Chapter 12
Introduction to GENESIS Programming

DAVID BEEMAN and MATTHEW A. WILSON

12.1 Simulating a Simple Compartment

Part I of this book has used several existing GENESIS simulations to introduce some of the theory underlying neural modeling. In Part II, we will build upon this background, using the GENESIS script language to create our own simulations. We will begin by simulating a simple neural compartment like that described in the first two sections of Chapter 2. Before proceeding with this tutorial, you may find it useful to review that material.

Figure 12.1 shows the equivalent electrical circuit of the basic passive neural compartment that is provided with GENESIS. Note that it is essentially the same as the "generic" neural compartment shown in Fig. 2.3, except that we have not shown the connections to neighboring compartments, nor added any variable conductance ionic channels. GENESIS provides these basic compartments, various types of channels that may be added, axonal connections to synapses and many other building blocks that are used to construct the simulation.

12.2 Getting Started with GENESIS

This chapter and the ones that follow describe the most important features and syntax of GENESIS. In order to avoid drowning you in a flood of arcane details, we will introduce the

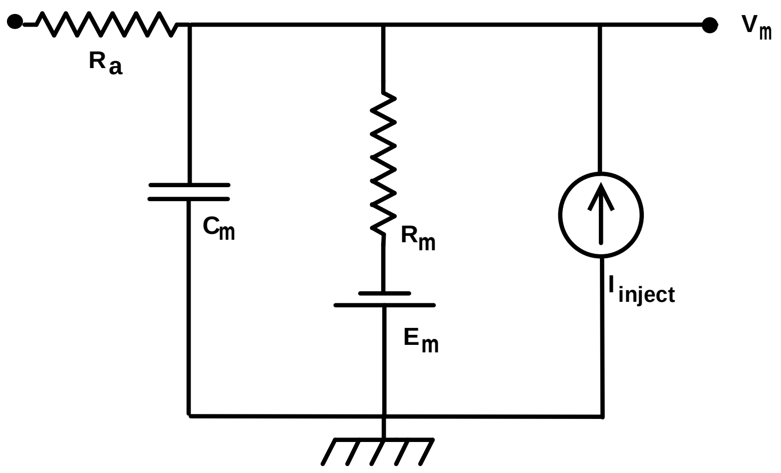

Figure 12.1 The equivalent circuit for a passive neural compartment.

use of GENESIS a little at a time, giving only the information that is needed at each stage of model building. GENESIS is continually evolving, and there will undoubtedly be new features incorporated into future versions. Although GENESIS presently runs only under the UNIX operating system with X-windows, future versions may support other operating systems and graphical interfaces.

Thus, these tutorial chapters should be used in conjunction with the GENESIS Reference Manual, which is distributed as a set of files accompanying the GENESIS distribution. These files may be used to generate the printed manual or may be viewed as part of the on-line GENESIS help facility. The manual is also provided in a hypertext version, so that it may be viewed with a web browser. The reference manual and the example scripts in the GENESIS *Scripts* directory will be periodically updated in order to keep you advised of any changes since the publication of this book. In order to be sure that you have the latest GENESIS distribution, please check the GENESIS WWW or ftp site (Appendix A).

The remainder of this chapter guides you through a brief session under GENESIS and help you to set up and run a simple simulation. To obtain the most benefit from the following sections, you should read them while logged into a workstation on which GENESIS has been installed. At various points, you will be asked to enter GENESIS commands through the keyboard. This will let you try them out and be sure that you understand the various GENESIS commands as they are introduced.

To run the simulator, first make sure that you are at the UNIX shell command prompt. At the prompt type "`genesis`". If your path is properly configured this should start up the simulator and display the opening credits. If you get a message such as "`genesis: Command not found`", check your path (`echo $path`) to be sure that it contains the *genesis* directory (often */usr/genesis*). Your home directory should contain a file named *.simrc*.

If you are still not able to run GENESIS, you or your system administrator should consult Appendix A, “Acquiring and Installing GENESIS”.

After the simulator has completed its startup procedure you should see the GENESIS command prompt, indicating that you are now in the GENESIS Script Language Interpreter (SLI). In the interpreter you can excute both UNIX shell and GENESIS commands. Try this by typing

```
    ls
```

This should invoke the UNIX *ls* command, displaying files in the current directory. Typing

```
    listcommands
```

should produce a list of available GENESIS commands (including *listcommands*). Note that some of these “commands”, like *cos* and *sin*, might more properly be called “functions”, as we may be more interested in the values that they return than any actions that they might perform. Nevertheless, we lump them all together as *commands*, and reserve the term *function* for a command or function that we may write ourselves in the GENESIS script language. Although there are a large number of available commands, you will typically use a much smaller subset of these.

It is also possible to combine GENESIS and UNIX shell commands. Typing

```
    listcommands | more
```

will “pipe” the output of the GENESIS command *listcommands* through the UNIX command *more*, thus allowing you to page through the listing.

```
    listcommands | lpr
```

will “pipe” the output to the printer and

```
    listcommands > myfile
```

will redirect the output into a file called *myfile*.

12.3 GENESIS Objects and Elements

The building blocks used to create simulations under GENESIS are referred to as *elements*. Elements are created from templates called *objects* or *element types*. In order to emphasize this distinction, we give the names of GENESIS objects in boldface type and use italics for the names of the elements that are created from them. The simulator comes with a number of basic objects. To list the available objects type

```
    listobjects
```

To get more information on a particular object type

```
    showobject <name>
```

where “`<name>`” is replaced by any name from the object list.

The **compartment** object is most commonly used in GENESIS simulations to construct parts of neurons. GENESIS also has another type of compartment, the **symcompartment**, in which the axial resistance R_a shown in Fig. 12.1 is symmetrically divided between the two sides of the compartment. However, the **compartment** is more computationally efficient to use, and is adequate for most modeling. As we will be using this object, try the command "`showobject compartment`" at this time. Most commonly used objects are documented more thoroughly with the GENESIS *help* command. For example, to obtain a detailed description of the equivalent circuit for the **compartment** object, type

```
help  compartment | more
```

For more information about the GENESIS on-line documentation, simply type "`help`".

12.3.1 Creating and Deleting Elements

To create an element from an object description, you use the *create* command. Try typing the *create* command without arguments:

```
create
```

This results in a *usage* statement that gives the proper syntax for using this command. Most commands will produce a usage statement if invoked without arguments or followed by "`-usage`". In the case of the *create* command, the usage statement looks like

```
usage:  create object name [object-specific-options]
```

In this exercise we will create a simple passive compartment. In order to keep track of the many elements that go into a simulation, each element must be given a name. To create a compartment with the name *soma*, type

```
create compartment /soma
```

Elements are maintained in a hierarchy much like that used to maintain files in the UNIX operating system. In this case, */soma* is a pathname which indicates that the soma is to be placed at the root or top of the hierarchy.

We will eventually build a fairly realistic neuron called */cell* with a soma, dendrites, channels and an axon. It would be a good idea to organize these components into a hierarchy of elements such as */cell/soma*, */cell/dend*, */cell/dend/Ex_channel*, and so on. If we do this, we need to create the appropriate type of element for */cell*. GENESIS has a **neutral** object for this sort of use. An element of this type is an empty element that performs no actions and is used chiefly as a parent element for a hierarchy of child elements.

To start the construction of our cell, give the commands

```
create neutral /cell
create compartment /cell/soma
```

As we no longer need our original element */soma*, we may delete it with the command

```
delete /soma
```

12.3.2 Examining and Modifying Elements

The commands for moving about within the GENESIS element hierarchy are similar to their UNIX counterparts. For example, to list the elements in the current level of the hierarchy use the *le* (list elements) command

```
le
```

You should see several items listed, including the newly created *cell*.

Each element contains data fields that contain the values of parameters and state variables used by the element. To show the contents of these data fields use the *showfield* command. For example,

```
showfield /cell/soma -all
```

will display the names and contents of the data fields of the "soma", along with some other information such as the number of incoming and outgoing messages. This example also illustrates the use of GENESIS command options. The option ("`-all`") follows any command arguments and may be abbreviated to the shortest unambigous character string, "`-a`", in this case. Notice that the **compartment** object has fields *Vm*, *Em*, *Rm*, *Cm*, *Ra*, and *inject*, corresponding to the labels in Fig. 12.1 and the variables in Eq. 2.1. To display the contents of a particular field, such as the membrane resistance field *Rm*, type

```
showfield /cell/soma Rm
```

When working in GENESIS you are always located at a particular element within the hierarchy which is referred to as the *working element*. This location is used as a default for many commands that require path specifications. For example, the *le* command used above normally takes a path argument. When the path argument is omitted the working element is used and thus all elements located under the working element are listed. To move about in the hierarchy use the *ce* (change element) command. To change the current working element to the newly created soma, type

```
ce /cell/soma
```

Now you can repeat the *showfield* command used above, omitting the explicit reference to the */cell/soma* pathname:

```
showfield -all
```

This should display the contents of the soma data fields. You may find the current working element by using the *pwe* (print working element) command. Try giving the command:

```
pwe
```

Note the analogy between these commands and the UNIX commands *ls*, *cd* and *pwd*. By analogy with UNIX, GENESIS uses the symbols "`.`" to refer to the working element, and "`..`" to refer to the element above it in the hierarchy. Try using these with the *le*, *ce* and *showfield* commands. Likewise, GENESIS has *pushe* and *pope* commands to correspond to the UNIX *pushd* and *popd* commands. These provide a convenient method of changing to a new working element and returning to the previous one. Try the sequence of commands

```
pushe /cell
pwe
pope
pwe
```

The contents of the element data fields can be changed using the *setfield* command. To set the transmembrane resistance of your soma, type

```
    setfield /cell/soma Rm 10
```

You can set multiple fields in a single command as in

```
    setfield /cell/soma Cm 2 Em 25 inject 5
```

Now if you use a *showfield* command on the element you should see the new values appearing in the data fields.

12.4 Running a GENESIS Simulation

Most elements have the capability of performing a self-consistency test that will report problems if some aspect of the element has been improperly specified. This test is invoked with the *check* command. After the simulation is set up, it is a good idea to give the command

```
    check
```

Before running a simulation the elements must be placed in a known initial state. This is done using the *reset* command which should be performed prior to all simulation runs:

```
    reset
```

If you now show the value of the compartment *Vm* field with the command

```
    showfield /cell/soma Vm
```

you will see that it has been reset to the value given by the parameter *Em*. To run a simulation use the *step* command, which causes the simulator to advance a given number of simulation steps. For ten steps, use

```
    step 10
```

If you now display the *Vm* field, you will see that the simulator actually did something and that the value has changed from its initial value due to the current injection. This field is an example of a *state variable*. GENESIS state variables are data fields that are automatically updated by the elements when they are "run" during a simulation. Normally, these are protected with a "readonly" status so that they may be inspected, but may not be modified with the *setfield* command.

12.4.1 Adding Graphics

Although we now have a working simulation, we need a better way to output the results at each simulation step. Although GENESIS provides ways to send data to files for later analysis, most simulations will make use of some graphical output to monitor the course of a simulation. With that in mind, we will attempt to add a graph to the simulation that will display the voltage trajectory of your soma.

Graphics are implemented using graphical objects from the XODUS library that are manipulated using the same techniques described above. The *form* is the graphical object that is used as a container for all other graphical items. XODUS forms are the "windows" that appear in the simulations which you used in the first part of this book. Thus, before making a graph, we need to make a form in which to put it. We will arbitrarily name our form */data*, and create it from the **xform** object.

```
    create xform /data
```

You may have noticed that nothing much seemed to happen. By default, forms are hidden when first created. To reveal the newly created form use the command

```
    xshow /data
```

An empty window should appear somewhere on your screen. To create a graph in this form with the name *voltage*, use the command

```
    create xgraph /data/voltage
```

Note that the graph was created beneath the form in the element hierarchy. This is quite important, as the hierarchy is used to define the nesting of the displayed graphical elements.

Finally, try the command

```
    xhide /data
```

to hide the form and its contents. Then use the *xshow* command to bring it back again. The combination of *xshow* and *xhide* is useful for popping up and putting away menus and graphs, as we have done in the simulations in Part I of this book.

12.4.2 Linking Elements with Messages

Now you have a soma and a graph, but you need some way of passing information from one to the other. Interelement communication within GENESIS is achieved through a system of links called *messages*. Messages allow one element to access the data fields of another element. For example, to cause the graph to display the voltage of the soma you must first pass a message from the soma to the graph indicating that you would like a particular data field to be plotted. This message is established with the command

```
    addmsg /cell/soma /data/voltage PLOT Vm *volts *red
```

The first two arguments give it the source and destination elements. The third argument tells it what type of message you are sending. In this case the message is a request to plot the contents of the fourth argument which is the name of the data field in the soma that you wish to be plotted. The last two arguments give the label and color to be used in plotting this field.

You can now run the simulation and view the results in the graph by giving the commands

```
    reset
    step 100
```

In order to plot another field in the same graph, just set up another message

```
addmsg /cell/soma /data/voltage PLOT inject *current *blue
reset
step 100
```

and you are displaying both the injection current and the voltage. If you type "`step 100`" again without resetting, the simulation will continue on for another 100 steps and the results will be off the scale of the graph. In the next chapter, you will learn how to set the scales for the *x-axis* and *y-axis* of the graph. However, you may always interactively change the scale for a graph axis by clicking the mouse on the axis and dragging it to higher or lower values. Hitting the "a" key while the mouse cursor is inside the graph causes the graph to automatically rescale to hold the plotted data.

Just as the *showfield* command allows you to examine the current value of data fields, the *showmsg* command lets you determine what messages have been established between elements. Both of these commands can be very useful when interactively debugging GENESIS simulations. For example, if you give the command

```
showmsg /data/voltage
```

after performing 100 simulation steps, it will produce the output

```
INCOMING MESSAGES
MSG 0 from '/cell/soma' type [0] 'PLOT' < data = 74.6631 >
    < name = volts > < color = red >
MSG 1 from '/cell/soma' type [0] 'PLOT' < data = 5 >
    < name = current > < color = blue >

OUTGOING MESSAGES
```

Incoming messages 0 and 1 correspond to the two messages that we sent to the graph, along with the values of the accompanying parameters after 100 steps. There are no outgoing messages from the graph. Try this command for */cell/soma* and verify that its outgoing message is consistent with the incoming message for the voltage graph.

You may remove messages with the *deletemsg* command. Try

```
deletemsg /data/voltage 1 -incoming
```

and use *showmsg* again to inspect the messages. Can you produce the same effect by deleting the corresponding outgoing message from the soma?

12.4.3 Adding Buttons to a Form

The **xbutton** graphical element is often used to invoke a function when a mouse button is clicked. Give the command

```
create xbutton /data/RESET -script reset
```

This should cause a bar labeled `RESET` to appear within the "data" form below the "voltage" graph. When the mouse is moved so that the cursor is within the bar and the left mouse button is clicked, the function following the argument `-script` is invoked. Now add another button to the form with the command

```
create xbutton /data/RUN -script "step 100"
```

In this case, the function to be executed has a parameter (the number of steps), so "`step 100`" must be enclosed in quotes so that the argument of `-script` will be treated as a single string.

At this stage, you have a complete GENESIS simulation that may be run by clicking the left mouse button on the bar labeled `RESET` and then on the one labeled `RUN`. To terminate the simulation and leave GENESIS, type either "`quit`" or "`exit`". If you like, you may implement one of these commands with a button also.

At this time, you should use an editor to create a script containing the GENESIS commands that were used to construct this simulation. The script should begin with

```
//genesis
```

and the filename should have the extension ".g". For example, if the script were named "*tutorial1.g*," you could create the objects and set up the messages with the GENESIS command

```
tutorial1
```

If you have exited GENESIS and are back at the UNIX prompt, you may run GENESIS and bring up the simulation with the single command

```
genesis tutorial1
```

In the following chapters, we will modify this script to create a more realistic model of a neuron that may be incorporated into simple neural circuits.

12.5 How GENESIS Performs a Simulation

You should now have a complete working GENESIS simulation, assembled into a script. Let us now examine what we have created and try to understand how it works.

Notice that, unlike a program in C, Pascal, or FORTRAN, there is no explicit looping over time. Although the GENESIS script language has a *for* construct similar to that used in C, it is not used for iteration over time. When a typical GENESIS simulation script is loaded (or the commands are entered interactively) the Script Language Interpreter merely sets up the simulation when it processes these commands. It does this by creating a number of simulation elements, initializing internal data fields with the *setfield* command, and setting up messages between elements. The iteration over time is performed implicitly when the *step* command sets the simulation in motion.

When using the *showobject* command for a particular object type (a **compartment** or **xgraph**, for example), you will notice that in addition to listing the data fields of the object and the types of messages that it may receive, it also lists the *actions* which that type of object may perform. For example, most objects are capable of performing the RESET and CHECK actions. When the *reset* or *check* command is given, each element performs the version of the RESET or CHECK action that is appropriate for its object type. Most GENESIS object types can perform the PROCESS action. This is the action that is performed 100 times when you give the `step 100` command to run the simulation. When it is performed, each element that is capable of this action goes through one iteration, checking for incoming messages and performing the computations necessary to update its internal fields.

With this *object-oriented* arrangement, each element is responsible for knowing how to perform its own actions, and affects other elements only by the exchange of messages. This modularity makes it easy to modify your simulation "on the fly", adding and deleting compartments, ionic channels, or entire groups of cells, or changing variables to be plotted with just a few commands or the click of a mouse button.

An **xbutton** object just responds to mouse clicks, so it doesn't perform a PROCESS action. An **xgraph** object interprets the PROCESS action by looking for incoming PLOT or PLOTSCALE messages and plotting the data that they carry as a function of the current simulation time. For a **compartment** object, "PROCESS" means to look for certain types of incoming messages and to perform the calculations needed to update the *Vm* field. This is done using the current values of its own data fields and any data accompanying incoming messages.

In other chapters, you will be introduced to some of the types of messages that various objects may receive. In general, the name of the message tells the receiving element what to do with the data that are sent with the message. Sometimes a message is used simply to set an internal data field of an element. For example, the INJECT message may be used as a way to set the *inject* field of a compartment at each simulation step. This would be used instead of the *setfield* command if we wanted to continously vary the current injection during the course of a simulation. For example, a **pulsegen** object (a pulse generator) could send its output state with an INJECT message to the soma, in order to provide pulses of current injection, rather than the steady injection current that we used in this simulation. You can find examples of this use of the **pulsegen** object in the *inputs.g* files that form part of the *Cable* and *Neuron* tutorials.

Further details of the internal operation of GENESIS are given in the GENESIS Reference Manual chapter "Customizing GENESIS," which describes how to create your own objects.

12.6 Exercises

1. Add a button that will allow you to exit the simulation.

2. Try some experiments with sending messages to */cell/soma*, instead of setting the *Em* or *inject* fields. You will need some element to send the message, and a field value to be sent. An easy way to accomplish this is to use the *x*, *y* or *z* field value of the */cell* element. Although these fields are normally used for positioning the cell in a network, they may be used for any purpose. Try setting the *x* field of */cell* to 5 and the *inject* field of */cell/soma* to zero. By using an INJECT message from */cell* to */cell/soma*, can you produce results that are equivalent to setting the *inject* field to 5? Does the message affect the actual value of this field? What does the EREST message do?