Chapter 7

C++ server

In the previous chapter, I have shown how the Genesis simulator can be used as a server on the remote computer. In the next phase of the project, I have realized the possibility of using own server implemented in C++. The C++ server is available over the network and responsible for solving differential equations arising from mathematical modelling.

The C++ server is based on the object classes of common neural elements in a hierarchical structure. Owing to their logical structure, they can also be used to learn the main principles of compartmental modelling. Moreover, they are easily extensible, thus aiding in the development of programs for a newly arising modelling tasks.

This chapter is organized as follows. First, the class hierarchy is described and the most important methods found in the object classes are reviewed. Then, the implementation of standard elements is presented. The process of writing the program that is used for the described classes is shown. Second, I present how the numerical integration is realized. The form of the differential equations and the numerical methods arising from modelling are described. Finally, I describe how the input files from a client are transformed for a simulation starting at the server side.

7.1 Classes hierarchy

7.1.1 Basic classes

As was presented in Section 3.2.3 describing principles of compartmental modelling approach, the mathematical result of this approach is a system of ordinary differential equations (ODE), one for each compartment or simulated channel. The system should be solved numerically, i.e. one needs to calculate values of simulated parameters in series for each time step of numerical integration.

All elements are presented in Fig. 7.1, so that the reader can get an idea of the main principles of the hierarchy of neural elements.

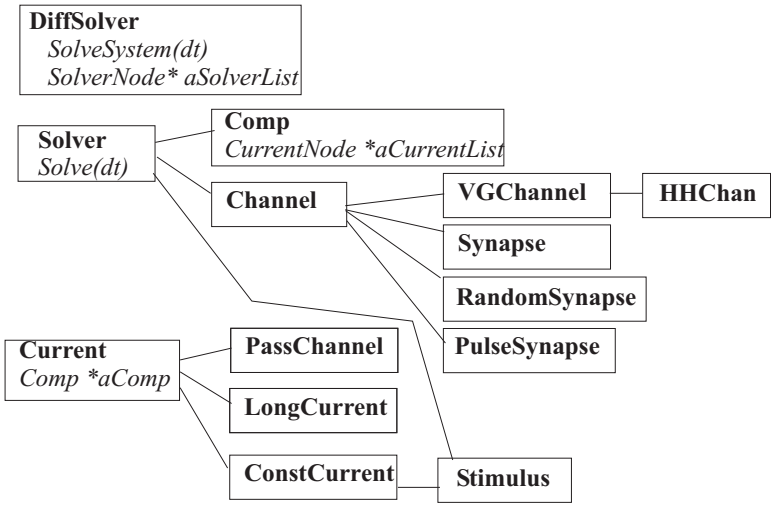

Figure 7.1: Hierarchy of neural element classes.

The class **Solver** is one of the base classes in the objects hierarchy. Each object, for which a solution should be found numerically, has the **Solver** as a base class. The function *Solve(dt)* should be overridden for each Solver subclass. Track of elements of type Solver is kept with the linked list of the class **DiffSolver**, which is designed to control all objects of the type Solver. The linked list is realized in the standard way [65]. The function *Add(Solver &)* adds a new Solver to the list, *Remove(Solver &)* removes a Solver from the list and shifts the pointers.

The function *SolveSystem()* is invoked for solving the system of ODE, which is equivalent to invoking the function Solve(dt) on all Solver objects. One goes from one Solver element in the list (which is linked with pointers) and calls the function Solve(dt) for each of them.

The class **Comp**, representing a compartment, is derived from the class Solver. It has the method Solve(dt) for calculating membrane potential for the next time index; the implementation of the numerical method will be described in the next section. Array $V[]$ is implemented for saving voltage values. The methods *GetV()* and *GetVprev()* provide access to the voltage of the current and previous time indexes. The method *SetV(real aV)* was developed for initialization of new voltage values; it is used for setting initial conditions (starting values of voltages) for numerical integration.

The class Comp has fields describing electrical properties of an isopotential segment of a membrane (according to the compartmental modelling approach): membrane capacitance *Cm*, membrane resistance *Rm*, axial resistance *Ra*, and resting membrane potential *Erest*.

Another important principle of class hierarchy is that each compartment

(element of the type Comp) holds the list of currents, which can be either from adjacent compartments or from channels embedded in the cell membrane. These currents make a contribution to the resulting form of differential equations for membrane voltages. In order to solve a differential equation for a compartment's voltage, one needs to take into account all of these currents.

The class **Current** presents a source of current into a compartment. By initializing a new Current object, a compartment for which this current will be added to the list of currents should be specified. The function *Add(Current* aCurrent)* and the function *Remove(Current* aCurrent)* of the class Comp include an element *aCurrent* in the list and remove it from the list, respectively. These functions are called by a constructor/destructor of the class Comp.

As described in Chapter 3, any channel is characterized by its conductance G and equilibrium potential E_{eq}:

$$I_i = G(E_{eq} - V) \tag{7.1}$$

The class Current contains the protected field *Eeq* and the public field G, corresponding to E_{eq} and G from Eq. 7.1. Field Eeq has been made protected, since its value should not be changed by external methods; one can get access to Eeq with the methods *SetEeq(real E)* and *GetEeq()*. As one can see from Eq. 7.1, the first term depends only on the values of the Current object fields G and E_{eq}; therefore the returning value $Eeq \cdot G$ of the function *GetEeqG()* can later be useful for solving a differential equation.

For a passive ion channel, which also called leakage channel, conductance G is constant. It can also be a variable, for example for voltage-gated or synaptic channels. In the second case, elements of type Current have time-dependent variable conductance G. Thus, the conductance of synaptically activated channels is typically described with a so-called "alpha function"

$$G_{syn}(t) = \frac{const}{\tau_1 - \tau_2} \cdot (\exp(-t/\tau_1) - \exp(-t/\tau_2)) \tag{7.2}$$

Voltage-gated channels have the conductance G, which depends on gating variables resulting in a differential equation (Eq. 7.7). Currents with variable conductance are derived from the class **Solver**, which provides the function Solve(dt) for updating the value of the conductance G.

7.1.2 Implementation of standard neural elements

Compartment of cylindrical form Usually, a compartment of a complex neural structure is approximately that of cylindrical form. The class **CylComp** derived from the class Comp has fields and methods describing the morphology of cylindrical compartments. It has fields for the radius R and length L of a compartment, and fields holding specific electrical characteristics of a compartment: specific resistance RM, specific capacitance CM, and specific axial resistance RA. When initializing an element of type CylComp, the values of membrane resistance R_m, membrane capacitance C_m, and axial resistance R_a

are calculated from the formulas introduced in Section 3.2.2 and Section 3.2.3 according to the membrane area of the cylindrical compartment (Eq. 3.15).

Compartment linking Two compartments can be connected by a pair of longitudinal currents. The class **LongCurrent** derived from the class Current provides a realization of each of these currents. The amount of a longitudinal current, which can flow in two different directions, depends on the values of axial resistance (reverse conductance) and potential of a source compartment. Thus, currents from the left compartment to the right can be calculated as

$$I = \frac{V_m^{''} - V_m}{R_a} \tag{7.3}$$

and in the opposite direction

$$I = \frac{V_m^{'} - V_m}{R_a^{'}} \tag{7.4}$$

where $V_m^{''}$ and $V_m^{'}$ are voltages in the left and right compartments, and $R_a^{'}$ is the axial resistance of the right compartment.

Current stimuli Constant currents injected into compartments can be described with the element **ConstInjection**. Its value does not depend on the potential of the sink compartment. The following "trick" is used to implement this property of a constant current. Inherited from the class Current field, E_{eq} is set to the constant value I and the method *GetEG()* has $E_{eq} = I$ as a return value. The field G is set to zero, so that we get as a result from Eq. 7.1:
$I_i = G(E_{eq} - V) = G \cdot E_{eq} - G \cdot V \Rightarrow GetEG() - G \cdot V = I - 0 = I$.

Pulsed current stimuli, whose values are variable (see Fig. 7.2), are implemented as a **Stimulus** object. The class Stimulus is derived from the classes ConstInjection and Solver. With the method Solve(dt), one can control the time during the pulse interval and set the value of the current:

$$\begin{cases} I = Injection, delay < t < delay + width \\ I = 0, t < delay \end{cases} \tag{7.5}$$

Figure 7.2: Pulsed Stimulus.

Currents through constant-conductance channels Channels with constant conductance (passive channels) can be represented by elements of the type **PassChannel**. The class **PassChannel** contains the methods and fields of the base class Current. They are inherited from the Current class fields *Eeq* and *G*. The field *I* describes the ionic current from this channel, and the method *GetI*() can be used to view its current value.

Channels with variable conductance The class **Channel** is used for describing currents flowing into a compartment from channels with variable conductance. It is derived from the basic classes Current and Solver. The method *Solve(dt)* is available for calculating new conductance values. The method *Solve(dt)* is redefined for the particular elements derived from the class Channel, according to conductance change rules.

The class **VGChannel** is realized as a base class for voltage-gated channels. It has fields *m* and *h* for activation and inactivation gating variables. The fields *mExp* and *hExp* hold the values of the exponential functions of variables m and h, which are used in the function describing conductance of a channel; the field *maxG* holds the maximum conductance of a channel. To calculate the conductance at a given time index using the method Solve(dt)), one needs to solve differential equations for m and h. The implementation of solution of the differential equations for m and h is performed by the functions *SolveM(dt)* and *SolveH(dt)*. The function Solve(dt) then calculates the total conductance $G = maxG \cdot m^{mExp} \cdot h^{hExp}$. To define the dynamics of voltage-gated variables, the functions SolveM(dt) and SolveH(dt) are overridden for each channel.

The mechanism for opening/closing voltage-gated channels is complex and depends on several parameters, e.g. calcium (Ca^{2+}) and potassium (K^+) concentrations and membrane voltage. The diverse types of voltage-gated channels can be implemented as extended objects of the class VGChannel by defining the dependence of conductance on voltage-gated variables.

Hodgkin-Huxley channel The class **HHChan**, describing voltage-gated channel with Hodgkin-Huxley dynamics, is derived from VGChannel. It contains the methods *alphaM(real V)*, *alphaH(real V)*,*betaM(real V)*, and *betaH(real V)*, all of which can take one of three different forms with three parameters A, B, and V0:

- exponential
- linear-exponential
- sigmoid,

whose expressions are given in Section 3.2.1, Eq. 3.2.1.

These functions are defined in the class HHChan as the methods *Exp(real V, A, B, V0)*, *LinExp(...)*, and *Sig(...)*. The class HHChan contains variables describing the form of alpha(V)- and beta(V)-functions:

- alpha_m, alpha_h, beta_m, and beta_h describe the form to be used for alpha(V)- and beta(V)-functions; they can be 1 = exponential, 2 = linear-exponential, or 3 = sigmoid,
- A_alpha_m, A_alpha_h, A_beta_m, A_beta_h, B_alpha_m, B_alpha_h, B_beta_m, B_beta_h, V0_alpha_m, V0_alpha_h, V0_beta_m, and V0_beta_h are constants for alpha(V)- and beta(V)-functions.

Depending on the variables alpha_m, alpha_h, beta_m, and beta_h in functions alphaM(), alphaH(), betaM(),and betaH(), one of the functions in one of the three standard forms (exponential, linear-exponential, sigmoid) will be evoked. All of these variables are initiated by creating a new HHChan object.

Synaptically activated channels and cell connections The class **Synapse** represents a synaptically activated channel, whose conductance is usually described with a time-dependent "alpha function". When the voltage of a presynaptic cell exceeds the threshold voltage, a spike at the presynaptic cell is initiated. It activates a synaptic channel of the postsynaptic cell.

Connections of the two cells can be established by initialization of a Synapse object. For the initialization, the following parameters are necessary: pre- and postsynaptic cells, the threshold voltage of the presynaptic cell for spike initiation *Vthresh*, maximum conductance of a synapse *maxG*, the equilibrium potential *Eeq*, the weight of a connection *(weight)*, and the time constants of the alpha functions *tau1* and *tau2*. Similar to the objects with variable conductance and derived from the class Channel, the function Solve(dt) updates the value of synaptic conductance. One does not need to solve a differential equation, rather, one needs to check the value of the current voltage at the presynaptic cell using the function GetV()), and if $V > Vthresh$, calculate the conductance by the alpha function formula:

$$G(t) = \frac{weight \cdot g_{max}}{\tau_1 - \tau_2} \cdot (\exp(-t/\tau_1) - \exp(-t/\tau_2)), \tag{7.6}$$

Here, the variable t changes its value from 0 to $pulseTime$, which describes duration of the postsynaptic pulse. Therefore, the class Synapse has an additional variable t to describe the time interval from the beginning of the current synaptic pulse. When $t > pulseTime$, conductance G will be set to zero.

Connection delay was implemented in the following way. The class Synapse contains an array VFrom that holds the voltage values of the presynaptic cell of some time interval preceding the current simulation step. The size of this array, which is based on the time interval, is determined by the variable $Vsize = delay/dt$, where *delay* is the connection delay and *dt* is a step size. Thus, if the connection delay was set nonzero, the element of the array corresponding to the preceding time interval defined by $Vsize$ (delay) is compared with Vthresh at the current simulation step. It becomes clear that in this case, the spike arrives at the postsynaptic cell after a certain connection delay.

RandomSynapse A synapse that receives randomly generated spikes is implemented as the class RandomSynapse. It will be activated with an alpha

function at the time when a spike arrives. The following parameters are responsible for the random spike generation: minimum *(min_amp)* and maximum *(max_amp)* amplitudes of a spike event, rate of spike generation *(rate)*, and the minimum time between spikes *(abs_refract)*. Because the conductance of RandomSynapse does not depend on the voltage of the presynaptic cell, the class RandomSynapse is derived from the class Channel. As usual, the method Solve(dt) provides an update of the conductance value. It will have a nonzero value for each generated spike and will have zero during the time interval between spikes. The class RandomSynapse has the same parameters as the class Synapse: delay and weight of activation and parameters of the alpha function.

The synapse activated by pulsed spike events is realized as a **PulseSynapse** object. The pulses are generated in the same way as for Stimulus objects (see Fig. 7.2), i.e. they have the parameters *pulse_delay* and *pulse_width*. PulseSynapse is derived from the class Channel and uses alpha functions for the postsynaptic conductance change.

7.2 Numerical integration

Arising from compartmental modelling, the equations Eq. 3.22, describing the voltage of a generalized compartment, and Eq. 3.6 for gating variables of a Hodgkin-Huxley channel are ordinary differential equations. They are coupled to the system of ODE. It can be seen that both of these equations assume the form

$$\frac{dy}{dt} = A - By, \tag{7.7}$$

where y is a simulated variable, and A and B are functions depending on y and t.

It has been shown that the *exponential Euler method* is the most effective numerical method for equations of this form [55]. In the case of Eq. 7.7, this method shows an optimal combination of accuracy and speed. Therefore, I have implemented it for numerical integration in the C++ class library.

At the time step dt, the solution at the time point $t+dt$ can be approximated by

$$y(t + dt) = y(t)D + A/B(1 - D), \tag{7.8}$$

where

$$D = \exp(-B \cdot dt). \tag{7.9}$$

Let us now consider the implementation of the numerical integration procedure for the elements of our C++ classes. It is realized with the method Solve(dt). The equation describing the compartment voltage (the Comp element) can be written as

$$A = \frac{E_m/R_m + \sum E_{eq}G}{C_m}, \tag{7.10}$$

$$B = \frac{1/R_m + \sum G}{C_m}, \tag{7.11}$$

where the sums are calculated for all Current elements attached to the list of an Comp object. The objects corresponding to the currents of ionic channels, synaptically activated channels, or injected stimuli are derived from the base class Current and use the method GetEeqG() as the class method and G as the field of the class. For currents flowing to neighboring compartments, the potential V_m of the source compartment, which can be got with the method GetV()), is used as E_{eq} and the inverse value of axial resistance R_a is used as G. After calculating A, B, and D, one needs just to update the voltage of the simulated compartment according to Eq. 7.8.

The solution of equations for gating variables of a Hodgkin-Huxley channel (see Eq. 3.6) is implemented by the methods SolveM(dt) and SolveH(dt). The numerical integration is performed as in Eq. 7.8, where A and B are calculated by

$$A = \alpha(V) \tag{7.12}$$
$$B = \alpha(V) + \beta(V) \tag{7.13}$$

The methods SolveM(dt) and SolveH(dt) will update the current values of m and h. Then the method Solve(dt) will update the conductance value G.

7.3 Application of neural element classes

After this brief description of the inheritance and implementation of the classes of neural elements, let us now consider the structure of the simulation program. The simulation program should include the definitions of the model elements. First, one needs to create a number of compartments, typically of cylindrical form, using the class CylComp. The parameters of compartments, such as radius, length, and specific electrical characteristics (RM, CM, RA), should be defined, as well as the starting value of the voltage V_m.

Voltage-activated channels can be added to simulated compartments in two ways: by overriding VGChannel to define the dynamics of gating variables or by using the standard class HHChan for Hodgkin-Huxley channels. Constant or pulsed injection currents can be set as ConstInjection or Stimulus objects.

Compartments are linked by two currents of opposite direction described with objects of the type LongCurrent. The synaptic input can be provided using RandomSynapse objects for random synaptic activation or PulseSynapse for pulsed synaptic activation. A simulated neuron can have a synaptic connection to another neuron, implemented with a Synapse object. Neurons can communicate directly through electrical connections, which are also called gap junctions. This type of connection can be realized using the class LongCurrent.

Numerical integration is performed by the method SolveSystem(dt) of DiffSolver, which is invoked in a loop for a definite number of steps.

7.4 Connection to the C++ server

The realization of the client-server connection, with the Genesis simulator used on the server side, has been given in Chapter 5. Similar to this realization, the connection to the C++ server is implemented using Java RMI. The methods of transferring model description files to the server and simulation results back to the client are implemented in the same way. Transformation of XML model description files is performed using methods of the open C++ transformation library Xerces-C.

7.4.1 Input files and simulation results

NeuroSim has been organized in such a way that one does not need to modify the code of the program when a new simulation model is defined. Instead, the server program gets the model description data in the form of the input file. This program is used by the client that prepares the model description data as the input files for the server. The C++ server program is available remotely through the network.

The model description files are described in XML format. The C++ program transforms input files using the Xerces-C SAX API for XML parsers [13]. The Xerces-C SAX API provides classes for reading XML data, and for the parsing and control of errors. The handler class DocumentHandler is called when XML elements are recognized, and ErrorHandler is called when an error occurs. Using the Xerces-C SAX API, I have created an instance of the SAXParser class and set my own handler for it. To this end, I have derived the MySaxHandler class from HandlerBase and overridden methods in HandlerBase.

Let us consider the HandlerBase methods used for transforming the input XML files. The overridden method *startDocument()* starts the transformation procedure of the XML document. The methods *startElement()* and *endElement()* will be invoked when a handler detects an opening or closing tag. When the opening tag of an element is found by the method *startElement()*, one of several boolean variables of the form *isElement***name** (for example *isElementComp*) is set to "true". For example, by notification of a tag `<comp>` or `<HHChan>`, a compartment or Hodgkin-Huxley channel element will be initialized. The properties of elements are described with the tag `<property>`, which contains the tags `<name>` and `<value>`. After recognizing the property name, the property will be initialized. The method *characters()* reads characters inside an element and sets the property values. When the closing tag is found in the document (using the method *closeElement()*), the modelled object and its parameters are added to the simulation and a variable *isElement***name** is set to "false". The method *endDocument()* closes the parsing of the document when all model elements are designed, and starts the simulation. The simulation step size and number of steps are also obtained from the input file. The method *endDocument()* was overridden to invoke the method *SolveSystem()* of the class DiffSolver.

Simulation results are saved in the files in the form of two columns, which are

the time and the simulated parameter value. Saving of the results is performed by the functions *CreateFile()* and *CloseFile()*. Thus, at each time step, the current value of the membrane voltage for the class Comp and the conductance value of the ionic channels will be saved in a file. The files with simulated data are then transferred to the client for subsequent analysis.

7.5 Conclusions

A classes hierarchy has been developed for performing compartmental modelling of real neural networks. The hierarchy is effective for realizing compartmental modelling of real neural networks.

Common neural elements have been implemented as C++ classes. Based on this implementation, the server side of the NeuroSim system with a client-server architecture has been implemented. C++ was chosen because it is widely recognized as a high-level language effective for performing mathematical calculations and application development; therefore, a large number of class libraries is available. C++ compilers are readily available for practically any operation system.

The C++ classes can also be used independently by researchers with experience in C or C++ programming. Using the standard object-oriented techniques, the implemented classes for common elements can be extended for new modelling tasks.

The NeuroSim client deals with data analysis and its visual presentation. Therefore, a graphical environment for our C++ library is also supplied.