Chapter 13
Simulating a Neuron Soma

DAVID BEEMAN

13.1 Some GENESIS Script Language Conventions

In the previous chapter, we modeled the simple passive compartment shown in Fig. 12.1. The membrane resistance R_m is in series with a leakage battery E_{rest}, and is in parallel with a membrane capacitance C_m and a current source I_{inject}. Assembled into a script, the commands used would look something like the listing shown in Fig. 13.1. Note the use of "`//`" for comments. Multi-line comments may be entered by bracketing the commented lines with "`/*`" and "`*/`", as in C. This script also illustrates the use of the backslash character, "\", in order to continue a statement (the second *addmsg* command) to another line. This will be useful when entering lengthy *setfield* commands.

In this chapter, we choose cell parameters that are typical of a neuron soma. We will then add voltage-dependent sodium and potassium ion channels to the compartment. Future chapters extend this to a complete multi-compartmental model of a neuron. The necessary background for understanding these sorts of models may be found in Part I of this book.

13.1.1 Defining Functions in GENESIS

As with any programming language, the GENESIS script language allows you to define functions in order to make your programs more modular and easier to modify. These functions should be grouped together at the beginning of the script, preceding any statements that use them. Because we will begin by modifying the simple compartment used in the previous chapter to make it look more like a neuron soma, it is a good idea to rewrite the

```
//genesis - tutorial1.g
// create a parent element
create neutral /cell

// create an instance of the compartment object
create compartment /cell/soma

// set some internal fields
setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5

// create and display a graph inside a form
create xform /data
create xgraph /data/voltage
xshow /data

// send a message (PLOT Vm) to the graph
addmsg /cell/soma /data/voltage PLOT Vm *volts *red
addmsg /cell/soma /data/voltage \
       PLOT inject *current *blue

// make some buttons to execute simulation commands
create xbutton /data/RESET -script reset
create xbutton /data/RUN -script "step 100"
create xbutton /data/QUIT -script quit

check       // perform a consistency check for each element
reset       // initialize each element before starting the simulation
```

Figure 13.1 A GENESIS script to simulate a simple compartment with current injection.

script to use functions for the creation of the soma and the graph. The general form of a function in GENESIS is

```
function <function_name>(arg1, arg2, ...)
    type1 arg1
    type2 arg2
    .
    .
    type_lv1 local_var1
    .
    <commands>
    .
end
```

Here, the data types are one of *int*, *float* or *str*. As in C, arguments are passed by value. An example of a GENESIS function would be:

```
function print_area(length, diameter)
    float length, diameter
    float area
     float PI=3.14159
     area = PI*diameter*length
     echo The area is {area}
 end
```

The command

```
print_area 5 5
```

should produce an output something like

```
The area is 78.5397
```

If you would like the result formatted to show fewer significant figures, try generating the usage statement for the *floatformat* command, or use the *help* command to get further information.

Note that when a GENESIS function is invoked, its arguments are given as a list, separated by spaces (the *command line form*), rather than as a list in parentheses separated by commas (the *function call form*).

This example also illustrates how one sets the value of a variable (*PI*) at the time that it is declared, as well as the use of braces ("curly brackets"). The predefined GENESIS function *echo* prints out its list of arguments to the screen. The braces around "area" cause it to be evaluated to the value represented by the variable *area*, rather than the string of characters "area". The following statements both produce the same output:

```
echo The area is {area}
echo "The area is "{area}
```

In the first statement, the function *echo* has four arguments. In the second, it has only one argument. The only time you may omit the braces when a variable name is to be evaluated is when it is used in an arithmetic expression. Thus, the following two statements are equivalent:

```
area = PI*diameter*length
area = {PI*diameter*length}
```

This is a point of much confusion among beginning GENESIS programers, so it would be a good idea to experiment with writing some simple functions that evaluate and echo the values of variables. Some further examples are given in the GENESIS Reference Manual.

Once you feel comfortable writing GENESIS functions, copy your script to a new file, *tutorial2.g*, and modify it so that a function is used to create the soma compartment. For maximum generality, define a function *makecompartment* that takes an element path name as an argument. Then, the command "`makecompartment /cell/soma`" should create the soma compartment beneath the parent element */cell* and set the internal fields. Also write a function *make_Vmgraph* to create and show the graph for *Vm* with its associated form and buttons. Once you have completed these changes to your script, verify that the simulation still works as before.

13.2 Making a More Realistic Soma Compartment

13.2.1 Some Remarks on Units

The internal fields used in GENESIS elements have no implicit units. In the statement "`setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5`", the *Rm*, *Cm* and *Em* fields could be in ohms, farads and volts. On the other hand, their values (10, 2 and 25) are of magnitudes that might more appropriately be expressed in kilohms ($K\Omega$), microfarads (μF) and millivolts (mV). The choices for these units will determine the units of time and current. Any inconsistency in the units that are used can result in confusion as well as incorrect results! One way to keep confusion to a minimum is to stick to SI (MKS) units. This is the approach taken with the *Neurokit* program and its associated prototype libraries of cell components. Unfortunately, quantities typical of cells tend to have either very large or very small values when expressed in SI units. For this reason, many people prefer to use *physiological units*. This approach was taken in the *MultiCell*, *Neuron* and *Squid* simulations. Table 13.1 may help you to keep the units straight. In this book, we stick to SI units.

Once having settled upon a consistent set of units, we need to find a way to determine the cell parameters, membrane resistance (R_m), membrane capacitance (C_m) and axial resistance (R_a) for a compartment of given dimensions. In order to specify parameters that are independent of the cell dimensions, *specific units* are used. For a cylindrical compartment, the membrane resistance is inversely proportional to the area of the cylinder, so we define a *specific membrane resistance* R_M, which has units of $\Omega \cdot m^2$. The membrane capacitance is proportional to the area, so it is expressed in terms of a *specific membrane capacitance* C_M, with units of farads per square meter. In future chapters, compartments are connected to each other through their axial resistances R_a. The axial resistance of a cylindrical compartment is proportional to its length and inversely proportional to its cross-sectional area. Therefore, we define the *specific axial resistance* R_A to have units of $\Omega \cdot m$.

Quantity	*SI units*	*physiological units*
resistance	ohm (Ω)	kilohm ($K\Omega = 10^3\Omega$)
capacitance	farad (F)	microfarad ($\mu F = 10^{-6}F$)
voltage	volt (V)	millivolt ($mV = 10^{-3}V$)
current	ampere (A)	microampere ($\mu A = 10^{-6}A$)
time	second (sec)	millisecond ($msec = 10^{-3}sec$)
conductance	siemen ($S = 1/\Omega$)	millisiemen ($mS = 10^{-3}S$)
length	meter (m)	centimeter ($cm = 10^{-2}m$)

Table 13.1 Correspondence between SI and physiological units for common quantities used in neural modeling.

For a compartment of length l and diameter d we then have

$$R_m = \frac{R_M}{\pi l d} \tag{13.1}$$

$$C_m = \pi l d C_M \tag{13.2}$$

$$R_a = \frac{4 l R_A}{\pi d^2} \tag{13.3}$$

13.2.2 Building a "Squid-Like" Soma

Our goal is to build a cylindrical soma compartment that has the same physiological properties as those of the squid giant axon studied by Hodgkin and Huxley (1952d). We will make our soma smaller, with both the length and diameter equal to 30 μm, but will use the same specific resistances and capacitances. (Note that when the length and diameter are the same, this will have the same surface area as a spherical soma.)

Therefore, we will begin our modification of the script by declaring and setting some global variables at the very beginning, before the function definitions:

```
// soma parameters - chosen to be the same as in SQUID (but in SI units)
float RM = 0.33333          // specific membrane resistance (ohms m^2)
float CM = 0.01             // specific membrane capacitance (farads/m^2)
float RA = 0.3              // specific axial resistance (ohms m)

// cell dimensions (meters)
float soma_l = 30e-6        // cylinder equivalent to 30 micron sphere
float soma_d = 30e-6
```

We also need to set some potentials. Considering the outside of the cell to be at zero potential, the resting potential inside the soma should be at -70 mV. For consistency

with the notation used in many GENESIS scripts, we call this variable *EREST_ACT*. In many simulations, this would be the value of E_m, the "battery" in series with the membrane resistance. However, Hodgkin and Huxley found it necessary to set E_m to a leakage potential E_{leak} that compensates for current flow through other channels (such as chloride channels) which were not explicitly taken into account in their model. E_{leak} is set to a value that results in no net current flow when the cell is at *EREST_ACT*. This results in E_{leak} being 10.6 *mV* more positive than *EREST_ACT*. Although we will add the ion channels later, now is a good time to define and set the sodium and potassium equilibrium potentials, which we will call *ENA* and *EK*. The script should now contain the additional statements

```
float EREST_ACT = -0.07         // resting membrane potential (volts)
float Eleak = EREST_ACT + 0.0106  // membrane leakage potential (volts)
float ENA   = 0.045               // sodium equilibrium potential
float EK    = -0.082              // potassium equilibrium potential
```

Once these variables have been declared and initialized, modify your *makecompartment* function to take additional arguments for the compartment length, diameter and rest potential. It should then set the *Em* field to the rest potential and calculate and set *Rm*, *Cm* and *Ra* using *RM*, *CM*, *RA* and the compartment dimensions. (As we are not connecting this compartment to another through its axial resistance, the value of *Ra* is irrelevant. Nevertheless, we might as well give it the correct value.) As we would like to make this function general enough to use for creating a dendrite compartment later, the *inject* field should be set after the function is invoked, rather than being set within the function definition. An appropriate value of the injection current for this simulation is 0.3 *nA*, so we should be able to create a soma with fields set to the proper values using the statements

```
create neutral /cell
makecompartment /cell/soma {soma_l} {soma_d} {Eleak}
setfield /cell/soma inject  0.3e-9
```

Before we can plot the results of the simulation, we need to be sure that the graph */data/voltage* is properly scaled. The deceptive simplicity of the previous simulation stems from the fact that the parameters were chosen so that the voltages and times fell within ranges that were consistent with the default values of the fields *xmin*, *xmax*, *ymin* and *ymax* for the **xgraph** object. These values may be inspected with the command

```
showfield /data/voltage -a
```

or

```
showfield /data/voltage xmin xmax ymin ymax
```

Likewise, the *setfield* command can be used to set these fields. When we later add voltage-activated ion channels to our model, we will expect to see action potentials that extend from slightly below the resting potential to a slightly positive value. A reasonable time scale to observe a few action potentials would be about 100 *msec*. In order to make it easy to modify these ranges, we can define some local variables in the *make_Vmgraph* function and modify the relevant part of the script to look something like this:

```
float vmin = -0.100
float vmax = 0.05
float tmax = 0.100  // default simulation time
create xform /data
create xgraph /data/voltage
setfield ^ xmax {tmax} ymin {vmin} ymax {vmax}
```

The caret symbol ("^") is a convenient shorthand to refer to the most recently created element. We could also have used

```
setfield /data/voltage xmax {tmax} ymin {vmin} ymax {vmax}
```

As the vertical scale is no longer appropriate for the plotting of the injection current, you should delete the line that sets up the message to plot the *inject* field. Once these changes have been entered and you have successfully loaded the script with no reports of syntax errors from GENESIS, click the left mouse button on the `RUN` button. Were the results what you expected? They probably were not.

13.2.3 GIGO (Garbage In, Garbage Out)

It is now time for a step in the construction of this simulation that has been delayed for far too long. Before performing any sort of computer simulation, you should analyze the situation and try to predict the main features of the results. Afterwards, look at the simulation results with a critical eye in order to resolve any differences between what you see and what you expected. If the results of the simulation are not what you expect, it is time for more thought. Either your understanding of the processes occurring in the system is incorrect (or incomplete), or there is something wrong with the program. In the former case, these sorts of surprises provide one of the main motivations for performing "computer experiments." By finding explanations for these unexpected results, we have used the simulation to increase our understanding of the system. In this case, the flat horizontal line seen in the V_m plot is an indication that we have neglected something important.

The GENESIS command "`help compartment | more`" will remind you of the equivalent circuit that we are modeling and the differential equation that is being solved. The on-line help shows a circuit diagram and an equation that are equivalent to Fig. 2.3 and Eq. 2.1.

The diagram reveals that the current I_{inject} flows through R_m to create a potential difference that is in series with E_m. (It also shows a variable channel conductance G_k and its associated equilibrium potential E_k but we have not added the ion channels yet.) Without the ion channels or adjacent compartments, it should be a simple matter for you to calculate the steady-state value of V_m. In this case, the equivalent circuit is reduced to that shown in Fig. 12.1, and Eq. 2.1 becomes

$$C_m \frac{dV_m}{dt} = \frac{(E_m - V_m)}{R_m} + I_{inject}. \tag{13.4}$$

Initially, V_m will equal E_m, and the steady state will be reached after a time given roughly by the time constant for charging the membrane capacitance, $\tau = R_m C_m$. You should now give the command "`showfield /cell/soma -a`" in order to inspect the values of these quantities and make some rough calculations. (HINT: You should conclude that V_m will level off at about -0.024 V, with a time constant of about 0.0033 *sec*. If your results are significantly different, you should check the way in which you calculated R_m and C_m from R_M and C_M.) The problem with our simulation seems to lie in the time dependence. Rather than asymptotically reaching the final value of V_m over the 100 simulation steps, we reach it after the first step.

Perhaps you have anticipated this result. The simulator has performed a stepwise numerical integration of a differential equation over 100 time steps. However, there has been no mention of the time interval used for each integration step. The section in the GENESIS Reference Manual on clocks discusses the commands *getclock*, *setclock* and *useclock*. One may specify up to 100 different clocks that have their time steps set by a command of the form "`setclock <clock># <stepsize>`". For example,

```
setclock 0 0.001
```

Clock number 0 is the global simulation clock that we need to set. The default step size is 1.0 in whatever units we are using. This was fine for the time scale that we used in the previous chapter, but is clearly much too large for the present simulation. When designing simulations, you will need to give some thought to the issue of picking an appropriate simulation step size. To some degree, this will be determined by experiment. As a starting point, you should pick a step size that would allow you to draw a smooth curve if you were to make a "connect-the-dots" type plot of the most rapidly varying variable at each time step. If the step size is adequately small, decreasing the size should produce no change in the results. Of course, using too small an integration step can needlessly slow down your simulation and become a source of round off error. As a practical consideration, you might have to decide what is a "tolerable" amount of difference from the ideal.

You can experiment with different step sizes by interactively issuing the *setclock* command to the GENESIS prompt. If you vary the step size, it will be more convenient for you to specify an amount of time for which the simulation should run, rather than a number of

steps to be performed. Fortunately, this may be accomplished by using the `-time` option of the *step* command. Modify the statement that creates the RUN button to read

```
create xbutton /data/RUN -script "step "{tmax}" -time"
```

Notice the use of spaces within the quotes. This prevents the command *step* from running into the value of the variable *tmax*. Likewise, the space before the option string "`-time`" separates it from the time value. If you have any doubts as to whether the string is being parsed correctly, use the *showfield* command to examine the *script* field of the button. In this case, it should evaluate to "`step 0.1 -time`". Often it is easiest to avoid the complexities of building up a command string in this manner by defining a special-purpose function to do the job. For example:

```
str tmax = 0.1   // define a global variable for the run time
.
.
function step_tmax
    step {tmax} -t
end
.
.
function make_Vmgraph
.
    create xbutton /data/RUN -script step_tmax
.
end  // make_Vmgraph
```

Although indiscriminate use of global variables in a program or simulation script should be avoided, the use of the *step_tmax* function with a global variable *tmax* lets you easily change the run time for the simulation. Can you think of an easy way to change the time scale of the graph whenever you change *tmax*?

Once you have found a satisfactory step size, set the simulation clock to this value within your script. You should now have a properly running (but boring) simulation of a passive soma compartment with no voltage activated channels. In the next chapter, we will add some ion channels in order to create action potentials. However, we will first offer some suggestions for tracking down errors in GENESIS simulations.

13.3 Debugging GENESIS Scripts

The GENESIS SLI provides descriptive error messages for most errors in syntax. For example, a misspelling in the line in Fig. 13.1 that creates the soma compartment might cause it to read

```
create compartement /cell/soma
```

You would then receive the messages

```
<tutorial1> line 7
** Error - could not find object 'compartement'
unable to create 'soma'
<tutorial1> line 18
** Error - addmsg : cannot find element '/cell/soma'
<tutorial1> line 19
** Error - addmsg : cannot find element '/cell/soma'
```

Notice that two additional errors occurred because the first one prevented */cell/soma* from being created. At some point your simulation scripts and their bugs will become complex enough that the interpreter will not recognize the exact error and will abort the simulation with only a message to the effect that a syntax error has occurred. In other cases, the scripts will be syntactically correct and will execute, but will produce obviously incorrect results.

The interactive nature of the SLI makes it easy to track down the point at which the simulation went astray. For example, the *le* command will let you know how far you got in creating the simulation elements before a fatal error was encountered. The *showmsg* command will let you see whether the desired linkages were set up between these elements. The *listglobals* command will list the names and values of any variables or functions that had been declared. If everything seems to be in place and properly connected, then use *step* to single step through the simulation, and use *showfield* to see if the element fields are being set to the proper values. As with any programming language, you may embed temporary print (*echo*) commands at critical points in the program to print out status information.

GENESIS also has a *debug* command that takes an integer argument to set the *debug level*. When used with no arguments, it displays the current debug level. The default is level 0. For level 1 or higher, most objects produce additional status information. Typically, increasing the number will increase the amount of information displayed. Unfortunately, a simulation that runs for more than a few steps may flood you with more output than you want. Thus, it is best to perform a single step at a time when using a non-zero debug level.

13.4 Exercises

1. In the *Neuron* tutorial (Chapter 6) the compartment-specific parameters R_M, R_A, C_M and compartment dimensions are given in physiological units. Using the tutorial, inspect the parameters and dimensions of the soma compartment and calculate the values of R_m, R_a and C_m in $K\Omega$ and μF.

2. Calculate the steady-state value of V_m that is expected in this simulation.