Chapter 15
Adding Dendrites and Synapses

DAVID BEEMAN

In the previous chapter's tutorial, we created a single soma compartment with Hodgkin-Huxley sodium and potassium channels. Your script for this simulation should look something like the one given in the listing of *tutorial3.g* in Appendix B. In this simulation, we will build upon this script in order to construct a multi-compartmental neuron with a dendrite compartment containing a synaptically activated channel, an active soma, and an axon.

15.1 Adding a Dendrite Compartment

We will start by making a single dendrite compartment that we will connect to the soma. For a more detailed model of a single neuron, we might have many such compartments linked together, possibly with much branching. In this case, it would be advisable to use one of GENESIS' implicit numerical integration methods that are described in Chapter 20, in order to avoid the need for very small time steps. However, for our two-compartment model, and for models containing only a few compartments, the default method used by GENESIS will provide sufficient numerical accuracy with a moderate integration step size.

An appropriate size for the compartment would be 100 *μm* long and 2 *μm* in diameter, so we add the following definitions to our script after the definitions of the soma dimensions:

```
float dend_l = 100e-6  // add a 100 micron long dendrite
float dend_d =   2e-6  // give it a 2 micron diameter
```

You may wish to verify that these dimensions are consistent with the criterion that the compartment length should be small compared to the electrotonic length of the compartment (Segev, Fleshman and Burke 1989). We can make the dendrite compartment with the same *makecompartment* function that was used to create the soma, giving it the name */cell/dend* and using the dendrite dimensions instead of the soma dimensions. As the dendrite compartment will not have Hodgkin-Huxley channels, the field *Em* should be set to *EREST_ACT*, rather than *Eleak*. Our cell is now becoming complex enough to merit a *makeneuron* function of its own. The main script should create the cell with the function call

```
makeneuron /cell {soma_l} {soma_d} {dend_l} {dend_d}
```

and the statements that make the dendrite compartment and the soma with its channels should go into the definition of this function. As it will make use of the *makecompartment* function, its definition will have to come after that of *makecompartment*.

As we will generally stimulate the neuron with input to the dendrite rather than with current injection, let's now set the soma injection field value to zero instead of using the previous value of 0.3×10^{-9} amperes. This should be done in the main body of the script, after the call to *makeneuron*. For consistency, the initial value of the dialog box in the function *make_control* should also be set to zero.

Now we need to link the dendrite to the soma. In Fig. 2.2, the dendrite compartment would correspond to the "primed" compartment shown at the left. The dendrite compartment needs to send both its axial resistance and its membrane potential at the previous simulation step to the soma compartment. This allows the soma to calculate the current entering from the dendrite compartment. This is done in the first message below, where the dendrite compartment is linked to the soma with a message of the type RAXIAL. This message has two value fields, *Ra* and *previous_state*. The *previous_state* field gives the value of the membrane potential *Vm* at the previous integration step. We use this field rather than *Vm* because GENESIS updates the fields of all the compartments in parallel, and we want each compartment to update its data fields using data from the previous simulation step.

As the dendrite knows its own axial resistance to the soma, it only needs to receive the soma's previous membrane potential in order to update its state. This is accomplished with the second message, which is named AXIAL. If we were to use the variable *path* as the name of the parameter that will be replaced by */cell* when *makeneuron* is invoked, the statements that set up the messages would be

```
addmsg {path}/dend {path}/soma RAXIAL Ra previous_state
addmsg {path}/soma {path}/dend AXIAL  previous_state
```

Now we need to add a synaptically activated channel to the dendrite compartment. The GENESIS object most suitable for this is the **synchan** object. This object was used to create

the synaptically activated channels in the *Neuron* tutorial, which we used in Chapter 6. Some older simulation scripts make use of a similar GENESIS object, the **channelC2**, which is used with the obsolete **axon** object.

The **synchan** may receive delta-function "spike events," each lasting for a single integration time step, from a SPIKE message. It then calculates a net channel conductance *Gk* summing the effects of each spike. The parameter fields *gmax, tau1* and *tau2* are used to determine the time behavior of the conductance. When the *tau1* and *tau2* fields are equal, a single spike impulse gives a conductance with the time dependence

$$Gk = gmax\frac{t}{tau1}exp(1-\frac{t}{tau1}). \tag{15.1}$$

This causes *Gk* to reach a maximum value of *gmax* after a time *tau1*. The initial rise in conductance is linear and the decay is exponential, with time constant *tau1*. This is the "alpha function" form, corresponding to Eq. 6.16.

When the two time constants differ, the conductance assumes a dual exponential form,

$$Gk = \frac{Agmax}{tau1-tau2}(exp(-\frac{t}{tau1})-exp(-\frac{t}{tau2})), \tag{15.2}$$

where *A* is a normalization constant chosen so that *Gk* assumes a maximum value of *gmax*. These fields correspond to the variables that appear in Eq. 6.17. As with voltage-activated channels, there is a field *Ek* for the equilibrium potential of the channel. You may find further information about this object with the GENESIS command "`help synchan`". The mathematics behind the implementation of the **synchan** object are described by Wilson and Bower (1989).

It may be useful to add other channels later, so it would be a good idea to write a fairly general function that lets us specify arguments for the path to the compartment, the name of the channel, and the values of the four parameters *Ek, tau1, tau2*, and *gmax*. You might start the declaration with:

```
function makechannel(compartment,channel,Ek,tau1,tau2,gmax)
```

The body of the function should be written so that the statement

```
makechannel /cell/dend Ex_channel {Ek} {tau1} {tau2} {gmax}
```

will create a channel named */cell/dend/Ex_channel* and will set the fields to the stated values.

The function also needs to link the channel to its parent compartment. As usual, this is done by passing messages. The compartment needs to know the conductance of the channel and the equilibrium potential of the channel (the voltage of the "battery" in series with the conductance). This is used by the compartment in its calculation of the net current flow into the compartment. Although the channel conductance is not dependent on the

membrane potential, the **synchan** object also calculates the channel current, so it needs to receive a message from the compartment that gives the membrane potential. The required messages may be set up with the statements, similar to the ones used with the Hodgkin-Huxley channels in the soma,

```
addmsg   {compartment}/{channel}   {compartment} CHANNEL Gk Ek
addmsg   {compartment}   {compartment}/{channel} VOLTAGE Vm
```

Use this function within *makeneuron* to create an excitatory channel having a sodium equilibrium constant, both time constants equal to 0.003 second, and a *gmax* of 5×10^{-10} siemens.

In order to do anything interesting with our neuron, we will need to give it some synaptic input to the dendrite excitatory channel. However, before continuing, it would be a good idea to test what we have so far. Try running the script as it exists at this stage, and use the dialog box to give the soma an injection current. Does it still work as before? Although there is no dialog box for injection current to the dendrite, you can give it some injection by typing the command "`setfield /cell/dend inject 0.3e-9`" to the GENESIS prompt. With no injection to the soma, does injection to the dendrite still produce action potentials in the soma? Can you detect any difference in the results when injecting the dendrite rather than the soma?

15.2 Providing Synaptic Input

The GENESIS Reference Manual describes various methods of activating a **synchan** element. The most common method is to convert the action potentials produced in another neuron to a sequence of unit amplitude spikes that are sent to the **synchan** with a SPIKE message. As we don't have another neuron to provide synaptic input to */cell/dend/Ex_channel*, it would be nice to have a function that would provide a randomly distributed train of spikes to a given channel. This might represent the random spontaneous "background level" firing of many other neurons that have inputs to our cell. Let's call the function *makeinput(path)*, so that "`makeinput /cell/dend/Ex_channel`" will provide the necessary input. GENESIS has a number of objects that can be used to generate trains of pulses. We will use the **randomspike** object to create an element that produces random spikes. Let's call this element */randomspike*, also.

The **randomspike** object has the settable parameter fields, *rate*, *min_amp*, *max_amp*, *reset*, and *reset_value*. The first of these gives the average number of randomly generated spike events per unit time. When an event has been generated, the amplitude of the event is a random variable uniformly distributed between *min_amp* and *max_amp*. There is also a field named *state* which is updated at every simulation step. The *state* field has the value of the event amplitude if an event has been generated. If an event is not generated, then

the value of the *state* field depends on the *reset* field. If *reset* is non-zero, then *state* takes on the value given in *reset_value*. Otherwise *state* will behave like a latch containing the amplitude of the previous event.

An average of twenty spikes per 0.1 second run of the simulation would be a good spiking rate to use in order to represent the input from several other neurons. We would like them all to be of unit amplitude, and to last for just a single time step. After */randomspike* is created, this can be accomplished with the statement

```
setfield ^ min_amp 1.0 max_amp 1.0 rate 200 reset 1 reset_value 0
```

We can then send the spike train to the channel with

```
addmsg /randomspike /cell/dend/Ex_channel SPIKE
```

Note that the SPIKE message uses no parameters. Whenever a spike event is produced, */randomspike* notifies the channel of the event. The **synchan** regards this as a spike of unit amplitude, without making use of the *state* field of the **randomspike** object.

A typical axonal connection from another neuron involves a propagation delay. This can be calculated from the axonal propagation velocity (on the order of 1 meter per second) and the length of the axon. Typical values of this delay are in the range of 1 to 10 *msec*. In addition, there can be a time lag of slightly less than 1 *msec* between the arrival of a presynaptic event and the postsynaptic response. Also, we may need to scale a large network down to a smaller model. This means that a single synaptic connection in our model might represent several similar inputs in the biological system. Therefore, we would like a way to scale the effect of a single synaptic connection by a "weight" factor as we did in the *Neuron* tutorial experiments described in Sec. 6.5.1.

Each synaptic connection that is established by the addition of a SPIKE message contains a field for the total delay and for the synaptic weight. In this particular case, we don't care about the delay, and the spike rate is fast enough to provide a reasonable amount of input for a single synapse. The synaptic connections are numbered starting with zero, so we would set these fields with the statement

```
setfield /cell/dend/Ex_channel synapse[0].delay 0  synapse[0].weight 1
```

It would be useful to plot the output of */randomspike* on the same graph as we use for the soma *Vm*. This could be done by sending a "`PLOT state`" message to the graph. However, we have a problem with the very different magnitudes of *Vm* and the state of *randomspike*. Fortunately, the **xgraph** object can receive a message called PLOTSCALE which allows one to specify a scale factor and offset for the field to be plotted, in addition to the label and color. The syntax for this message is

```
addmsg source_element dest_graph \
    PLOTSCALE field *label *color scale offset
```

Use this message to plot the spikes, giving them a height of 0.01. After having made these additions to your script, try it out. Are the results reasonable?

A plot of the conductance of */cell/dend/Ex_channel* can help our understanding of the conditions under which action potentials are generated in the soma. There should be room for such a graph to the right of the graph */data/voltage*. Use your understanding of the function *make_Vmgraph* to make an analogous function to create a graph *channel_Gk* within the form */condgraphs*. Set the vertical scale (*ymax*) to $10\,gmax$. Notice the effect of each spike on the channel conductance, and the relationship of the conductance to the production of action potentials.

You should be aware of the fact that we have done some things the hard way in this section in order to illustrate some GENESIS features. For example, the use of the PLOTSCALE message to the graph was not strictly necessary. As the **synchan** object uses a SPIKE message only to detect the existence of a spike event, we are free to set the *min_amp* and *max_amp* fields of */randomspike* to values that are convenient for plotting. If we had been interested only in randomly activating the channel, we could have done it much more easily by setting the **synchan** *frequency* field to the desired frequency of random activation. You may wish to try deleting the SPIKE message (or deleting */randomspike*) and setting the *frequency* field to 200.

15.3 Connections Between Neurons

Normally, our cell would have a synaptic connection to another neuron, as in the *MultiCell* demonstration. In order to see how this would be done, we can provide a positive feedback connection from our cell to itself.

If we were interested in the details of the propagation of action potentials along an axon, we might wish to build a multi-compartmental model of an axon connected to the soma. For most purposes, we can use a much simpler model of an axon, regarding it as a simple delay line for the propagation of spikes. As we have described, the axonal propagation delay is combined with the synaptic latency and is implemented within the **synchan** object.

Typically, a presynaptic terminal releases a quantity of neurotransmitter near the peak of an action potential. This means that we can achieve some computational efficiency by converting each action potential to a single spike before it is sent to the **synchan**, instead of explicitly calculating the postsynaptic response to the time-varying presynaptic membrane potential. (When it is necessary to model the postsynaptic response to a graded non-spiking potential, an ACTIVATION message may be sent to the **synchan**.) We can accomplish this conversion by linking the soma to a **spikegen** object with an INPUT message. The

command "`help spikegen | more`" reveals that this object sets its *state* field to a value *output_amp* for a single time step. This occurs whenever it receives an input greater than the value of the field *thresh* and there has not been a spike for at least the interval specified by *abs_refract*. As the action potentials are quite steep near $Vm = 0$, this would be a good value to use for the *thresh* field of the spike element.

During an action potential, *Vm* will generally be above the threshold for longer than a single time step. As we want each action potential to generate a single spike, we also need to set the *abs_refract* field to a value corresponding to the minimum expected interval between action potentials. A typical refractory period would be on the order of 0.01 seconds. As with the */randomspike* element, we would like the spikes to be of unit amplitude. The necessary statements would be of the form

```
create spikegen /cell/soma/spike
setfield /cell/soma/spike thresh 0 abs_refract 0.010 output_amp 1
addmsg  /cell/soma  /cell/soma/spike  INPUT Vm
```

At this point, you should add similar statements to *makeneuron* that will add a **spikegen** element to the soma, with the name of the cell being specified by the *path* argument.

Once the cell has been created, we can use a SPIKE message to establish a synaptic connection from */cell/soma/spike* to */cell/dend/Ex_channel*, just as for the connection from */randomspike*. As a single synaptic input to a dendrite is generally not sufficient to excite a neuron, we will weight the input by a factor of 10, as if the cell were receiving inputs from ten identical synapses. We will also give the input a propagation delay of 5 *msec*. Try this out, adding the necessary statements to your simulation script. Remember that this second connection will be numbered as *synapse[1]*.

As a final embellishment to our simulation, we can add a "button" to the conductance graph form that toggles this feedback connection on and off. The **xtoggle** widget was mentioned in Chapter 14 as a possible way to toggle the *overlay* flag field of an **xgraph** object. This object is similar to the **xbutton** object, with many of the same fields and options. In addition, the field *state*, which may be accessed with the GENESIS *getfield* function, toggles back and forth between 0 and 1 when the toggle button is clicked with the left mouse button. It also has the fields *offlabel* and *onlabel*. These may be set to the two strings that will be displayed when the toggle state is 0 or 1. As usual, the `-script` option can be used to specify the name of a function to be invoked when the toggle is clicked. Typically, this function will inspect the state of the toggle and use an *if-else* construct to perform the appropriate operation. In our case, the function might look like this:

```
function toggle_feedback
    int msgnum
    if ({getfield /control/feedback state} == 0)
```

```
            deletemsg  /cell/soma/spike 0 -out
            echo "Feedback connection deleted"
        else
            addmsg /cell/soma/spike /cell/dend/Ex_channel SPIKE
            msgnum = {getfield /cell/dend/Ex_channel nsynapses} - 1
            setfield /cell/dend/Ex_channel \
                synapse[{msgnum}].weight 10 synapse[{msgnum}].delay 0.005
            echo "Feedback connection added"
        end
    end
```

In this function, we have made use of the *nsynapses* field of the **synchan** object. As it is often easy to lose track of the number of the most recently created connection, we can use *nsynapses* to find the number of SPIKE messages that exist. It is then decremented by one to take into account the fact that the first synapse is number 0.

Create an **xtoggle** element with appropriate labels in the */control* form and use this function to allow it to toggle the feedback connection on and off. If you have done everything correctly, your simulation results should resemble those in Fig. 15.1, when the feedback connection is "off."

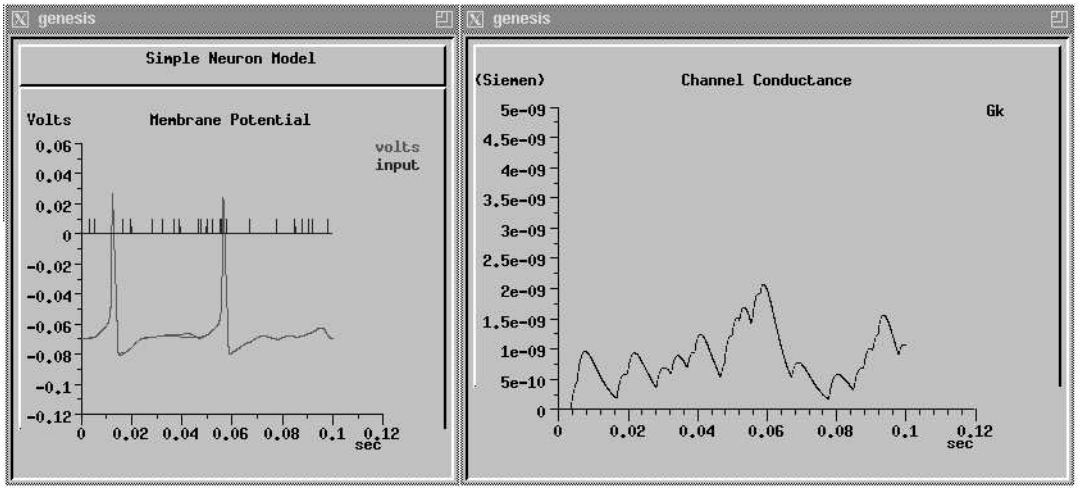

Figure 15.1 Typical results for the simulation when the dendrite excitatory channel is stimulated with random spike events. The feedback connection from the axon has been toggled "off."

15.4 Learning and Synaptic Plasticity

To implement learning or other forms of adaptive behavior in a GENESIS simulation, we need some way to modify the synaptic weight, or to otherwise change the effect of providing synaptic input. The following sections describe some of the ways that this can be done.

15.4.1 Continous Modification of the Synaptic Weight

When implementing learning algorithms, you will likely want to modify specific connection weights, as with:

```
setfield /cell/dend/Ex_channel synapse[0].weight {new_weight}
```

However, you will want to make these changes continously, with changing values of the variable *new_weight*, while the simulation is being stepped.

This could be done with a function written in the GENESIS script langauge. There is a GENESIS object called **script_out** that could be used to invoke this function at specified intervals during the simulation. Alternatively, you could use a **synchan** as the basis for an extended object that performs some weight-changing algorithm as part of its PROCESS action.

15.4.2 Use of the MOD Message

The **synchan** is able to receive a MOD message, which is intended for implementing neuromodulation, but could also be used to cause learned time-dependent modification of synaptic activation. This message simply scales the channel activation for the current time step by a factor that is sent with the MOD message. Note that this globally affects all the synapses in the **synchan**. If you want some synapses to be modifiable, but not others, you should divide your channel into two **synchans** and send a MOD message only to the modifiable one. As with the method of directly setting the synaptic weight fields, you would most likely use a **script_out** or extended object to provide the modification algorithm and calculate a value to be sent to the **synchan** with the MOD message.

15.4.3 Hebbian Learning with the hebbsynchan

Hebb (1949) postulated a simple rule for learning that was based on a correlation of the presynaptic and postsynaptic activity of a neuron. More recently, Hebb's rule has been applied to the understanding of the basis of long term potentiation (LTP), a persistent increase in synaptic efficiency that can be rapidly induced. Brown, Kairiss and Keenan (1990) have given a detailed review of these Hebbian synapses and the biophysical mechanisms that

underlie their behavior. A modern definition of a Hebbian synapse defines it as one that increases its strength with correlated pre- and postsynaptic activity, and decreases its strength with negatively correlated activity. An anti-Hebbian synapse modifies its synaptic strength by rewarding negatively correlated activity and punishing correlated activity.

The **hebbsynchan** object provides for both Hebbian and anti-Hebbian modification of the weight field of a synaptic connection. This object, described in the GENESIS Reference Manual, is very similar to the **synchan**, except that the synaptic weights are not fixed, but vary as a function of both the pre- and postsynaptic activities. In addition to the *weight* and *delay* fields, these synapses have a field called *pre_activity* which represents an averaging of the presynaptic spiking activity through that synapse. Note that each synapse has its own *pre_activity* field, just as each synapse has its own field for its weight and delay. The postsynaptic activity is the same for all synapses in the **hebbsynchan**, and is a function of the averaged membrane potential of the compartment to which the **hebbsynchan** is connected. The *Scripts/examples/hebb* directory contains a demonstration based on the script that was developed in this chapter, but which uses the **hebbsynchan** instead of the **synchan**.

15.4.4 Customizing the synchan or hebbsynchan

If your learning algorithm does not fall into a category that is implemented by the **hebbsynchan**, you should consider writing your own customized synaptic channel object in C, to be compiled into the simulator. Although it may be helpful to use **script_out** or extended objects for initial development and testing of your learning model, your simulation will run faster if you use compiled objects. The GENESIS Reference Manual chapter "Customizing GENESIS" provides detailed instructions for adding new objects and commands to GENESIS. If your new object is based on a modification of the **synchan** or **hebbsynchan**, the section "Creating New Synaptic Objects" will be particularly useful. In order to make modification of the weight change algorithm for the **hebbsynchan** easy, it has been isolated to a single function in the source file *hebbsynchan.c*.

15.5 Where Do We Go from Here?

At this point there are a number of directions to go for learning more about GENESIS programming. The demonstration simulation *MultiCell* connects two neurons such as we have created here in an excitatory-inhibitory loop to produce bursts of pulses. The accompanying file *MultiCell.doc* gives detailed commentary on the syntax of the scripts that are used in the simulation. You may wish to modify your script for this tutorial to create a second neuron and produce your own version of *MultiCell*. It would also be a good idea to study the scripts for the *CPG* simulation from Chapter 8. The *Neuron* tutorial scripts have good examples of functions for providing short trains of spikes as inputs to a synapse.

For learning how to copy a cell into a large array of interconnected cells, the *Orient_tut* demonstration (discussed in Chapter 18) is worthy of study. This simulation also demonstrates some advanced XODUS features using the "draw" (**xdraw**) widget. Some of these are discussed in Chapter 22.

The next two chapters use the GENESIS *cell reader* to create this same cell with a few concise commands by reading a data file. The use of the cell reader is the preferred method for constructing complex neurons with many compartments and channels.

15.6 Exercises

1. Examine the scripts for the *MultiCell* simulation and determine the parameters that were used for the two neurons and the mutual connections between them. Use your *makeneuron* function to create the two cells and then provide synaptic connections with the same characteristics as those in *MultiCell*. Demonstrate that the firing patterns are the same.

2. Try gradually reducing the amount of delay in the feedback connection used in our model. Why does this eventually *increase* the interval between action potentials?

3. At the end of Chapter 14, we mentioned the *overlay* field of the **xgraph** object. Add a toggle button to each of your graphs so that you can switch back and forth between overlay mode.

4. When we connected the soma compartment to the dendrite compartment, we set up the messages so that the membrane potentials of the two compartments were connected through the dendrite's axial resistance. Make another copy of your simulation script that connects the two compartments through the soma's axial resistance. Use the *showmsg* command to verify that the connections are really different.

   Then modify both versions of the simulation so that there are no channels, and provide 0.3 *nA* current injection to the dendrite compartment, instead of to the soma. (Rather than modifying the simulation script, you may find it easiest to delete the channels once the simulation is loaded, and to set the dendrite *inject* field from the GENESIS prompt.)

   Explain the differences between the results that you obtain for the two situations. What happens if you significantly reduce the simulation time step? (You may either use *setclock* for this, or use a dialog box like the one described in Exercise 1 of Chapter 14.) How small does it need to be in order to get accurate results when the two compartments are connected through the soma's axial resistance? Explain this result in terms of the relevant time constants that arise from the resistances and capacitances

in this model. When making a realistic neural model, why is it conceptually wrong to connect the compartments through the soma *Ra*?