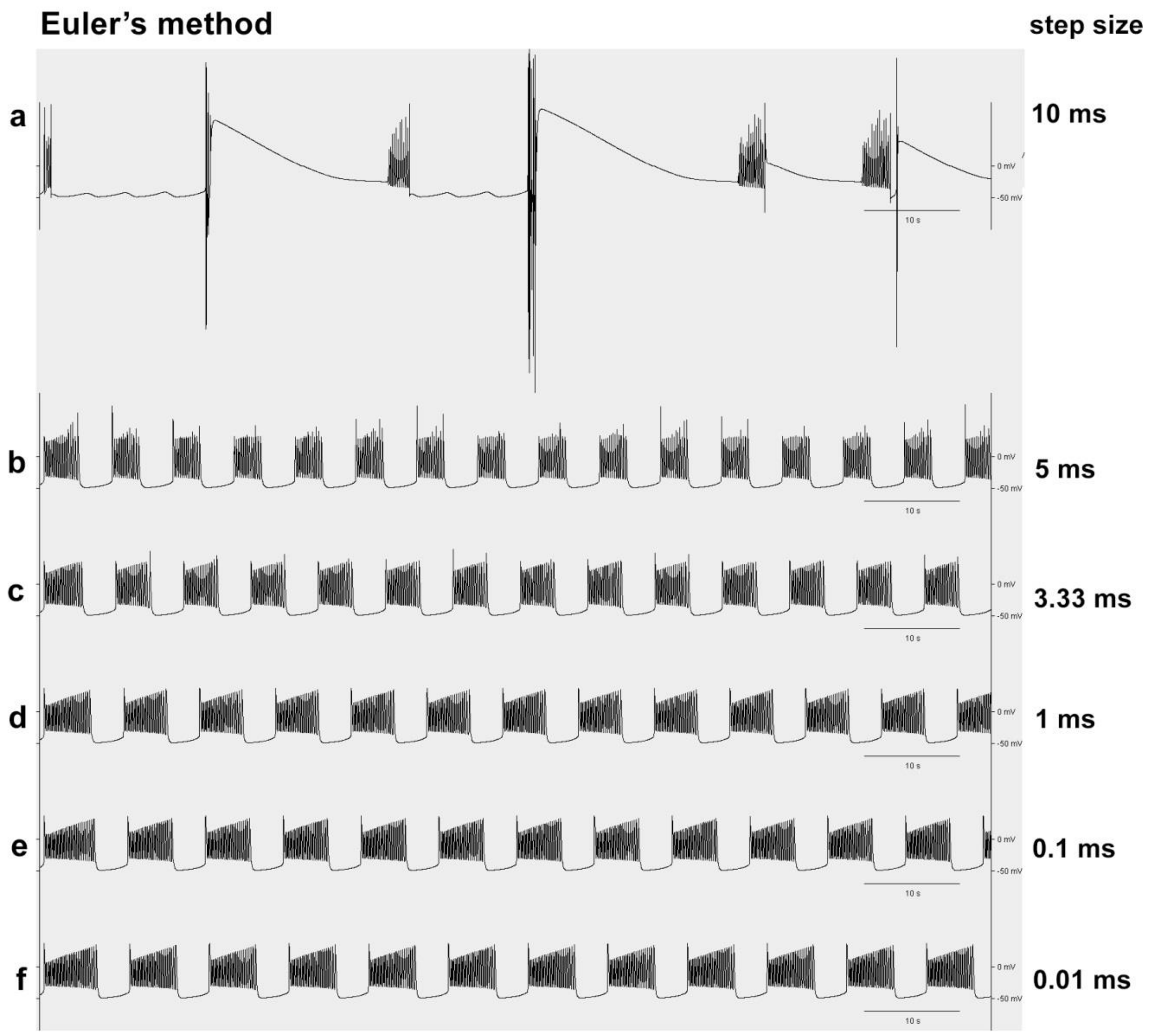

Figure 4. Using explicit Euler's method and comparing the effect of different step size on the bursting patterns, i.e. the progression of membrane potential in the neuronal model. The step size was varied from (a) 10 ms down to (f) 0.01 ms per step.

The semi-implicit Euler method implicated a minor modification in the algorithm compared to the explicit Euler method. The semi-implicit Euler method appeared more stable than the explicit (Fig. 5), indicating that this method might be more useful in dynamical systems such as the periodic neuron model.

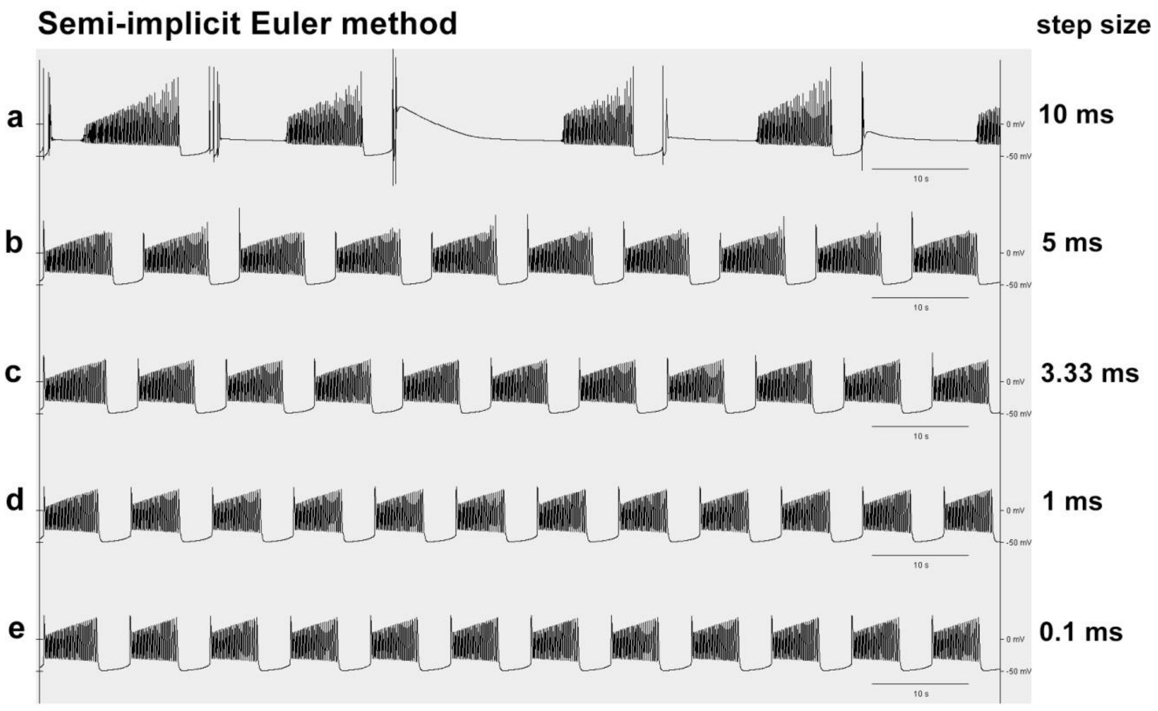

Figure 5. Using semi-implicit Euler's method and comparing the effect of different step size on the bursting patterns, i.e. the progression of membrane potential in the neuronal model. The step size was varied from (a) 10 ms down to (e) 0.1 ms per step.

For the Runge-Kutta's fourth order method, the starting step size of 10 ms generated more stable pattern compared to the Euler method, and the range of the membrane potential was more confined. However, there were some abnormalities in the bursting affected by numerical errors from the too large step size (Fig. 6a). The convergence of the pattern was quicker than for the Euler method, and seemed stable already at 5 ms. Decreasing the step size further did not visibly improve the bursting pattern, indicating that the method rapidly converged with the decreased step size.

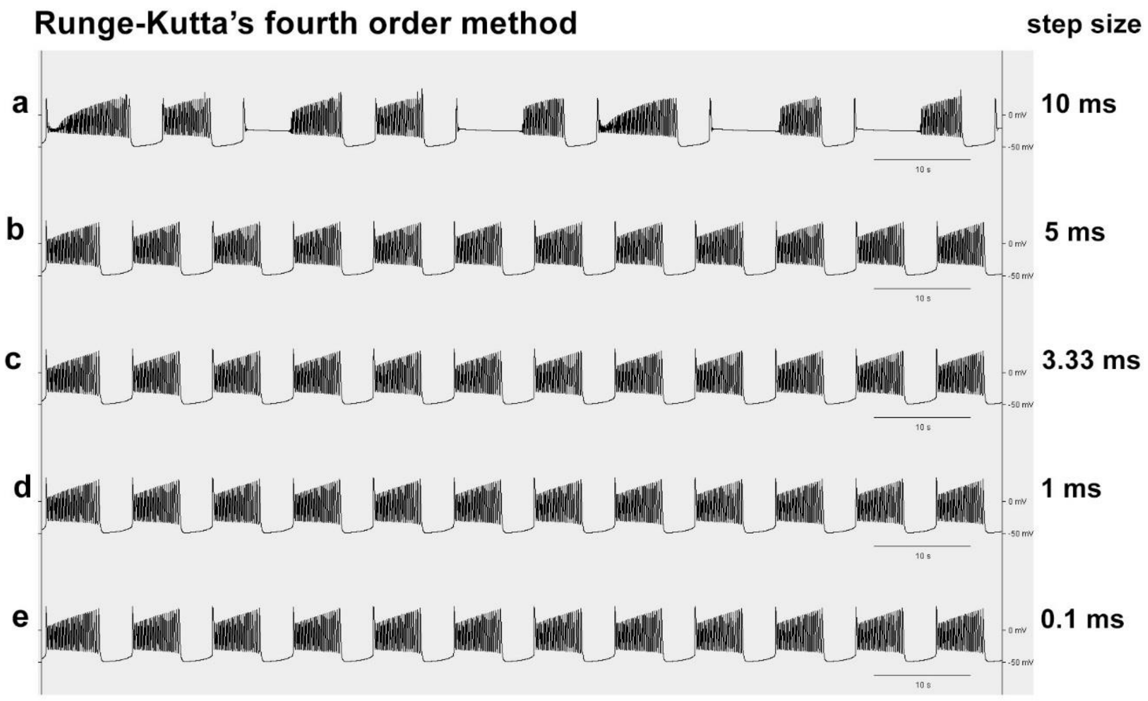

Figure 6. Using Runge-Kutta's fourth order method and comparing the effect of different step size on the bursting patterns. The step size was varied from (a) 10 ms down to (e) 0.1 ms per step.

In the comparison it was apparent that the explicit Euler's method, although having the advantage of being a simple algorithm, did not give satisfactory results unless very short step size was chosen, < 0.1 ms (Table 2). The semi-implicit Euler method was more stable than the explicit method, and was in addition slightly simpler in the algorithm, and therefore seems as a better choice for the periodic neuron model simulations. The second order methods, i.e. Midpoint method, Modified Euler and Heun's method did generate good results, and in particular the Modified Euler. As expected, the Runge-Kutta's fourth order method did generate the best results, although the number of steps in the algorithm did make the method comparably slow.

Due to the convincingly quick convergence of the Runge-Kutta fourth order method, it was chosen for the subsequent analyses, and with the step size set to 0.1 ms. With these settings a simulation required approximately 2 seconds to be completed on a standard office computer used in this study (Processor: Intel(R) Core ™ i5-4670 CPU 3.40 GHz; System: Windows 7 Professional 64-bit.)

Stability of Euler's explicit method

It was apparent from the simulations that if the step size was not short enough, the pattern was disturbed for all numerical methods (Table 2, Fig. 4, 5, 6). The explanation for this lies in an intrinsic instability of these methods, which interplays both with the step size and the eigenvalues of the linearized system at each state.

To understand the mathematical background behind the instability of the Euler's method, we can start by looking at a simple model consisting of a scalar ordinary differential equation (ODE).

As a test problem, consider the initial value problem with a scalar linear ODE

$$\frac{dy}{dt} = \lambda y(t), \text{ with } y(0) = y_0$$

where $\lambda \in \mathbb{C}$ is a parameter equivalent to the eigenvalue of a linear system.

The exact solution to the equation is

$$y(t) = y_0 e^{\lambda t}$$

The equation is stable in the sense of Lyapunov (or not growing in magnitude) if real part of the eigenvalue $\mathrm{Re}(\lambda) \leq 0$, and asymptotically stable if $\mathrm{Re}(\lambda) < 0$. In the latter case the solution is exponentially decaying, i.e. $\lim_{t\to\infty} y(t) = 0$. The imaginary part $\mathrm{Im}(\lambda)$ gives an oscillation of the solution curve, which decrease in amplitude with time if the solution is asymptotically stable, and increase in amplitude if the solution is unstable.

The numerical methods, such as Euler's method, involve discretization of the initial value problem into a difference equation. The step size, denoted h, is normally a small real number which is always positive when time progress forward.

The explicit Euler's method can be generally expressed as

$$y_{n+1} = y_n + hf(t_n, y_n)$$

When subjecting the scalar linear ODE to discretization using the Euler's explicit method we obtain

$$y_{n+1} = y_n + h\lambda y_n$$

By regrouping the terms and emanating from the initial value we get

$$y_{n+1} = (1 + h\lambda)y_n = (1 + h\lambda)^{n+1} y_0$$

Hence, the Euler's method is stable if

$$|1 + h\lambda| \leq 1$$

The method is apparently convergent in a disk in the complex plane centered at $h\lambda = -1$ and with the radius 1. It is also apparent that the convergence is dependent on h, and as long as $\mathrm{Re}(\lambda) < 0$ convergence is obtained if the step size is sufficiently small. In other words, for $\mathrm{Re}(\lambda) < 0$ the method is conditionally stable (stable for sufficiently small h), but for $\mathrm{Re}(\lambda) > 0$ the method is unconditionally unstable.

With the same reasoning, we can generalize to a linear system of ODEs. This can be expressed as

$$\frac{dy}{dt} = Ay$$

where A is a $d \times d$ matrix which we suppose has a basis of eigenvectors. It can be shown that the general solution can be written in the compact form

$$y(t) = \sum_{i=1}^{d} C_i e^{\lambda_i t} u_i$$

where $\lambda_1, \ldots, \lambda_d$ are the eignevalues, $u_1, \ldots, u_d$ are the corresponding eigenvectors, and $C_1, \ldots, C_d$ are coefficients.

Since the eigenvalues are in the exponents of the general solution, it means that the stability is determined by the sign of the real parts of the eigenvalues. If all eigenvalues lie in the closed left half-plane of the complex plane, i.e. the real part $\mathrm{Re}(\lambda_i) \leq 0$, then the origin is stable in the sense of Lyapunov. Also, if all eigenvalues have a negative real part, i.e. the real part $\mathrm{Re}(\lambda_i) < 0$, then the origin is asymptotically stable. On the other hand, if for any eigenvalues the real part $\mathrm{Re}(\lambda_i) > 0$, then the solution is unstable.

If we apply discretization of the linear system of ODEs using Euler's explicit method we get

$$y_{n+1} = y_n + hAy_n = (I + hA)y_n.$$

If we let y_n be expressed in the eigenbasis $(u_1, \ldots, u_d)$, we may write

$$y_n = \alpha_1^n u_1 + \alpha_2^n u_2 + \cdots + \alpha_d^n u_d.$$

If we now apply Euler's explicit method, we find

$$\begin{aligned} y_{n+1} &= (I + hA)(\alpha_1^n u_1 + \alpha_2^n u_2 + \cdots + \alpha_d^n u_d) \\ &= \alpha_1^n (I + hA)u_1 + \alpha_2^n (I + hA)u_2 + \cdots + \alpha_d^n (I + hA)u_d \\ &= \alpha_1^n (u_1 + hAu_1) + \alpha_2^n (u_2 + hAu_2) + \cdots + \alpha_d^n (u_d + \mathrm{hA}u_d) \\ &= \alpha_1^n (u_1 + h\lambda_1 u_1) + \alpha_2^n (u_2 + h\lambda_2 u_2) + \cdots + \alpha_d^n (u_d + \mathrm{h}\lambda_d u_d) \\ &= \alpha_1^n (I + h\lambda_1)u_1 + \alpha_2^n (I + h\lambda_2)u_2 + \cdots + \alpha_d^n (I + h\lambda_3)u_d = \sum_{1=1}^{d} \alpha_i^n (1 + h\lambda_i)u_i \end{aligned}$$

Since $u_1, \ldots, u_d$ are eigenvectors. On the other hand we can also write

$$y_{n+1} = \sum_{i=1}^{d} \alpha_i^{n+1} u_i$$

and comparing these last two equations and using the uniqueness of representation, we have

$$\alpha_i^{n+1} = (1 + h\lambda_i)\alpha_i^n.$$

It follows from this that the origin is a stable fixed point if $|1+h\lambda_i| \leq 1$, i = 1, 2, …, d, and is asymptotically stable if $|1 + h\lambda_i| < 1$, i = 1, 2, …, d. This condition has to be fulfilled by all eigenvalues in order for the system to be stable. As for the scalar ODE, the region of absolute stability of explicit Euler's method lies within a disk centered at $h\lambda = -1$ with a radius = 1.

For non-linear systems, such as the leech heart interneuron, it is possible to linearize the system at each state. Then a linearized system is obtained of the same form as above:

$$\frac{dy}{dt} = Ay$$

This implicates that each state may have different eigenvalues, and the eigenvalues will change as the state is changing with time.

For the linearized system, if it is possible to estimate the maximum values of the eigenvalues, the stability can be assessed. If |1 + hλ| at any state > 1, the Euler method is no longer stable at this particular state, and a disturbed pattern will be observed. However, as the state change with time also the eigenvalues change. During the simulation, using Euler's method, if |1 + hλ| returns to < 1, also the system returns to a stable course.

Stability regions of Runge-Kutta methods

In a more general sense, the stability of a numerical method is decided by its stability function. As seen above, Euler's explicit method (which actually is a Runge-Kutta method of order one) has the stability function

$$R(h\lambda) = 1 + h\lambda$$

since if we let μ = hλ, the Euler's method is stable when

$$|R(\mu)| = |1 + \mu| < 1.$$

The modified Euler's function is actually a Runge-Kutta second order method and its stability function can be obtained. Starting from the scalar discretized formulation

$$y_{n+1} = y_n + \tfrac{h}{2}[f(y_n) + f\big(y_n + hf(y_n)\big)].$$

By using the test problem

$$\frac{dy}{dt} = \lambda y(t)$$

we get

$$y_{n+1} = y_n + \frac{h}{2}[\lambda y_n + \lambda(y_n + h\lambda y_n)] = y_n + h\lambda y_n + \frac{h^2\lambda^2}{2} y_n$$

and we obtain the stability function

$$R(\mu) = 1 + \mu + \frac{1}{2}\mu^2$$

Similarly, the stability function for Runge-Kutta fourth order method is found to be

$$R(\mu) = 1 + \mu + \frac{1}{2}\mu^2 + \frac{1}{6}\mu^3 + \frac{1}{24}\mu^4$$

This is agreeing with the Taylor series expansion of e^{μ} which is expected (Frank 2008).

In the region where $|R(\mu)| < 1$, the numerical method can be considered as stable. The appearance of this stability region can be visualized in plots for the Runge-Kutta methods of different orders (Fig. 7).

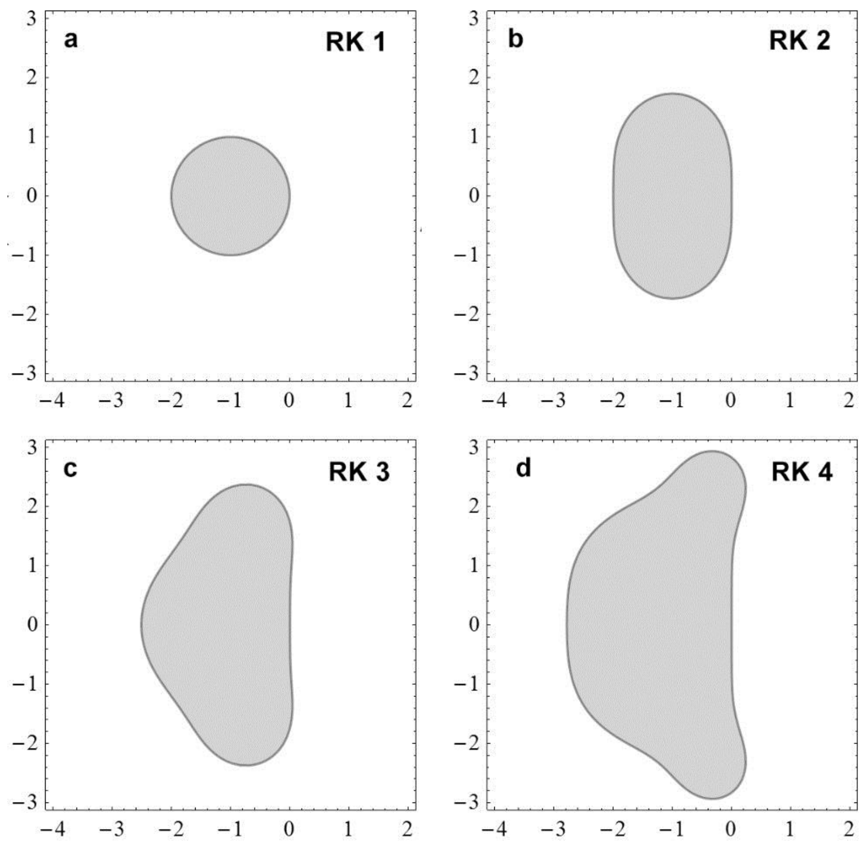

Figure 7. The stability regions of Runge-Kutta methods, a. first order (Euler's method) which is a circle centered at μ = -1, b. second order, c. third order, and d. fourth order. The horizontal axis represents the real part, and the vertical axis represents the imaginary part of μ.

Explicit vs semi-implicit Euler methods

Although the algorithms of the explicit and the semi-implicit Euler methods appear similar, they differ in the efficiency during the simulations of the neuron model. As seen in table 2 and Figure 4, 5, the semi-implicit method converges better, with the decreasing step size, to the stable values. Here follows a mathematical analysis and stability comparison of the two methods.

Since the model exhibits periodicity, it is natural to compare the methods in a simple periodic system, such as the harmonic oscillator.

$$\frac{dx}{dt} = u$$

$$\frac{du}{dt} = -\omega^2 x$$

This two-dimensional dynamic system can be discretized using the explicit Euler method

$$x_{n+1} = x_n + hu_n$$

$$u_{n+1} = u_n - h\omega^2 x_n$$

This results in the propagator

$$\Psi_h\left(\begin{pmatrix}x\\u\end{pmatrix}\right)=\begin{pmatrix}1 & h\\-h\omega^2 & 1\end{pmatrix}\begin{pmatrix}x\\u\end{pmatrix}=:M(\theta)\begin{pmatrix}x\\u\end{pmatrix},\theta=h\omega$$

The matrix M(θ) has eigenvalues

$$\lambda_{\pm}=1\pm\sqrt{-h^2\omega^2}=1\pm\sqrt{-\theta^2}$$

This means that for all h ≠ 0, the method will diverge from the cycle of the orbit of the oscillator, since $|\lambda_+| = |\lambda_-| > 1$. Since det M(θ) > 1, the simulated oscillator will generate an orbit with increasing diameter instead of the circular stable orbit which should be the result from this dynamical model of an harmonic oscillator.

Discretizing the harmonic oscillator using semi-implicit Euler method gives

$$x_{n+1}=x_n+hu_n$$

$$u_{n+1}=u_n-h\omega^2x_{n+1}$$

which gives us

$$x_{n+1}=x_n+hu_n$$

$$u_{n+1}=(1-h^2\omega^2)u_n+(-h\omega^2)x_n$$

This results in the propagator

$$\Psi_h\left(\begin{pmatrix}x\\u\end{pmatrix}\right)=\begin{pmatrix}1 & h\\-h\omega^2 & 1-h^2\omega^2\end{pmatrix}\begin{pmatrix}x\\u\end{pmatrix}=:M(\theta)\begin{pmatrix}x\\u\end{pmatrix},\theta=h\omega$$

The matrix M(θ) has eigenvalues

$$\lambda_{\pm}=1-\frac{1}{2}h^2\omega^2\pm\sqrt{h^2\omega^2\left(\frac{1}{4}h^2\omega^2-1\right)}=1-\frac{1}{2}\theta^2\pm\sqrt{\theta^2(\frac{1}{4}\theta^2-1)}$$

Since

$$\det M(\theta)=1-h^2\omega^2+h^2\omega^2=1$$

We know that $|\lambda_{\pm}|^2 = 1$. From this we can conclude that semi-implicit Euler method will result in a conserved system which preserves the magnitude of the oscillation, which was not the case for the explicit Euler method.

For h = 0, we have the two eigenvalues $\lambda_- = \lambda_+ = 1$. This reflects the situation when there is no step size, and the system will stay at the initial values. Therefore h = 0 is not useful in practice. As h is increased, the eigenvalues will get complementary positive and negative imaginary parts, and will move along the unit circle in the complex plane. At $h=\sqrt{2}/\omega$h, both eigenvalues are purely imaginary with the values ±i. When further increasing h to h = 2/ω, both eigenvalues meet at the real negative value -1. For θ < 4, we get real negative values for both eigenvalues. The lower eigenvalue λ_-

will have an increasing modulus with an increasing h, whereas the upper eigenvalue λ_+ will tend to zero along the negative real axis.

From these results it is apparent that the semi-implicit Euler method exhibits more stable properties than the explicit Euler in a highly cyclic system such as the harmonic oscillator. These stability properties may explain why also in the periodic neuron model, the semi-implicit Euler method generated more accurate simulations.

Exploring parameter space

In the neuron model, there are nine defined parameters, together forming a nine-dimensional parameter space. The approach for the exploration was to initiate the exploration from a set of select default values, and then vary the parameters one by one in order to achieve over-viewable results. For default values (Table 1) of the nine parameters used in the model, a point in the parameter space was chosen where a rich signal output of the model was apparent, with silence, subthershold oscillations and bursting (see below under multistability section). From these default settings, the user interface of the Java program facilitated the variation of the parameters and allowed the signal output, with different firing patterns, to be observed.

From previous studies (Cymbalyuk et al. 2002, Malashchenko et al. 2011) it was apparent that the leak currents were of a particular significance, and small deviations in this parameter generated an altered signal output both in the neuron biological preparations and in the mathematical model. Therefor this parameter was firstly chosen to be examined in this study. However, the other parameters were also explored with altered firing patterns as a result, but this data will not be presented further.

When moving the gleak parameter far from the default value, in the range 4 < gleak < 5, the signal output went from silence to spiking (Fig. 8). As the gleak parameter was further increased, a concomitant increase in amplitude of the action potentials was observed.

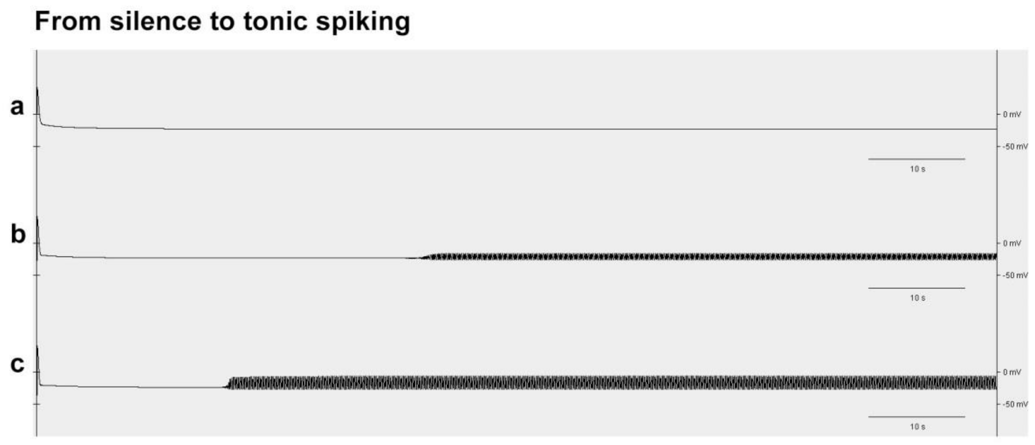

Figure 8. The occurrence of tonic spiking when varying the gleak parameter: (a) silence at gleak = 4, (b) low amplitude tonic spiking at gleak = 5, (c) increased amplitude of tonic spiking at gleak = 6.

When continuing increasing the gleak values, the amplitudes of the action potentials gradually increased. At a distinct point, bursting started to appear at what appeared to be a bifurcation point between the values gleak = 12.348 and gleak = 12.349 (Fig. 9). The bursts occurring after the bifurcation point, were relatively long.

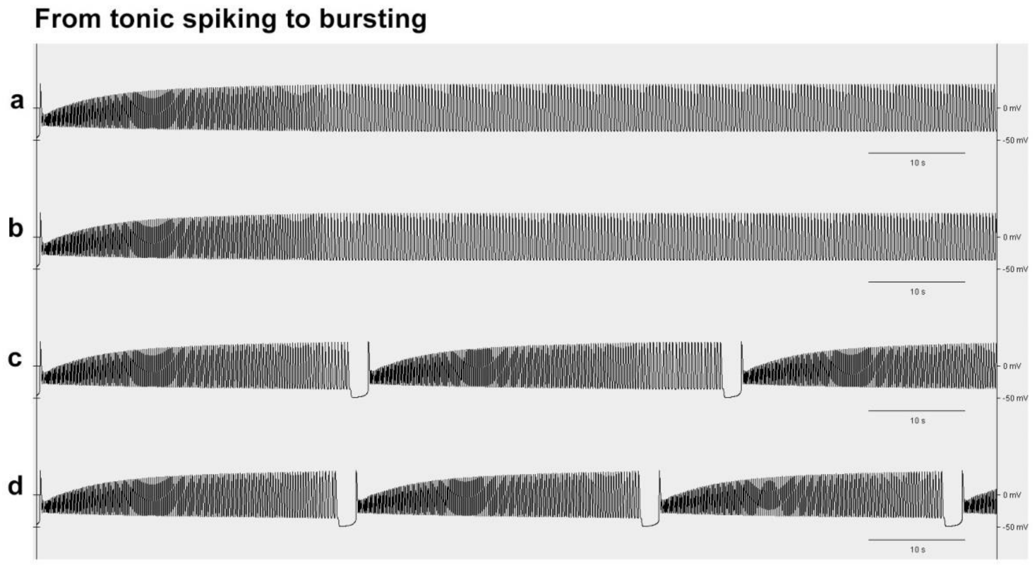

Figure 9. The transition from spiking to bursting occurred between gleak = 12.348 and 12.349. The gleak values in the figure were (a) 12.347, (b) 12.348, (c) 12.349 and (d) 12.350.

The transition from silence to tonic spiking, and the transition from tonic spiking to bursting, were visualized in phase portraits (Fig. 10). The same Java program was used, but the class Graph4D was replaced with a phase portrait plotting class (Appendix C2). The two variables selected for visualization were membrane potential (V) and the activation variable of the calcium current (mCaS). The phase portraits show when the system goes to silence (Fig. 10a), a small oscillation (Fig. 10b), greater oscillation (Fig. 10c) and finally an oscillation with a longer detour, i.e. bursting (Fig. 10d).

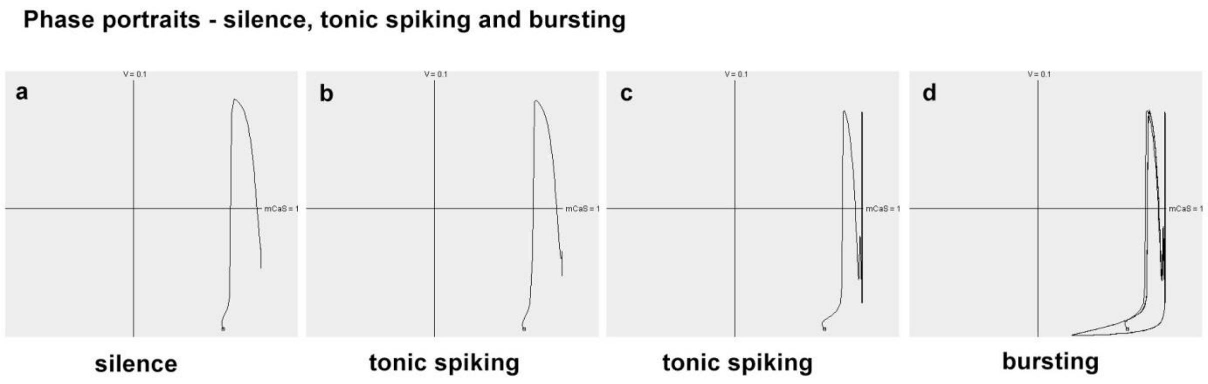

Figure 10. Phase portraits showing the transition from silence (a) to tonic spiking (b), and from tonic spiking (c) to bursting (d). The gleak values were (a) 4, (b) 5, (c) 12.348 and (d) 12.349. The small square in each graph represents the starting point for the simulation. The initial value for mCaS were set to 0.7.

When gleak was increased to a certain point, bursting disappeared (Fig. 11). The disappearance bursting followed at a distinct point, after which silence occurred after a few initial bursts (Fig. 11b).

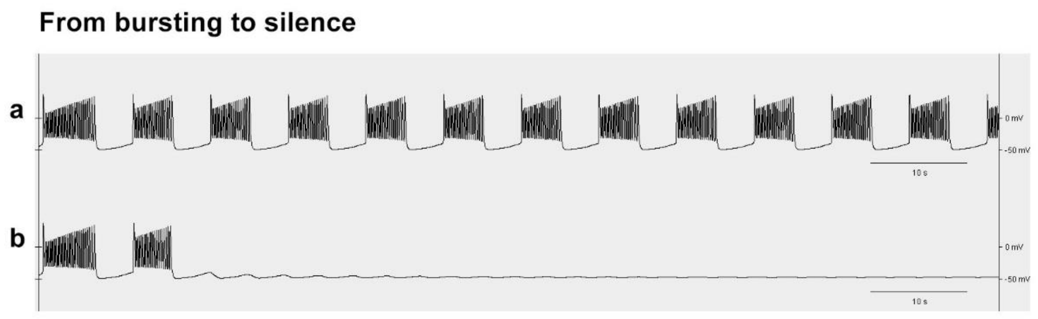

Figure 11. The bursting goes to silence as the gleak parameter changes between (a) 15.481 and (b) 15.482.

For further characterization, spiking frequency, burst period and interburst interval were estimated as a selected set of parameters were varied. Three parameters were chosen, representing the maximum conductance through the cell membrane of the leak (gleak), sodium (gNa) and calcium ions (gCaS). Interestingly, the bursting pattern responded differently when varying these parameters.

When gleak was varied within the parameter range producing bursts, 12.349 < gleak < 15.481, there was an increased spiking frequency in the middle part of the investigated interval. The overall change in spiking frequency was moderate, and the frequency stayed within the range 5.1 – 5.6 (Fig. 12a). The burst period, however, showed a more dramatic change when the gleak parameter was varied, and decreased fourth fold as the gleak parameter was varied within the bursting range (Fig. 12b). In contrast, the interburst interval increased slightly with increased gleak (Fig. 12c).

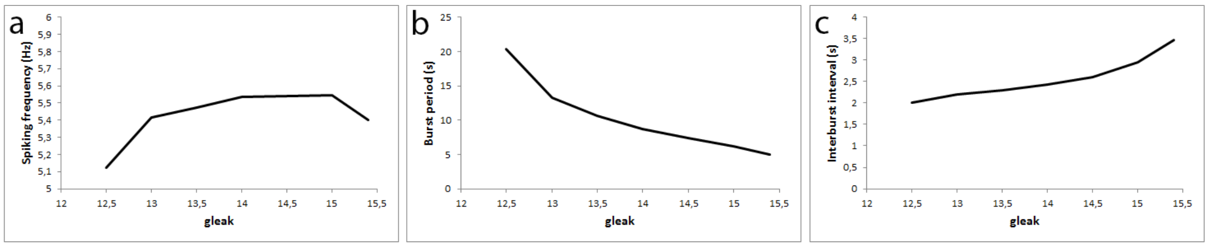

Figure 12. Bursting pattern depending on the gleak parameter. (a) spiking frequency was relatively stable, although a slight increase was seen in the middle part of the range, (b) burst period decreased, while (c) interburst interval increased slightly.

The gNa is a parameter representing the maximum sodium current. Since this current is the fastest in the model, and represents a strong inward current, it was of particular interest. Hypothetically it could influence the spiking frequency due to a faster inflow of ions in the beginning of spikes, however this was not apparent from the obtained results (Fig. 13a). Instead, the spiking frequency was moderately upshifted in the middle part of the parameter region investigated, and decreased in as the parameter was further increased. For the burst period and interburst interval, opposite pattern compared to gleak was observed (Fig. 13b-c).

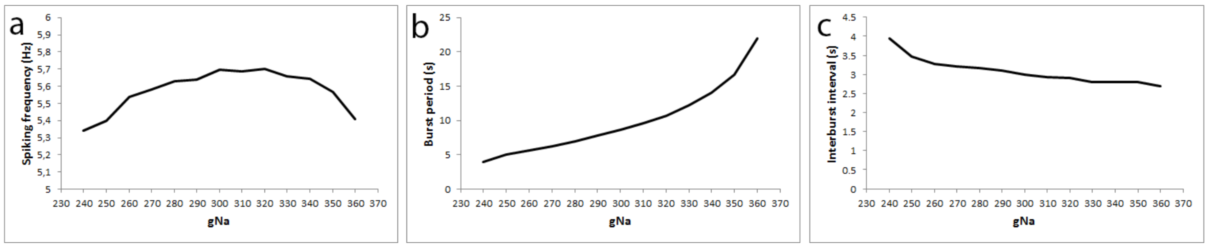

Figure 13. Bursting pattern depending on the gNa parameter. (a) spiking frequency was relatively stable, although a slight increase was seen in the middle part of the range, (b) burst period increased, (c) interburst interval decreased slightly.

The gCaS parameter represents a slow calcium current, with an inward direction which is similar to the sodium current. Also the effect on the spiking pattern (Fig. 14a-c) was similar to that of gNa. This could be explained by the two ions having the same inward direction, although acting in different time scales.

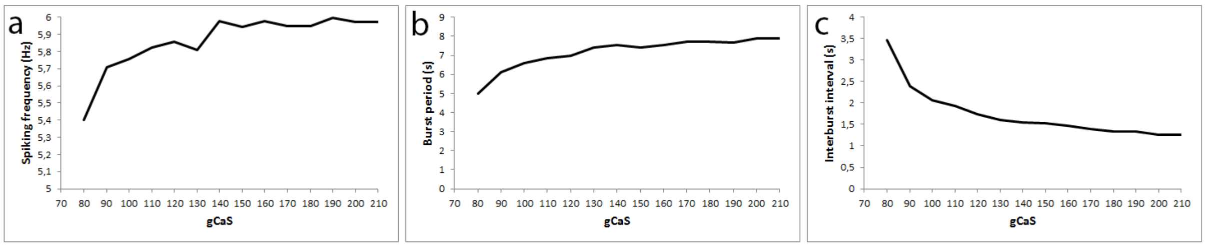

Figure 14. Bursting pattern depending on the gCaS parameter. (a) spiking frequency was relatively stable, although increased with increased gCaS, (b) burst period increased slightly, and (c) interburst interval decreased. Interestingly, the bursting did not disappear as the parameter was increased, even up to gCaS = 1000 the bursting persisted (data not shown).

To get a more complete picture, the other parameter was varied and the effect on the spiking frequency was observed (data not shown), although the frequency was never observed outside the range 5 – 6 Hz. This indicates that other parameters, not defined in the model is the main regulators of spiking frequency.

Changing the Eleak rapidly disrupted the bursting, if changed ever so little from the default value, in both cases if increased or increased (data not shown).

Multistability

In more complex dynamical systems, which are multi-dimensional and non-linear, more than one stable equilibrium may coexist in the state space. In other words, the given dynamical system may have two or more stable behaviors existing in the phase plane, i.e. there may be bistability or multistability.

A prerequisite for multistability, is that the attracting regimes are separated by a repelling regime. The domain of attraction is the set of initial states in the state space that asymptotically lead to the equilibrium. The separating regime can for example be a saddle periodic orbit or a saddle equilibrium.

An example of a simple dynamical system with bistability.

$$\dot{x} = \mu x - x^3$$

$$\dot{y} = -y$$

As μ is varied from a negative value (Fig. 15a) to a positive value (Fig. 15b), two stable equilibria occurs separated by the stable manifold of the saddle-equilibrium in (0, 0) represents the separatrix, separating the two domains of attraction.

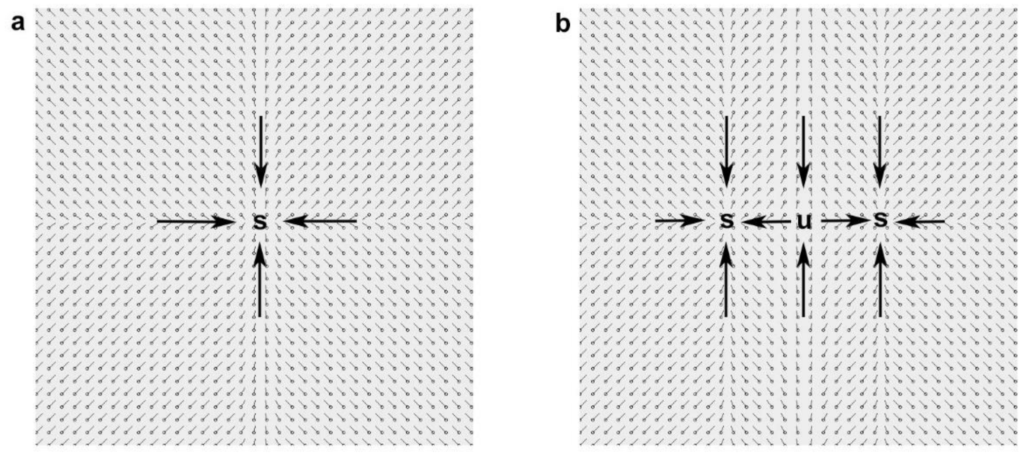

Figure 15. An example of bistability. As the bifurcation parameter μ is changed, a stable equilibrium (s) at (0, 0) turns to an unstable saddle node (u) separating two stable equilibria.

In the leech interneuron model, bistability and even tristability was observed. For the tristability, the regimes observed were: silence, subthreshold oscillations, tonic spiking and bursting (Fig. 16).

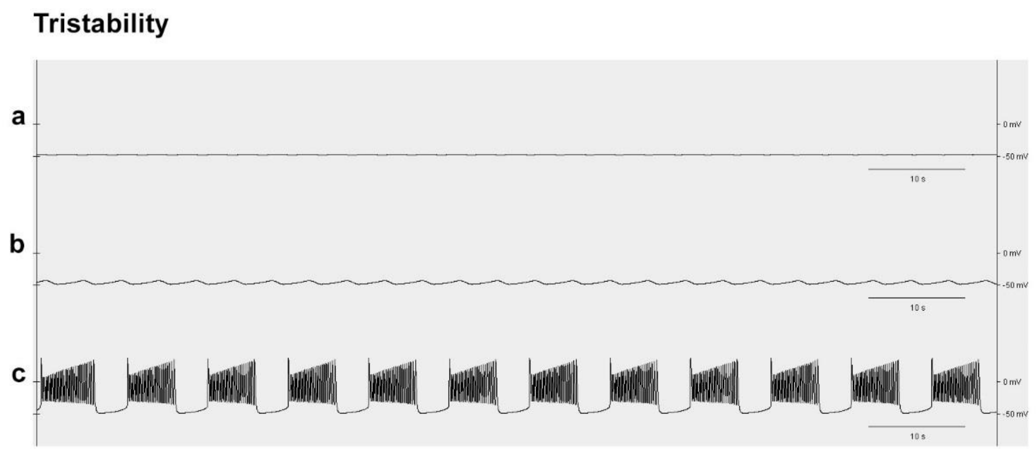

Figure 16. Multistability with three regimes. (a) decaying oscillations which goes to silence, (b) subthreshold oscillations, (c) bursting. The simulations were run for 100 s. The initial values of the state variables were [V h_{Na} h_{CaS}] = [-0.047 0.99 0.012] and m_{CaS} = 0.5 (in a), 0.6 (in b), 0.7 (in c).

In the phase portraits, the stable regimes are visualized. The silence occurred after the values stabilized near the initial values (Fig. 17a). The subthreshold oscillations occurred as a closed cycle with small amplitude (Fig. 17b) which appeared to enclose the silent regime. Bursting appeared as a larger excursion of the state variables (Fig. 17c). It was not concluded from the results if the subthreshold oscillations were truly a stable regime, or if it actually was an unstable limit cycle where the initial values were accidently selected in such a way that the values remained exactly on the unstable cycle throughout the simulation period. This would be interesting task to investigate further.

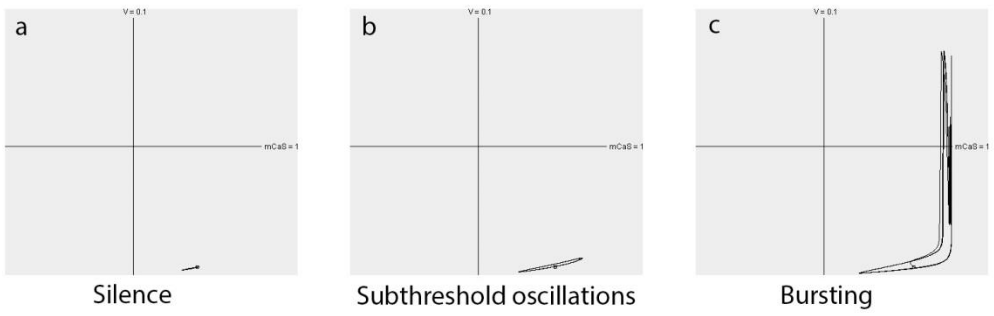

Figure 17. Phase portraits of three regimes where two of the four state variables are chosen for the axes: V and mCaS. The simulations were rum for 100 s. The initial values of the state variables were [V h_{Na} h_{CaS}] = [-0.047 0.99 0.012] and m_{CaS} = 0.5 (in a), 0.6 (in b), 0.7 (in c). The small square in each graph represents the starting point for the simulation.

Multistability and qualitative changes in phase portraits as a parameter is varied are typically studied in bifurcation theory. Therefore, bifurcation analyses would be the natural continuation for exploring multistability and for obtaining understanding of the mathematical background behind the phenomenon seen in the neuron model.

CONCLUSIONS

The leech heart interneuron can be modelled using Hodgkin-Huxley type of equations, resulting in the generation of firing patterns similar to measurements of the real neuron. One such extensive canonical model is represented by 14 coupled differential equations (Hill et al. 2001). To perform a rigorous analysis of this model is a difficult task. Therefore, simplified models have been proven to be an important means for enabling deeper analyses and revealing aspects of action potential generation.

In the work of Malaschenko et al. (2011) a 4-dimensional model is used, representing a simplification of the 14D canonical model. Such a simplification may not always solely represent theoretical models but may in fact be obtained in practice by addition of specific ion-channel inhibitors to the biological systems. Making the simplification of the original model is therefore not only a means for theoretical dissection of the generation of action potential, but may also potentially be validated experimentally.

A Java program developed in this study with the purpose of enabling the exploration of the parameter range of the model of Malashchenko et al. 2014. During the development, different numerical algorithms were tested. Numerical methods with different complexity, ranging from first order explicit Euler to fourth order Runge-Kutta, produces similar bursting patterns in the leech interneuron model provided that the step size in the algorithms is sufficiently small. However, the methods differed in how small step size was required in order to obtain a stable output.

The 4D model is highly simplified compared to the original 14D canonical model. Still the model exhibits a rich variety of stable regimes and includes multistability at certain parameter ranges. Along a range of parameter values of gleak the regimes observed were: silence, tonic spiking, bursting with increasing frequency and again silence. This confirmed the previous results (Mashchenko et al. 2011), and showed the usefulness of the Java program in exploring the parameter space.

Two additional parameters were varied, gNa and gCaS, representing the maximum conductance for each of the respective sodium and calcium ion. A biochemical interpretation of this parameter would be the amount or concentration of functional voltage-gated ion channels in the cell membrane. It is known that in general there is a great variation between individuals and within an individual at different developmental stages.

Different parameters showed a broad spectrum of effects on the firing patterns. Generally the bursting frequency was retained fairly constant 5-6 Hz, while burst duration varied depending on which parameter was changed. When some parameters were changed, the bursting was not greatly affected, while other (such as Eleak) rapidly abolished bursting when varied from the specified default values.

The study of Hodgkin-Huxley type of equations has been proven to be fruitful in understanding aspects of interneuron signaling, but at the same time challenging since it combines concepts from a broad range of sciences such as mathematics, physics, biochemistry and biology. It is at the same time a fascinating thought that in the intellectual attempts to grasp insights in neuron function, the subject of study constitutes the basis for human perception itself.

REFERENCES

Burden, R. L., J. D. Faires (1988). Numerical analysis, fourth edition. PWS-KENT Publishing Company, Boston, USA.

Cocatre-Zilgien, J. H., F. Delcomyn (1992). Identification of bursts in spike trains. Journal of Neuroscience Methods 41:19-30.

Cymbalyuk, G. S., Q. Gaudry, M. A. Masino, R. L. Calabrese (2002). Bursting in leech heart interneurons: Cell-autonomous and network-based mechanisms. Journal of Neuroscience 22:10580-10592.

Cymbalyuk, G., A. Shilnikov (2005). Coexistence of tonic spiking oscillations in a leech neuron model. Journal of Computational Neuroscience 18:255-263.

Edelstein-Keshet, L. (1988). Mathematical Models in Biology. Society for Industrial and Applied Mathematics, Philadelphia.

Frank, J. (2008). Numerical modelling of dynamical systems. Chapter 10. Lecture notes available at http://homepages.cwi.nl/~jason/Classes/numwisk/index.html

Golewash, J. (2014). Ionic current variability and functional stability in the nervous system. BioScience 64:570-580

Hill, A. A. V., J. Lu, M. A. Masino, O. H. Olsen, R. L. Calabrese (2001). A model of a segmental oscillator in the leech heartbeat neuronal network. Journal of Computational Neuroscience 10:281-302.

Izhikevich, E. M. (2000). Neural excitability, spiking and bursting. International Journal of Bifurcation and Chaos 10:1171-1266.

Hodgkin, A. L., A. F. Huxley (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. Journal of Physiology 117:500-544.

Kristan Jr., W. B., R. L. Calabrese, W. O. Friesen (2005). Neuronal control of leech behavior. Progress in Neurobiology 76:279-327.

Malashchenko, T., A. Shilnikov, G. Cymbalyuk (2011). Six types of multistability in a neuronal model based on slow calcium current. PLOS ONE 6:1-10.

Shilnikov, A., R. Gordon, I. Belykh (2008). Polyrhythmic synchronization in bursting networking motifs. Chaos 18:037120.

Shilnikov, A., R. L. Calabrese, G. Cymbalyuk (2005). Mechanism of bistability: Tonic spiking and bursting in a neuron model. Physical Review E 71:056214.

```
//Appendix A
//This is a Java application that models a leech heart interneuron

import javax.swing.*;
import java.awt.*;

class HeartNeuron extends JFrame {

  //Output vektors
  double [] vektor1 = new double[10000];
  double [] vektor2 = new double[10000];
  double [] vektor3 = new double[10000];
  double [] vektor4 = new double[10000];

  //deltaTime will sample every 1/100 second
  //The model is run for 100 seconds
  //i.e. vektor.length * deltaTime = 100
  double deltaTime = 0.01;

  //The actual deltaTime between iterations is deltaTime/timeSplit.
  //Increasing the iterations will improve the accuracy of the numerical
  method.
  //E.g. using timeSplit = 100, the 100 seconds is divided into 100 * 10 000 =
  1 000 000 steps.
  int timeSplit = 100;

  //Starting values of the state variables
  double startV;
  double starthNa;
  double startmCaS;
  double starthCaS;

  //Parameters
  double C;
  double gNa;
  double ENa;
  double gCaS;
  double ECaS;
  double gleak;
  double Eleak;
  double Bh;
  double BhCaS;

  //A canvas for drawing the graph
  Graph4D graph;

//Constructor of the program
HeartNeuron(String title) {

  super(title); //Set the name of the window displaying the graph
  setLayout(new GridLayout(1, 1));
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  //Gives the state variables and parameters their default values
  setDefault();

  //Calculate the model
```

```java
  calculate();

  //Create and draw the graph according to the calculations
  graph = new Graph4D();
  graph.redraw(vektor1, vektor2, vektor3, vektor4);

  setSize(2000, 800); //Set the window size
  add(graph); //Add the graph to the window
  setVisible(true); //Make the window visible
}

//Method for resetting both state variables and parameters to default
public void setDefault() {
  setDefaultState();
  setDefaultParameters();
}

//Method for resetting state variables to default
public void setDefaultState() {
  //State variables
  startV = -0.047;
  starthNa = 0.99;
  startmCaS = 0.7;
  starthCaS = 0.012;
}

//Method for resetting parameters to default
public void setDefaultParameters() {
  //Parameters
  C = 0.5;
  gNa = 250;
  ENa = 0.045;
  gCaS = 80;
  ECaS = 0.135;
  gleak = 15.4;
  Eleak = -0.0502;
  Bh = 0.031;
  BhCaS = 0.06;
}

//A function that calculates numerically the model using one of the numerical methods:
// Euler method
// Semi-implicit Euler method
// Midpoint method
// Modified Euler method
// Heun's method
// Runge-Kutta's fourth order method
public void calculate() {
  double V = startV;
  double hNa = starthNa;
  double mCaS = startmCaS;
  double hCaS = starthCaS;

  for (int i = 0; i < vektor1.length; i++) {

    vektor1[i] = V*1000;   //Membrane potential, 1 mV is 1 pixel
    vektor2[i] = hNa; //Sodium inactivation
    vektor3[i] = mCaS; //Calcium activation
    vektor4[i] = hCaS; //Calcium inactivation

    //An internal iteration making the calculations more accurate, i.e. decreasing step size to deltaTime/timeSplit.
    for (int j = 0; j < timeSplit; j++) {
      double h = (deltaTime/timeSplit);

      //The Euler method
      double Vn = V;
      V += h*calculateV(Vn, hNa, mCaS, hCaS);
      hNa += h*calculatehNa(Vn, hNa, mCaS, hCaS);
      mCaS += h*calculatemCaS(Vn, hNa, mCaS, hCaS);
      hCaS += h*calculatehCaS(Vn, hNa, mCaS, hCaS);


      /**
      //The semi-implicit Euler
      V += h*calculateV(V, hNa, mCaS, hCaS);
      hNa += h*calculatehNa(V, hNa, mCaS, hCaS);
      mCaS += h*calculatemCaS(V, hNa, mCaS, hCaS);
      hCaS += h*calculatehCaS(V, hNa, mCaS, hCaS);
      */

      /**
      //The Midpoint Method
      double k1 = calculateV(V, hNa, mCaS, hCaS);
      double l1 = calculatehNa(V, hNa, mCaS, hCaS);
      double m1 = calculatemCaS(V, hNa, mCaS, hCaS);
      double n1 = calculatehCaS(V, hNa, mCaS, hCaS);
      double k2 = calculateV(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
      double l2 = calculatehNa(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
      double m2 = calculatemCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
      double n2 = calculatehCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS + h*n1/2);
      V += h*k2;
      hNa += h*l2;
      mCaS += h*m2;
      hCaS += h*n2;
      */

      /**
      //The Modified Euler method
      double k1 = calculateV(V, hNa, mCaS, hCaS);
      double l1 = calculatehNa(V, hNa, mCaS, hCaS);
      double m1 = calculatemCaS(V, hNa, mCaS, hCaS);
      double n1 = calculatehCaS(V, hNa, mCaS, hCaS);
      double k2 = calculateV(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
      double l2 = calculatehNa(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
      double m2 = calculatemCaS(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
      double n2 = calculatehCaS(V + h*k1, hNa + h*l1, mCaS + h*m1, hCaS + h*n1);
```

```
        V += h*(k1 + k2)/2;
        hNa += h*(l1 + l2)/2;
        mCaS += h*(m1 + m2)/2;
        hCaS += h*(n1 + n2)/2;
        */

        /**
        //The Heun's method
        double k1 = calculateV(V, hNa, mCaS, hCaS);
        double l1 = calculatehNa(V, hNa, mCaS, hCaS);
        double m1 = calculatemCaS(V, hNa, mCaS, hCaS);
        double n1 = calculatehCaS(V, hNa, mCaS, hCaS);
        double k2 = calculateV(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3,
        hCaS + h*n1*2/3);
        double l2 = calculatehNa(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3,
        hCaS + h*n1*2/3);
        double m2 = calculatemCaS(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3,
        hCaS + h*n1*2/3);
        double n2 = calculatehCaS(V + h*k1*2/3, hNa + h*l1*2/3, mCaS + h*m1*2/3,
        hCaS + h*n1*2/3);
        V += h*(k1 + 3*k2)/4;
        hNa += h*(l1 + 3*l2)/4;
        mCaS += h*(m1 + 3*m2)/4;
        hCaS += h*(n1 + 3*n2)/4;
        */

        /**
        //The Runge-Kutta order four method
        double k1 = calculateV(V, hNa, mCaS, hCaS);
        double l1 = calculatehNa(V, hNa, mCaS, hCaS);
        double m1 = calculatemCaS(V, hNa, mCaS, hCaS);
        double n1 = calculatehCaS(V, hNa, mCaS, hCaS);
        double k2 = calculateV(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS +
        h*n1/2);
        double l2 = calculatehNa(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS +
        h*n1/2);
        double m2 = calculatemCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS +
        h*n1/2);
        double n2 = calculatehCaS(V + h*k1/2, hNa + h*l1/2, mCaS + h*m1/2, hCaS +
        h*n1/2);
        double k3 = calculateV(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS +
        h*n2/2);
        double l3 = calculatehNa(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS +
        h*n2/2);
        double m3 = calculatemCaS(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS +
        h*n2/2);
        double n3 = calculatehCaS(V + h*k2/2, hNa + h*l2/2, mCaS + h*m2/2, hCaS +
        h*n2/2);
        double k4 = calculateV(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);
        double l4 = calculatehNa(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS + h*n3);
        double m4 = calculatemCaS(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS +
        h*n3);
        double n4 = calculatehCaS(V + h*k3, hNa + h*l3, mCaS + h*m3, hCaS +
        h*n3);
        V += h*(k1 + 2*k2 + 2*k3 + k4)/6;
        hNa += h*(l1 + 2*l2 + 2*l3 + l4)/6;
        mCaS += h*(m1 + 2*m2 + 2*m3 + m4)/6;
```

```java
      hCaS += h*(n1 + 2*n2 + 2*n3 + n4)/6;
      */

      }
    }
}

double calculateV(double V, double hNa, double mCaS, double hCaS) {
  double VLocal = (-(gNa*Math.pow((1/(1 + Math.exp(-150*(V+0.028)))),3)*hNa*(V-ENa)+gCaS*mCaS*mCaS*hCaS*(V-ECaS)+gleak*(
  return VLocal;
}

double calculatehNa(double V, double hNa, double mCaS, double hCaS) {
  double hNaLocal = ((1/(1 + Math.exp(500*(V + Bh)))-hNa)/0.0405);
  return hNaLocal;
}

double calculatemCaS(double V, double hNa, double mCaS, double hCaS) {
  double mCaSLocal = ((1/(1 + Math.exp(-420*(V + 0.0472)))-mCaS)/(0.005 + 0.134/(1 + Math.exp(-400*(V + 0.0487)))));
  return mCaSLocal;
}

double calculatehCaS(double V, double hNa, double mCaS, double hCaS) {
  double hCaSLocal = ((1/(1 + Math.exp(360*(V + BhCaS)))-hCaS)/(0.2 + 5.25/(1 + Math.exp(-250*(V + 0.043)))));
  return hCaSLocal;
}

//A method making redrawing the graph function accessible from outside the object.
public void redrawGraph() {
  graph.redraw(vektor1, vektor2, vektor3, vektor4);
  System.out.println("Done calculating");
}

//Mehtods for returning the state variables
public double getV () {
  return startV;
}

public double gethNa () {
  return starthNa;
}

public double getmCaS () {
  return startmCaS;
}

public double gethCaS () {
  return starthCaS;
}

public double getC () {
  return C;
}
```

```java

public double getgNa () {
  return gNa;
}

public double getENa () {
  return ENa;
}

public double getgCaS () {
  return gCaS;
}

public double getECaS () {
  return ECaS;
}

public double getgleak () {
  return gleak;
}

public double getEleak () {
  return Eleak;
}

public double getBh () {
  return Bh;
}

public double getBhCaS () {
  return BhCaS;
}

//Methods for setting the state variables and parameters from outside the object.
public void setV(double value) {
  startV = value;
}

public void sethNa(double value) {
  starthNa = value;
}

public void setmCaS (double value) {
  startmCaS = value;
}

public void sethCaS (double value) {
  starthCaS = value;
}

public void setC (double value) {
  C = value;
}

public void setgNa (double value) {
  gNa = value;
}

public void setENa (double value) {
  ENa = value;
}

public void setgCaS (double value) {
  gCaS = value;
}

public void setECaS (double value) {
  ECaS = value;
}

public void setgleak (double value) {
  gleak = value;
}

public void setEleak (double value) {
  Eleak = value;
}

public void setBh (double value) {
  Bh = value;
}

public void setBhCaS (double value) {
  BhCaS = value;
}

//Main, that creates instances of the program and the control panel.
public static void main (String[] args) {
  HeartNeuron neuron = new HeartNeuron("Heart Neuron");
  Kontrollpanel kontroll = new Kontrollpanel(neuron);
}
}
```

```
//Appendix B

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class Kontrollpanel extends JFrame implements ChangeListener{
  HeartNeuron neuron;

  SpinnerModel control1;
  SpinnerModel control2;
  SpinnerModel control3;
  SpinnerModel control4;
  SpinnerModel control5;
  SpinnerModel control6;
  SpinnerModel control7;
  SpinnerModel control8;
  SpinnerModel control9;
  SpinnerModel control10;
  SpinnerModel control11;
  SpinnerModel control12;
  SpinnerModel control13;
  SpinnerModel control14;


//The constructor
  Kontrollpanel(HeartNeuron parent)  {
    neuron = parent;
    reset();
  }


void reset(){

  JPanel panel  = new JPanel();
        panel.setLayout(new GridLayout(14, 1));


        panel.add(new TextArea("V", 1, 0, 3));
  control1 = new SpinnerNumberModel(neuron.getV(), -0.1, 0.1, 0.001);
        JSpinner spinner1 = new JSpinner(control1);
        control1.addChangeListener(this);
        panel.add(spinner1);

  panel.add(new TextArea("hNa", 1, 0, 3));
  control2 = new SpinnerNumberModel(neuron.gethNa(), 0, 1, 0.01);
        JSpinner spinner2 = new JSpinner(control2);
        control2.addChangeListener(this);
        panel.add(spinner2);

        panel.add(new TextArea("mCaS", 1, 0, 3));
  control3 = new SpinnerNumberModel(neuron.getmCaS(), 0, 1, 0.1);
        JSpinner spinner3 = new JSpinner(control3);
        control3.addChangeListener(this);
        panel.add(spinner3);

        panel.add(new TextArea("hCaS", 1, 0, 3));
        control4 = new SpinnerNumberModel(neuron.gethCaS(), 0, 1, 0.001);
        JSpinner spinner4 = new JSpinner(control4);
        control4.addChangeListener(this);
        panel.add(spinner4);

  panel.add(new TextArea("C", 1, 0, 3));
  control5 = new SpinnerNumberModel(neuron.getC(), 0, 1, 0.1);
        JSpinner spinner5 = new JSpinner(control5);
        control5.addChangeListener(this);
        panel.add(spinner5);

  panel.add(new TextArea("gNa", 1, 0, 3));
        control6 = new SpinnerNumberModel(neuron.getgNa(), 0, 1000, 10.0);
  JSpinner spinner6 = new JSpinner(control6);
        control6.addChangeListener(this);
        panel.add(spinner6);

  panel.add(new TextArea("ENa", 1, 0, 3));
        control7 = new SpinnerNumberModel(neuron.getENa(), 0, 0.1, 0.005);
  JSpinner spinner7 = new JSpinner(control7);
        control7.addChangeListener(this);
        panel.add(spinner7);

  panel.add(new TextArea("gCaS", 1, 0, 3));
  control8 = new SpinnerNumberModel(neuron.getgCaS(), 0, 1000, 10.0);
  JSpinner spinner8 = new JSpinner(control8);
        control8.addChangeListener(this);
        panel.add(spinner8);

  panel.add(new TextArea("ECaS", 1, 0, 3));
  control9 = new SpinnerNumberModel(neuron.getECaS(), 0, 0.3, 0.005);
  JSpinner spinner9 = new JSpinner(control9);
        control9.addChangeListener(this);
        panel.add(spinner9);

  panel.add(new TextArea("gleak", 1, 0, 3));
  control10 = new SpinnerNumberModel(neuron.getgleak(), 0, 50, 0.1);
  JSpinner spinner10 = new JSpinner(control10);
        control10.addChangeListener(this);
        panel.add(spinner10);

  panel.add(new TextArea("Eleak", 1, 0, 3));
  control11 = new SpinnerNumberModel(neuron.getEleak(), -0.1, 0.1, 0.001);
  JSpinner spinner11 = new JSpinner(control11);
        control11.addChangeListener(this);
        panel.add(spinner11);

  panel.add(new TextArea("Bh", 1, 0, 3));
  control12 = new SpinnerNumberModel(neuron.getBh(), 0, 0.1, 0.001);
  JSpinner spinner12 = new JSpinner(control12);
        control12.addChangeListener(this);
        panel.add(spinner12);

  panel.add(new TextArea("BhCaS", 1, 0, 3));
  control13 = new SpinnerNumberModel(neuron.getBhCaS(), 0, 0.1, 0.001);
  JSpinner spinner13 = new JSpinner(control13);
```

```
      control13.addChangeListener(this);
      panel.add(spinner13);

panel.add(new TextArea("Default", 1, 0, 3));
control14 = new SpinnerNumberModel(0, -5, 5, 1);
      JSpinner spinner14 = new JSpinner((control14));
      control14.addChangeListener(this);
      panel.add(spinner14);

      setContentPane(panel);
      setSize(200,600);
      setVisible(true);
}

public void stateChanged (ChangeEvent evt) {
  Object source = evt.getSource();
if (source == control14) {
  neuron.setDefault();
  reset();
} else {
  neuron.setV((double)control1.getValue());
  neuron.sethNa((double)control2.getValue());
  neuron.setmCaS((double)control3.getValue());
  neuron.sethCaS((double)control4.getValue());
  neuron.setC((double)control5.getValue());
  neuron.setgNa((double)control6.getValue());
  neuron.setENa((double)control7.getValue());
  neuron.setgCaS((double)control8.getValue());
  neuron.setECaS((double)control9.getValue());
  neuron.setgleak((double)control10.getValue());
  neuron.setEleak((double)control11.getValue());
  neuron.setBh((double)control12.getValue());
  neuron.setBhCaS((double)control13.getValue());
}

neuron.calculate();
neuron.redrawGraph();
}
}
```

```java
//Appendix C1

import java.awt.*;

class Graph4D extends java.awt.Canvas {

  double [] vektor1;
  double [] vektor2;
  double [] vektor3;
  double [] vektor4;

  int graphWidth = 1500;  //Indicates how many pixels the graph will be drawn

  void redraw(double [] v1, double [] v2, double [] v3, double [] v4) {
    vektor1 = v1;
    vektor2 = v2;
    vektor3 = v3;
    vektor4 = v4;
    this.repaint();
  }

  public void paint(Graphics g) {
    g.setColor(Color.black);
    //Left side of graph
    g.drawLine(100, 200, 100, 400); //Vertical line
    g.drawLine(95, 300, 100, 300); //Draws a line indicating 0 mv
    g.drawLine(95, 350, 100, 350); //Draws a line indicating -50 mV

    //Right side of graph
    g.drawLine(graphWidth+100, 200, graphWidth+100, 400); //Vertical line
    g.drawLine(graphWidth+100, 300, graphWidth+105, 300); //Draws a line indicating 0 mV
    g.drawLine(graphWidth+100, 350, graphWidth+105, 350); //Draws a line indicating -50 mV
    g.drawString("0 mV", graphWidth+110, 305); //Writes 0 mV
    g.drawString("-50 mV", graphWidth+110, 355); //Writes 50 mV

    g.drawLine(1400, 370, 1400 + graphWidth/10, 370); //Bar indicating seconds
    g.drawString("10 s", 1465, 390);

    //g.drawLine(95, 600, 105, 600); //Draws a line indicating 0 A current

    //Drawing the black line
    for (int i = 1; i < vektor1.length; i++) {
      g.drawLine(100 + (i-1)*graphWidth/vektor1.length, 300 -(int)vektor1[i - 1], 100 + i*graphWidth/vektor1.length, 300 -(int)vektor1[i]);
    }
  }
}
```

```java
//Appendix C2

//Graph state plane
import java.awt.*;

class Graph4D extends java.awt.Canvas {

  double [] vektor1;
  double [] vektor2;
  double [] vektor3;
  double [] vektor4;

  int graphWidth = 1500;  //Indicates how many pixels the graph will be drawn

  void redraw(double [] v1, double [] v2, double [] v3, double [] v4) {
    vektor1 = v1;
    vektor2 = v2;
    vektor3 = v3;
    vektor4 = v4;
    this.repaint();
  }

  public void paint(Graphics g) {
    g.setColor(Color.black);

    g.drawLine(100, 300, 500, 300);
    g.drawLine(300, 100, 300, 500);
    g.drawString("V = 0.1", 285, 95);
    g.drawString("mCaS = 1", 505, 305);

    //Indicate initial values
    g.drawRect(300 + (int)(vektor3[0]*200) - 2, 300 - (int)(vektor1[0]*4) - 2,
    4, 4);

    //Drawing the black line
    for (int i = 1; i < vektor1.length; i++) {
      g.drawLine(300 + (int)(vektor3[i-1]*200), 300 - (int)(vektor1[i-1]*4), 300
      + (int)(vektor3[i]*200), 300 - (int)(vektor1[i]*4));
    }
  }
}

```