Chapter 18
Constructing Neural Circuits and Networks

MICHAEL VANIER AND DAVID BEEMAN

18.1 Introduction

In this chapter, we demonstrate how to use GENESIS to set up a simple network of biologically realistic neurons. This will not be a "neural network" in the usual sense of a network of highly abstract units with no direct connection to biological neurons (such as a back-propagation network). Rather, the approach we take is to simulate a group of biological neurons at a moderate level of detail and then connect them in a network. In the process we discuss a number of GENESIS functions that are used for this purpose, as well as a few script commands that have not been described earlier in this book. Our examples are taken from a tutorial simulation called *Orient_tut*, which is a simplified model of orientation selectivity originally written by Upinder S. Bhalla. This tutorial contains several script files, of which about half deal with setting up the XODUS graphical user interface. We do not discuss these scripts in this chapter; the GENESIS commands used to set up the interface are described in Chapter 22, "Advanced XODUS Techniques." The emphasis in this chapter is on showing you how to use GENESIS commands whose primary purpose relates to simulation of networks of neurons. Commands that have been discussed in detail in other chapters are mentioned only briefly here. Another example of a large network simulation in GENESIS is provided in Chapter 9, on the piriform cortex simulation.

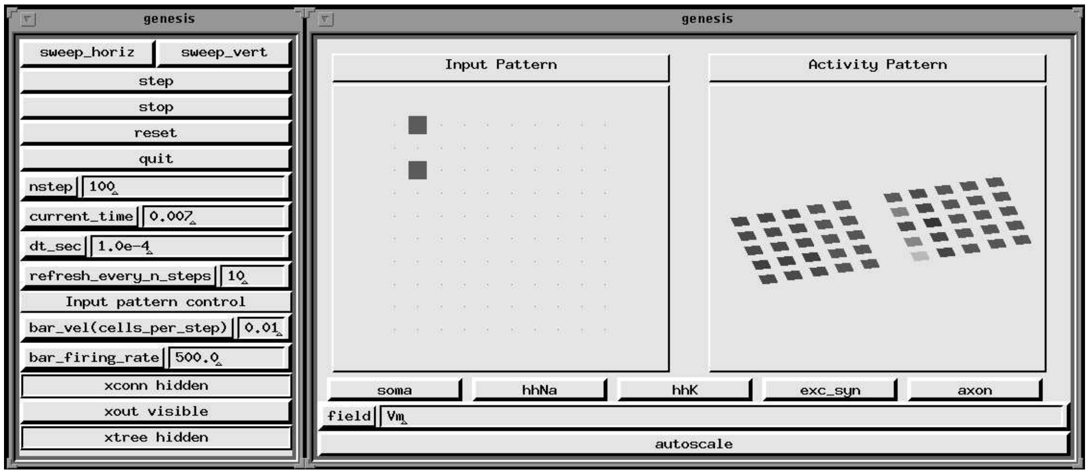
18.2 The *Orient_tut* Simulation

The *Orient_tut* simulation is a simplified model of orientation selectivity in the visual system. It is available in the *Scripts/orient_tut* directory in the GENESIS distribution. This simulation consists of two groups of cells, "retinal" cells and "V1" cortical cells. The retinal cells generate spikes randomly at a rate controlled by the simulation. They in turn make synaptic contacts with two types of V1 cells: horizontal bar detectors and vertical bar detectors. The simulation allows you to sweep a horizontal or vertical bar pattern across the "retina." You will observe that sweeping a horizontal bar across the retina causes the horizontally selective cells in V1 to become activated, whereas sweeping a vertical bar across the retina causes the vertically selective cells in V1 to become activated. As you will see, this selectivity results from the specific patterns of connectivity between the retinal cells and the horizontally and vertically selective V1 cells. The degree of orientation selectivity is quite weak: occasionally a vertical bar will cause a horizontally-selective cell to become active and vice versa. The exercises at the end of the chapter discuss ways of improving the orientation selectivity. One should bear in mind that this is not a very realistic simulation of the mammalian visual system; for instance, there is no representation of the lateral geniculate nucleus (LGN) and there are no feedback connections within V1. The aim of this simulation is not to provide a state-of-the-art model of orientation selectivity but to show in a relatively simple setting most of the commands that are used in setting up network simulations in GENESIS.

18.3 Running the Simulation

The simulation is started by changing to the *Scripts/orient_tut* directory and typing "`genesis Orient_tut`". A graphical display will come up like that in Fig. 18.1. The control panel is in the upper left corner. This contains various buttons, toggles and dialog boxes, all of whose functions are explained in the *README* file for the simulation. For the purposes of this chapter, the most important buttons are the two buttons on the top row, called `sweep_vert` and `sweep_horiz`. Clicking the mouse on `sweep_vert` causes a vertical bar of retinal cells to become active, slowly creeping from left to right, accompanied by some background firing. You can look at the firing pattern of the retinal cells in the display to the right of the control panel, under the label `Input Pattern`. You will see squares that flick from the background color to red and back again; this happens whenever a spike occurs in the corresponding cell. There are 100 such cells in the display, and they are arranged according to their *x*-*y* coordinates. This part of the display is actually composed of two XODUS objects called **xview** and **xdraw**; see Chapter 22 for details. We refer to such a display henceforth as a *draw widget*.

You can stop the simulation by clicking on the `stop` button. If you click on the



nstep 100
current_time 0.007
dt_sec 1.0e-4
refresh_every_n_steps 10
bar_vel(cells_per_step) 0.01
bar_firing_rate 500.0
field Vm

Figure 18.1 The upper portion of the *Orient_tut* simulation display. The window to the right of the control panel shows the retinal receptor cell array and the two planar arrays of horizontally and vertically selective V1 cells.

`sweep_horiz` button, a bar of active cells will creep from the bottom of the draw widget to the top. The outputs of the cells are displayed in a different draw widget, under the label `Activity Pattern`. This widget has two groups of 25 squares, representing the 25 horizontally and vertically selective V1 cells (horizontal to the left, vertical to the right). You can view the value of any of several fields in this widget, but the most useful one (and the default) is the membrane potential (V_m) of the V1 cells. The *Vm* field of the cell's soma compartment is displayed as the color of the square, with blue being hyperpolarized and red being depolarized. When a horizontal bar is swept across the retina, observe how a bar of horizontal V1 cells becomes active and tracks the motion of the bar across the retina. There will also be some activity in the vertically selective cells, but considerably less. Conversely, when a vertical bar is swept across the retina, a bar of vertically selective cells becomes active and tracks the inputs, whereas the horizontally selective cells are much less active. This result follows from the different connectivity patterns of the horizontally- and vertically selective cells. This can be viewed by clicking on the toggle marked `xconn hidden`. This brings up yet another draw widget with three cell displays shown in perspective; the leftmost one represents the retinal cells, the upper right one represents the vertical cells and the lower right one represents the horizontal cells. Clicking the mouse on a retinal cell will show you the projection pattern of that cell in both V1 fields, whereas clicking on a V1 cell will display the receptive field of that cell in the retinal field. You should be able to figure out what confers the orientation selectivity upon the V1 layers by looking at the projective and receptive fields in this way.

For more details on the graphical interface, see Chapter 22 or the *README* file accompanying the simulation. Now, we will discuss the GENESIS commands needed to set up the network for this simulation.

18.4 Creating a Network Simulation

The process of setting up a neural network simulation in GENESIS can be divided up conceptually into several stages. First, the basic low-level components of the simulation that are used to construct cells must be specified. These include compartments, ion channels, and other elements that may include elements to simulate calcium fluxes, detect when spike thresholds have been reached, and so on. The second stage is to link these low-level elements into single cells representing the different cell classes to be incorporated into the model. The third stage is to create arrays of these cells corresponding to the biological structures being modeled. In our case, this means creating an array of retinal cells, and arrays of both vertically and horizontally selective V1 cells. In the fourth stage, we connect the cells in a network, specifying the connection strengths, axonal and synaptic transmission delays, and all other parameters of the synaptic connections. Then, we create any needed external inputs to the system. The final stage is to set up the user interface and output routines that allow the user to run the simulation, view the data being generated, and/or save the data to disk for further analysis.

Of course, you may not always rigorously follow this order. Usually, you will want to do some testing and debugging while you are constructing your network. This may mean providing some input to the system and some graphical display before you have completely finished stage four. Section 18.7.3 describes some utility functions that will be useful in debugging network simulations.

The focus of this chapter is on the third and fourth stages (setting up the network) since other chapters give more details on setting up single cell models and user interfaces. However, we will first briefly describe the manner in which the *Orient_tut* script files handle the first two stages.

18.5 Defining Prototypes

In the *Orient_tut* simulation, the prototype elements are defined in the files *constants.g* and *protodefs.g*. The first script defines the values of constants such as the Nernst equilibrium potentials for the different ion classes (Na, K and Cl), resting potentials, single channel conductances, channel densities, cell dimensions, axonal and synaptic propagation delays, and so on. In some GENESIS simulations the Nernst potentials are computed by the **nernst** object as the simulation is running; however, since the *Orient_tut* simulation is somewhat less detailed, the Nernst potentials are assumed to be constant. The *protodefs.g* script defines the component elements needed to construct the cells. The different objects used include a basic compartment object, active Na^+ and K^+ Hodgkin-Huxley type ionic channels, synaptic channels (depolarizing Na^+ channels, hyperpolarizing K^+ channels and shunting Cl^- channels), and a spike detector **spikegen** object. These objects have been described in previous chapters, especially Chapter 15.

In *protodefs.g* we create the **neutral** element */library* to store the basic synaptic and ionic channel types, as well as a basic compartment element and a spike detector (**spikegen** element). Then we disable the library using the command "`disable /library`" so that the elements in */library* are not simulated during the simulation. The purpose of this is to use the library elements as templates that can be copied to other places where they will be linked into the main simulation, as described in Chapters 16 and 17. In this case they will be used to make prototype cells, which we will copy into the network arrays. Another approach would be to make the library prototypes into GENESIS extended objects, as discussed in Sec. 14.5. However, the approach described here is adequate for our purposes.

After *protodefs.g* has been loaded, all the prototype channels and compartments are subelements of the */library* element. Now we want to use these elements to build up prototype cells for the simulation. There are two types of cells: the receptor cells, which can be thought of as very crude representations of retinal ganglion cells, and the V1 cells, which represent the synaptic "targets" of the retinal cells. These cell types are defined in the script files *retina.g* and *V1.g*, which we describe next.

In *retina.g*, a sample "receptor" cell is created in the library. This is not a real cell in the sense of having compartments or ion channels. It is merely a spike generator that can be connected to postsynaptic cells in the same way as a cell. Only one object is required for this: the **randomspike** object. **randomspike** is a spike generator that generates random (Poisson-distributed) spikes at an average rate set by its *rate* field; for more details, see Chapter 15.

The *retina.g* script creates a prototype receptor cell called */library/rec*, where *library* and *rec* are both **neutral** elements. It then creates the **randomspike** element (random spike generator) called *input* in *library/rec* and sets its average firing rate and absolute refractory period.

The script *V1.g* works in a similar manner, building a V1 cell in the library (as */library/soma*). In this case, the base element (*/library/soma*) is derived from a **compartment** object, not a **neutral** object. We could have made the base element from a **neutral** object, but conceptually all the other parts of the V1 cell are connected to the soma anyway, so this was more convenient. The V1 cell consists of a single compartment (*/library/soma*) that is linked to Hodgkin-Huxley Na^+ and K^+ channels, excitatory and inhibitory synaptically activated channels, and a **spikegen** element, which detects spikes occurring in the cell.

18.6 Creating Arrays of Cells

Now that we have a prototype receptor cell and a prototype V1 cell, we are ready to create the retina and V1 arrays. The statement

```
createmap /library/rec /retina/recplane \
    {REC_NX} {REC_NY} \
    -delta {REC_SEPX} {REC_SEPY} \
    -origin {-REC_NX * REC_SEPX / 2} {-REC_NY * REC_SEPY / 2}
```

in *retina.g* is used to make multiple copies of *rec* and its children into */retina/recplane*, arranging their *x*-*y* coordinates on a two-dimensional grid. */retina/recplane* is a **neutral** element created by the command for the purpose of storing the copies of the receptor cell. The usage for the *createmap* command is

```
createmap source dest Nx Ny -delta dx dy -origin xmin ymin -object
```

where *Nx* and *Ny* are the number of cells on a side in the *x* and *y* directions (giving a total of $Nx \times Ny$ cells in all), *dx* and *dy* are the physical separations in the *x* and *y* directions, and *xmin* and *ymin* give the position of the first element to be created (i.e., with the lowest *x*-*y* values). If the entity to be copied is a GENESIS object in its own right (as opposed to the base of a tree of elements) you should use the `-object` option. This is mainly for use with GENESIS extended objects (see the GENESIS Reference Manual) and is not needed in this simulation. Here *REC_NX* and *REC_NY* (the dimensions of the receptor cell array, i.e., the number of cells on a side in the *x* and *y* directions) are both 10, and the cell separations (*REC_SEPX* and *REC_SEPY*) are each 40×10^{-6} (meters; i.e., 40 *μm* — all units are SI unless otherwise noted).

Thus, the command "`le /retina/recplane`" gives "`rec[0-99]/`", meaning that it has created *rec[0]* through *rec[99]*. The "`/`" indicates that each element has child elements. In this case the only child element will be the **randomspike** element *input*. The "cells" *rec[0]* through *rec[99]* are **neutral** elements because all they need to do is to contain the random spike generator. It would also have been possible to create these copies with the command "`copy /library/rec /retina/recplane/rec[0] -repeat 100`". However, the use of *createmap* assigns values to the *x* and *y* coordinate fields of the elements corresponding to their locations on the grid. These coordinates will be used not only to display the cells in a draw widget, but to assign synaptic connections, synaptic weights and propagation delays.

Occasionally, it might also be useful to specify a non-rectangular geometry for the positions of the retinal cells. For example, we might wish for the retinal cells to be arranged so that they fill up the interior of a circular region centered on the origin with a radius of 200 microns. In this case *createmap* will not work, since it arranges the cells in a rectangular grid. However, we could write a script function to implement this arrangement as follows:

```
function make_circular_retina
  int i,j
  int k = 0
  for(i = -5; i <= 5; i = i + 1)
    for(j = -5; j <= 5; j = j + 1)
      if({i*i + j*j <= 25})
        copy /library/rec /retina/recplane/rec[{k}]
        position /retina/recplane/rec[{k}] \
          {20e-6 * i} {20e-6 * j} 0.0
        k = k + 1
      end
    end
  end
end
```

This function is instructive in that it demonstrates the use of some of GENESIS' looping and conditional constructs. Like any modern programming language, GENESIS has a full range of such commands, including *for*, *foreach*, *while*, *if/else*, and so on. All such commands terminate with an *end* statement. The syntax of these commands is similar to that of the corresponding C functions (or C shell, in the case of *foreach*); for full details, see the GENESIS Reference Manual. In this case the function generates pairs of integers ranging from -5 to 5 and tests whether the sum of their squares is less than or equal to 25, i.e., whether the numbers are within a circle of radius 5 centered at the origin. If so, a retinal cell is copied from the library to the */retina/recplane/rec[]* array and given *x*-*y* coordinates that are scaled versions of *i* and *j* using the *position* command. The *position* command, as its name suggests, sets the position coordinates of an element to equal its last three arguments (which are *x-value*, *y-value*, *z-value*, respectively). You may wish to check that the above function does in fact arrange the cells in a circular array with a diameter of 200 microns centered on the origin.

This function also illustrates another feature of GENESIS programming: if there is no built-in command to perform a particular task, you can usually write a script function to do it. If there is a built-in command, however, it is nearly always more efficient (i.e., faster) than an equivalent script function.

Now we return to the actual simulation. The statement

```
showfield /retina/recplane/rec[0] -all
```

reveals (among other things)

```
xyz        = ( -2.000000e-04 , -2.000000e-04 , 0.000000e+00 )
```

This means that the *x, y, z* positions of the cell are (-200, -200, 0) (in microns). Likewise,

```
rec[1]  --> xyz = ( -1.600000e-04 , -2.000000e-04 , 0.000000e+00 )
rec[5]  --> xyz = (  0.000000e+00 , -2.000000e-04 , 0.000000e+00 )
rec[9]  --> xyz = (  1.600000e-04 , -2.000000e-04 , 0.000000e+00 )
rec[54] --> xyz = ( -4.000000e-05 ,  0.000000e+00 , 0.000000e+00 )
rec[55] --> xyz = (  0.000000e+00 ,  0.000000e+00 , 0.000000e+00 )
```

This shows that all the retinal cells have the same *z* coordinate (0) but are positioned differently in *x*-*y* space.

In a similar manner, the script *V1.g* makes 25 copies of the V1 cell in */V1/horiz* with the statement:

```
createmap /library/soma /V1/horiz \
    {V1_NX} {V1_NY} \
    -delta  {V1_SEPX} {V1_SEPY} \
    -origin {-V1_NX * V1_SEPX / 2} {-V1_NY * V1_SEPY / 2}
```

Executing the command "`le /V1/horiz`" gives "`soma[0-24]/`"; again, the final "`/`" indicates that the array of cells have subelements connected to them. This array is 5 by 5 with spacings of 80×10^{-6} *m* (*V1_SEPX* and *V1_SEPY*) in the *x* and *y* directions. These represent the V1 cells selective to horizontally oriented stimuli. A similar statement is used to create the plane of vertically selective cells (called */V1/vert*). All of the cells in both planes have *z* coordinates of 0.0. In this case the spatial coordinates of the horizontal and vertical cells overlap, but this will cause no problems since the two groups of cells can be referred to and manipulated separately.

At this point we have three arrays of cells, representing the source "retinal" cells, and the vertically and horizontally selective destination "V1" cells. The next step is to set up connections between these cells to form a network.

18.7 Making Synaptic Connections

There are two different ways of specifying the nature of connections between cells in GENESIS. We can individually specify each connection and its associated parameters, or we can use special GENESIS commands that define and parameterize groups of connections between groups of elements. The *Orient_tut* simulation employs the second approach, using the commands *planarconnect*, *planardelay*, and *planarweight* (in the script file *ret_V1.g*). First, though, we will remind you how we manually set up connections in Chapter 15. This approach might be feasible in modeling a system with precise connections between identified cells, such as some invertebrate systems and the central pattern generator circuits treated in Chapter 8. For larger networks with less precise connectivities, the commands operating on groups of elements and connections are more appropriate.

A synaptic connection in GENESIS is made between a spike generating object (such as **randomspike** or **spikegen**) and a synaptic channel object (such as **synchan**). Axons as such are not modeled explicitly (although there is an obsolete **axon** object that is used in some older simulations). In general, the somatic compartment of a cell will be linked to a **spikegen** object, which will detect when a spike has occurred and pass that information along to the synaptic channel object. The V1 cells in our case are set up in this way, although in this simple model the V1 cells are not connected to anything (but see the exercises at the end of the chapter). Alternatively, a **randomspike** object can be used to provide randomly occurring spikes at a given frequency to the synaptic channel object. The retinal cells in our model are modeled as **randomspike** objects, and we impose a firing pattern on the retina by manipulating the firing rates of the retinal cell directly (discussed in Sec. 18.8).

18.7.1 Specifying Individual Synaptic Connections

The connection between the spike generating object and the synaptic object is established by adding a message between the two objects. For instance, to connect the retinal cell `/retina/recplane/rec[0]` with an excitatory synapse on the cell `/V1/horiz/soma[0]`, we could use the following command.

```
addmsg /retina/recplane/rec[0]/input /V1/horiz/soma[0]/exc_syn SPIKE
```

Here, the presynaptic element is a **randomspike** object and the postsynaptic element is a **synchan** object, or synaptically activated channel, discussed in Chapter 15. Since the synaptic connection presumably includes a time delay between the time the spike occurred and the time its influence is felt on the synaptic channel, due to both the axonal and synaptic transmission delays, we have to specify that delay explicitly (the default is a delay of 0, which is not very realistic). Also, the effects of different synapses on their postsynaptic targets will differ in magnitude. This is modeled by a weight field on the synapse. Assuming that the above SPIKE message created synapse number 0 on the **synchan** object, the weight and delay may be set as follows.

```
setfield /V1/horiz/soma[0]/exc_syn synapse[0].weight 2.0 \
     synapse[0].delay 1e-4
```

for a weight of 2.0 and a delay of 0.1 *msec* (10^{-4} *sec*). It should be emphasized that synaptic "weight" is a dimensionless field: a weight of 1.0 means that a single spike will cause a conductance change with a maximum height of *gmax*, the "maximal" conductance of the synapse. However, a weight of 2.0 will cause a conductance change that is twice as large. Thus, the *gmax* field in the **synchan** object refers to the maximal conductance of the channel when the synaptic weight is 1.0.

To make things even easier on us, we could alternatively write a script-level command to set up synapses as follows.

```
function makesynapse(pre,post,weight,delay)
  str pre, post
  float weight, delay
  int syn_num
  addmsg {pre} {post} SPIKE
  syn_num = {getfield {post} nsynapses} - 1
  setfield {post} synapse[{syn_num}].weight {weight} \
              synapse[{syn_num}].delay {delay}
end
```

When we add a SPIKE message, the last synapse corresponds to the message just added. We have to set *nsynapses* to (*nsynapses* $-$ 1) since synapses are numbered starting with 0. Once this function is defined, setting up a synapse is as easy as typing:

```
makesynapse /retina/recplane/rec[0]/input /V1/horiz/soma[0]/exc_syn \
            2.0  1e-4
```

We've just shown you how to set up synapses individually in GENESIS. However, in any reasonably sized network simulation, it would be extremely tedious to set up all the connections in this way. We could use a *for* loop along with the *makesynapse* function defined above, but there is an easier way. GENESIS includes several commands that can be used to set up large numbers of synapses all at once, thus simplifying the process of setting up network-level simulations. This is the approach taken in the *Orient_tut* simulation, and is described next.

18.7.2 Commands Involving Groups of Synapses

Connecting Groups of Synapses

The connections between the retina plane and the two *V1* planes are made in *ret_V1.g* with statements like:

```
planarconnect /retina/recplane/rec[]/input                  \
              /V1/horiz/soma[]/exc_syn                      \
        -relative                                           \
        -sourcemask box -1 -1  1  1                         \
        -destmask   box {-V1_SEPX * 2.4}   {-V1_SEPY * 0.6} \
                        { V1_SEPX * 2.4}   { V1_SEPY * 0.6}
```

The purpose of this rather complex command is to connect a region of identical elements to another region of identical elements (the elements of the second region are not necessarily the same kind of elements as those of the first region). The *planar* in *planarconnect* refers to the fact that the source elements are viewed as lying in a two-dimensional plane. Since

GENESIS objects can have three-dimensional locations, this means that only the *x* and *y* dimensions are used in this command. The command is intended to be used for elements located in a two-dimensional sheet with a constant *z* value, which is true for objects created with the *createmap* command described above. The full usage for this command is:

```
planarconnect source_elements destination_elements \
    [-relative]   \                // relative or absolute connection_mode
    -sourcemask  {box,ellipse} \   // elliptical or rectangular region
     x1 y1 x2 y2 \                 // range of source cells
    [-sourcehole {box,ellipse} \
     x1 y1 x2 y2] \                // range of source cells not to connect
    -destmask    {box,ellipse} \
     x1 y1 x2 y2 \                 // range of destination cells
    [-desthole   {box,ellipse} \
     x1 y1 x2 y2] \                // range of dest cells not to connect
    [-probability p]               // probability of making connection
```

Incidentally, notice that in GENESIS commands that span several lines and that are continued using backslashes (\) you can include comments after the backslashes. The empty brackets (*rec[]* and *soma[]*) indicate that all of the **spikegen** elements in the retinal cells (i.e., */retina/recplane/rec[0-99]/input*) will be connected to the **synchan** elements in */V1/horiz/soma[0-24]/exc_syn*. "`-relative`" means that the (*x*, *y*) coordinates of the destination elements will be measured relative to those of the source elements. The default is to use the absolute coordinates of the destination elements. The `-sourcemask` option specifies the range of source elements to connect, as either a rectangle (box) in the *x*-*y* coordinate space of the elements or an ellipse. For a rectangular (box) region, the coordinates *x*1 and *y*1 refer to the minimum *x* and *y* values of the rectangular region, and the coordinates *x*2 and *y*2 refer to the maximum *x* and *y* values of the rectangular region. For an elliptical region, *x*1 and *y*1 are the coordinates of the center of the ellipse whereas *x*2 and *y*2 are the lengths of the principal axes in the *x* and *y* directions, respectively. The curly braces mean you have to choose one or the other of {*box, ellipse*}.

In this case, the coordinates `-1 -1 1 1` for the sourcemask span a region far larger than the total extent of the source region (2 meters square), so that all source elements will be connected. You can specify multiple source regions by specifying several lines of the form `-sourcemask {box,ellipse} x1 y1 x2 y2`. The `-sourcehole` option indicates a range of elements not to connect; this is useful when you want to connect all elements in a rectangular region except for some inside the region (see below for examples). Again, you can specify multiple "holes" if you want. The `-destmask` and `-desthole` options similarly specify the coordinates of the destination elements. Finally, the `-probability` option specifies the probability of connections, which is 1.0 by default (all elements in the source region(s) specified are connected with all elements in the destination region(s) specified).

The way this works in practice is as follows. GENESIS looks at the list of source elements and rejects those not in the source region. For each source element within the source region, it scans the list of destination elements and picks out those whose position is in the destination region (measured either in absolute coordinates or relative to the specific source element, if the `-relative` option has been selected). Then it makes a synaptic connection between the source and destination elements. If the `-probability` option has been selected the connection will be made with the given probability, so not all possible connections will be made.

Confused? Here are some examples. First, say we wanted the source region to consist of all the *rec* cells except for a rectangular region of 20 microns square in the middle, with the destination cells the same as above. Then you would use the command:

```
planarconnect /retina/recplane/rec[]/input \
    /V1/horiz/soma[]/exc_syn   \
    -relative                  \  // coordinates measured wrt source cells
    -sourcemask box            \  // rectangular region
     -1 -1  1  1               \  // range of source region -- all cells
    -sourcehole box            \  // rectangular region
     -20e-6 -20e-6 20e-6 20e-6 \  // range of source hole
    -destmask box              \
    {-V1_SEPX * 2.4}  {-V1_SEPY * 0.6} \
    { V1_SEPX * 2.4}  { V1_SEPY * 0.6} // range of dest region
```

Alternatively, say we wanted to have two destination regions for the connections between the receptors and the vertically selective V1 cells, one of which includes all the cells whose *x* coordinates are between 10 and 20 *µm* less than the receptor cells' *x* coordinates and one of which includes all the cells whose *x* coordinates are between 10 and 20 *µm* more than the receptor cells' *x* coordinates. Also suppose we wanted to exclude a circular part of the source region centered at the origin and 20 *µm* in diameter, but otherwise include all the source cells. Then we could write this:

```
planarconnect /retina/recplane/rec[]/input \
    /V1/vert/soma[]/exc_syn   \
    -relative                 \ // coordinates measured wrt source cells
    -sourcemask box           \
     -1 -1  1  1              \ // range of source region -- all cells
    -sourcehole ellipse       \
      0  0  20e-6 20e-6       \ // circular region centered at origin
    -destmask box             \
     -20e-6 -1 -10e-6 1       \ // range of first dest region
    -destmask box             \
     10e-6 -1  20e-6 1          // range of second dest region
```

Of course, these examples are just to illustrate the options available; we don't claim that these connection patterns are ideal for generating good orientation selectivity.

There is also a three-dimensional analog to *planarconnect*, which we'll mention here for completeness even though it isn't used in *Orient_tut*. It is *volumeconnect*, with the following usage.

```
volumeconnect source_elements destination_elements \
  [-relative]
  -sourcemask {box,ellipsoid} x1 y1 z1 x2 y2 z2
  [-sourcehole {box,ellipsoid} x1 y1 z1 x2 y2 z2]
  -destmask   {box,ellipsoid} x1 y1 z1 x2 y2 z2
  [-desthole   {box,ellipsoid} x1 y1 z1 x2 y2 z2]
  [-probability p]
```

The syntax is exactly the same as *planarconnect*, except that $x1$, $y1$, and $z1$ refer to the minimum x, y, and z coordinates while $x2$, $y2$, $z2$ refer to the maximum coordinates for a box region; for an ellipsoid region $x1$, $y1$, and $z1$ are the coordinates of the center of the region while $x2$, $y2$, $z2$ are the lengths of the principal axes in the x, y and z regions respectively.

NOTE: if, through an error in syntax, you mistakenly specify source or destination elements that don't exist, no error message will be given. Therefore, it is a good idea to check to see if the connections exist. One way to find out what synaptic connections exist is to use the following command:

```
showmsg /retina/recplane/rec[54]/input
```

This refers to the **randomspike** element called *input*, which is part of receptor cell 54. The output gives the messages sent from this element:

```
MSG 0 to '/V1/horiz/soma[10]/exc_syn' type [-1] 'SPIKE'
MSG 1 to '/V1/horiz/soma[11]/exc_syn' type [-1] 'SPIKE'
MSG 2 to '/V1/horiz/soma[12]/exc_syn' type [-1] 'SPIKE'
(etc.)
```

This shows that the receptor is connected to the excitatory synapses of the somata of several cells. The elements receiving the messages are **synchan** objects. If you have a lot of messages being passed from the source element aside from the SPIKE message, you can type

```
showmsg /retina/recplane/rec[54]/input | grep SPIKE
```

which will only display the SPIKE messages. The *showmsg* command doesn't tell you what the weight or delay is for each connection; that information is stored in the **synchan** objects (on the postsynaptic side). Likewise, you can check that

```
showmsg /retina/recplane/rec[55]/input | grep SPIKE
```

shows targets in the V1 horiz layer of somas 11–19, not including 15.

On the postsynaptic side, we can use the *showmsg* command to display the connections (`SPIKE` messages) coming into a synapse (**synchan** object). For example, if we type

```
showmsg /V1/horiz/soma[24]/exc_syn | grep SPIKE
```

we get

```
MSG 1 from '/retina/recplane/rec[74]/input' type [-1] 'SPIKE'
MSG 2 from '/retina/recplane/rec[75]/input' type [-1] 'SPIKE'
MSG 3 from '/retina/recplane/rec[76]/input' type [-1] 'SPIKE'
MSG 4 from '/retina/recplane/rec[77]/input' type [-1] 'SPIKE'
(etc.)
```

See Sec. 18.7.3 for a more comprehensive way of obtaining information about synaptic connections.

Setting the Delay Fields of Groups of Synapses

The transmission delays are set with the *planardelay* function. In this simulation, the command

```
planardelay /retina/recplane/rec[]/input -radial {CABLE_VEL}
```

uses the *x* and *y* coordinates to calculate the radial distance from each of the source elements (*rec[0]/input* through *rec[99]/input*) to each target for the synaptic connections. This distance is divided by the scale factor, *CABLE_VEL*, in order to assign a value for the delay field of the axon connection. *CABLE_VEL* stands for the velocity of axonal propagation, in m/sec. Dividing the distance between two objects by the propagation velocity gives the time delay for a spike occurring at one cell to reach the postsynaptic cell, assuming that the axons are oriented radially. Note that the distance between the two planes does not enter into this calculation. Also note that source elements for this command must be derived from objects that can send SPIKE messages, which usually means **randomspike** objects or **spikegen** objects.

The full syntax for the *planardelay* function is:

```
planardelay sourcepath
    [-fixed delay]
    [-radial conduction_velocity]
    [-add]
    [-uniform scale]
```

```
        [-gaussian stdev maxdev]
        [-exponential mid max]
        [-absoluterandom]
```

There are several options for the *planardelay* function. The first two options (`-fixed` and `-radial`) are mutually exclusive. `-fixed` means that the delays from the source are all nominally equal to `delay`. `-radial` means that the delays from the source are scaled according to the radial distance between the source and the targets. *conduction_velocity* represents the conduction velocity of the spike along the (hypothetical) axon. As mentioned above, the computed delay between two elements equals the radial distance between the elements (computed by the function) divided by the conduction velocity. The `-add` option causes the delays computed using either the `-fixed` or `-radial` commands to be added to the preexisting delay (the default is to simply replace the existing delay with the new value). This can be useful if you are modeling cells connected by fiber tracts that have sections with different conduction velocities, for example, when fast-conducting axons give rise to slower-conducting axon collaterals. In that case you can call *planardelay* once to set up the delays from the axon and call it again using the `-add` option to add the delays from the axon collaterals using a different conduction velocity.

The other options represent ways of adding random components to the delays. Since these same options are used for several commands, they are discussed in further detail later in this section under "**Adding Randomness to Weights and Delays**."

There is also a three-dimensional analog of this command, called *volumedelay*, with the same syntax,

```
    volumedelay path
        [-fixed delay]
        [-radial conduction_velocity]
        [-add]
        [-uniform scale]
        [-gaussian stdev maxdev]
        [-exponential mid max]
        [-absoluterandom]
```

The only difference between *planardelay* and *volumedelay* is that *volumedelay* calculates the radial distance using all three dimensions instead of just the *x* and *y* dimensions.

There is also a separate command called *syndelay*, for adding a small synaptic component to the delays. This is useful when cells are very close together and the delay calculated using the `-radial` option of *planardelay* or *volumedelay* is unrealistically small. The usage of this command is:

```
    syndelay path delay
        [-add]
        [-uniform scale]
        [-gaussian stdev maxdev]
        [-exponential mid max]
        [-absoluterandom]
```

In this case the path specification is to a group of postsynaptic objects (i.e., **synchan** elements). The delay is equal to the "delay" argument of the command, with the appropriate random component added. If you want to add this delay to a delay previously determined using *planardelay*, say, use the `-add` option as with *planardelay*. If not, the computed delays become the delays of the synapses and if you want to add axonal delays you will have to use *planardelay* or *volumedelay* with the `-add` option. In general, one usually sets the axonal delay first and then adds on the synaptic delay if desired. It is important to note that the synaptic delay is NOT a separate field in the synapse; the axonal and synaptic delays are lumped together in a single "delay" field.

Setting the Weight Fields of Groups of Synapses

Finally, we have to assign weights to the synapses, since the *planarconnect* function initializes all weights to zero. In the *Orient_tut* simulation, this is done with the command

```
planarweight /retina/recplane/rec[]/input -fixed 0.22
```

This command is of the form

```
planarweight sourcepath
        [-fixed weight]
        [-decay decay_rate max_weight min_weight]
        [-uniform scale]
        [-gaussian stdev maxdev]
        [-exponential mid max]
        [-absoluterandom]
```

where *sourcepath* normally refers to the **spikegen** or **randomspike** elements of the source cells. In this command, the first options (`-fixed` and `-decay`) are mutually exclusive and determine whether the weights fall off with distance from the source. The `-fixed` option makes all weights from that sourcepath nominally equal to *weight*. The `-decay` option works like *planardelay* calculating a radial distance between the source elements and each target. The parameter *decay_rate* gives the rate for an exponential decay of the weights with radial distance. Note that *decay_rate* has the units of $meters^{-1}$. The function describing the weight as a function of *max_weight* and *min_weight* for this option is

$$\begin{aligned} weight = {} & (max\_weight - min\_weight) \times \exp(-decay\_rate \times radial\_distance) \\ & + min\_weight. \end{aligned}$$

The reason one might want weights that decay exponentially with distance depends on the conceptual framework of the simulation. If each synapse in your network simulation is intended to represent one synapse in the real system, then the weights could have any (physiologically reasonable) value, and it might be best to set them to a random value within reasonable limits if you had no *a priori* reason to set them to particular values for particular cells. On the other hand, since a simulator typically models a large network of neurons with a much smaller number of simulated cells, each cell can be thought of as representative of a group of cells in a particular region. In this case, it makes sense that, on average, simulated cells close together will have stronger connections between them (i.e., synapses with larger weights) than simulated cells located farther apart from each other, for the simple reason that the real cells of which the simulated cells are representative will in general form more connections between them if they are close together than if they are far apart. It has to be kept in mind that unless you intend to model every cell in the network, each cell is really an abstraction of a class of cells, and the strength of the synapses has to reflect the nature of this abstraction. If you don't want exponential decay of weights, you should use the `-fixed` option. In the *Orient_tut* simulation we use the `-fixed` option since we aren't modeling connections between V1 cells (but see the exercises at the end of the chapter). Remember that if you don't use *planarweight* (or *volumeweight*, described next), you have to set the weights explicitly, since they are equal to zero by default.

As usual, there is also a three-dimensional analog of this command, called *volumeweight* with the same syntax and function:

```
volumeweight sourcepath
        [-fixed weight]
        [-decay decay_rate max_weight min_weight]
        [-uniform scale]
        [-gaussian stdev maxdev]
        [-exponential mid max]
        [-absoluterandom]
```

The only difference between *volumeweight* and *planarweight* is that, for the `-decay` option, the distances are calculated using all three dimensions instead of just the *x* and *y* dimensions.

Adding Randomness to Weights and Delays

The above commands for setting weights and delays have a set of options for adding a random component to the weights and delays set. These options are the same for all these commands, and have the form:

```
<command>
<command-specific options>
```

```
            [-uniform scale]
            [-gaussian stdev maxdev]
            [-exponential mid max]
            [-absoluterandom]
```

Each of the first three options selects a random number out of a particular probability distribution. The *scale* option of `-uniform` gives a random number uniformly distributed in the range {*-scale, scale*}. The `-gaussian` option gives a Gaussian-distributed random value with a mean of zero, a standard deviation of *stdev*, and a maximum deviation of *maxdev*. The `-exponential` option gives an exponentially distributed value with a minimum value of zero, *1/e* point (i.e., the point at which the probability density function has decayed to *1/e* of its maximum value) at *mid* and maximum value of *max*. The *max* or *maxdev* arguments are useful in cases where you want to truncate the ends of a distribution to prevent weights and delays from being larger or smaller than some limit. For instance, a suitable choice of *max* or *maxdev* will prevent the possibility of setting weights or delays to a negative value, which is biologically meaningless. However, as an added precaution, the commands for setting groups of weights and delays will set negative values that may arise from using the random options to zero.

The way these options are used is as follows. First the weight or delay is calculated according to the command-specific options of the command. Let's say that *val* is the value of a weight or delay before any randomness is added. After the random number is included, we have

$$val = val + (val \times random\_number)$$

unless the `-absoluterandom` option is used, in which case we have

$$val = val + random\_number$$

In other words, normally the value of the random number is scaled to the existing weight or delay before adding it to the weight or delay. This is reasonable in most cases, since you usually want to add a certain proportion of variability to all weights or delays. Thus it's easy to add, say, up to 10% randomness to your weights — just use `-uniform 0.1`. If you want to add random numbers from the same distribution to all weights or delays regardless of their original size, use the `-absoluterandom` option (which you can abbreviate as `-abs`).

In all these cases, the random number is chosen separately for each synaptic connection. The command "`randseed <number>`" will initialize the random number generator with *number*. If *randseed* is called with no arguments it will initialize the random number generator with the current time, giving random numbers that will be different each time you run the simulation.

Here are a couple of examples. In the above case, if you wanted to have delays corresponding to conduction velocities uniformly distributed between 1 and 2 m/sec (i.e.,

1.5 ± 0.5 *m/sec*, or $1.5 \pm 33\%$) you could type

```
    planardelay /retina/recplane/rec[]/input -radial 1.5 -uniform 0.33
```

(We're assuming you're using SI units here.) Note that the scaling is important here, since the delays are calculated by dividing the distance by the conduction velocities. Thus, delays corresponding to nearby elements will be shorter than those corresponding to elements separated by a greater distance. Therefore, it is important that the random component of the delay be scaled to a value that is a constant proportion of the total delay.

If you wanted weights normally distributed with a mean of 2.0, a standard deviation of 10% of the mean (i.e., 0.2), and a maximum deviation of 40% of the mean (0.8), giving the weights the range of $\{1.2, 2.8\}$, you could type

```
    planarweight /retina/recplane/rec[]/input -fixed 2.0 -gaussian 0.1 0.4
```

In this case, we could have used the `-absoluterandom` option with the arguments `-gaussian 0.2 0.8` to get the same effect, since here the weights are all the same at the beginning. It is very important to bear in mind that the arguments to the options involving randomness are relative to the actual weight or delay value unless the `-absoluterandom` option is used.

After setting up weights and delays with the above commands, it would be a good idea to check that they are in the desired ranges. We could use *showfield* for this, but it would be a tedious procedure to track down all the connections and to inspect the various **synchan** fields. Fortunately, GENESIS has some commands that make this easier.

18.7.3 Utility Functions for Synapses

In general, we do not want to be concerned with the synapse number (hereafter called the synapse *index*, since it represents the index of an array of synapses) when setting up weights and delays. In addition, sometimes we may need to access other information about synapses, such as the source element of a given synapse or the total number of synapses. GENESIS provides several utility functions for this purpose, which we describe here. These functions will be particularly useful for debugging a simulation. Although the commands described above are much more efficient than the use of *for* loops for the establishment of network connections, it is easy to make a mistake in syntax and not get the connections that were intended.

The function *getsyncount* has the usage

```
    getsyncount [presynaptic-element] [postsynaptic-element]
```

This function is used to count synapse numbers. Either one or both options must be specified. The most common usage is to specify only the presynaptic element. In this case, it returns the number of SPIKE messages that are sent by that element. If only the postsynaptic element (e.g., a **synchan**) is present, it returns the number of synapses in that element. As we have seen in Chapter 15 and in Sec. 18.7.1, we could also obtain this result by using *getfield* to retrieve the *nsynapses* field of the postsynaptic element. If both arguments are present, it returns a count of the number of synapses in the postsynaptic element that receive SPIKE messages from the presynaptic element. (This will almost always return 0 or 1, as redundant connections between the same source and destination are more efficiently handled by increasing the synaptic weight of the connection.)

The function *getsynindex* has the usage

```
getsynindex <presynaptic-element> <postsynaptic-element> [-number n]
```

It is used to find the index of synapses between the given presynaptic and postsynaptic elements. The `-number` option will give the index of the *n*th synapse between the presynaptic and postsynaptic target. This option should rarely be necessary, since usually there is at most one synapse between a given presynaptic and postsynaptic element. If no matching synapse is found, a warning message is printed and the function returns -1.

We can use the *getsynindex* function in GENESIS to help us set the weights and delays of a synapse whose presynaptic element is known but whose index is not. For example we could type

```
int syn_num = {getsynindex /retina/recplane/rec[0]/input \
           /V1/horiz/soma[0]/exc_syn}
setfield /V1/horiz/soma[0]/exc_syn synapse[{syn_num}].weight 2.0 \
     synapse[{src}].delay 1e-4
```

The function *getsynsrc* has the usage

```
getsynsrc <postsynaptic-element> <index>
```

This function returns a string that is the path of the presynaptic element sending the SPIKE message to the synapse of the postsynaptic element with the given index.

The function *getsyndest* has the usage

```
getsyndest <presynaptic-element> <n> [-index]
```

This function returns a string that is the path of the postsynaptic element which receives the *n*th SPIKE message sent by the presynaptic element. The `-index` option returns the index of the synapse corresponding to this message. Alternatively, after having found the destination synapse, you may find its index by using *getsynindex*.

As an example of the use of the above functions, we can write a script function to give information about all the synapses in a particular **synchan**:

```
function synapse_info(path)
    str path, src
    int i
    float weight, delay
    floatformat %.3g
    for(i = 0; i < {getsyncount {path}}; i = i + 1)
        src    = {getsynsrc {path} {i}}
        weight = {getfield {path} synapse[{i}].weight}
        delay  = {getfield {path} synapse[{i}].delay}
        echo synapse[{i}]: \
            src = {src} weight = {weight} delay = {delay}
    end
end
```

This function also uses the *floatformat* command to set the format of the output to *%.3g*, which displays at most three significant figures and rounds the output to reasonable values. The GENESIS default is to print out 10 significant digits, which is often unnecessary.

We can use this function to check the ranges of the weights and delays of the synapses in the *Orient_tut* simulation. For example,

```
synapse_info /V1/horiz/soma[12]/exc_syn
```

gives the output

```
synapse[0]: src = /retina/recplane/rec[30]/input weight = 0.22 delay = 0.000165
synapse[1]: src = /retina/recplane/rec[31]/input weight = 0.22 delay = 0.000126
synapse[2]: src = /retina/recplane/rec[32]/input weight = 0.22 delay = 8.94e-05
synapse[3]: src = /retina/recplane/rec[33]/input weight = 0.22 delay = 5.66e-05
synapse[4]: src = /retina/recplane/rec[34]/input weight = 0.22 delay = 4e-05
(etc.)
```

18.8 Setting Up the Inputs

After the network is constructed, we will usually want a way to provide some input to the network. The details of this will generally be specific to your simulation. The inputs to the *Orient_tut* network are specified in the file *ret_input.g*. This file defines several functions whose purpose is to sweep a vertical or horizontal bar across the retinal cells. This is done by imposing an average firing rate on these cells by setting the *rate* field of the **randomspike** elements in */retina/recplane/rec[0-99]/input*. There are a number of ways to accomplish this in GENESIS. One possibility would be to set up a *for* loop that runs through all the elements and sets the rate to a high or low value depending on the position of the cell in space. Another approach, which is used in the file, is to define a function called *do_autosweep* which is invoked on each step of the simulation. Yet another approach would be to define a GENESIS extended object that performs the same function. More information is given in the *README* file in the *Orient_tut* directory and in the comments in the *ret_input.g* file.

18.9 Summary

In this chapter we have shown you the GENESIS commands for creating groups of cells and connecting them in networks. We have discussed how to set up synapses individually and also have described various commands that allow you to set up groups of synapses simultaneously. These commands enable you to configure the connectivity patterns, synaptic weights, and synaptic delays of a network in very flexible ways. We have used the *Orient_tut* simulation as an example to show how these commands are used in a real simulation. The *Orient_tut* simulation also implements a graphical user interface that allows the user to look at various aspects of the network as it is being simulated, including the firing patterns of the input (retinal) cells, the membrane potentials of the output (V1) cells, and the connection patterns in the network. The objects and commands used to set up the interface are described in Chapter 22.

18.10 Exercises

1. Verify that with *CABLE_VEL* = 1, the delays for the connections from rec[54]/axon to the targets in the V1 horiz layer of somas 11, 12 and 13 are correct (i.e., are equal to the radial distances).

2. Modify the "*synapse_info*" function, using the synaptic utility functions described previously, to generate the pathname, synapse index, weight, and delay values of all synapses projecting from a given **randomspike** element, i.e., all synapses receiving SPIKE messages from that element. You might want to save this function for later use.

3. Look at the file *ret_V1.g*. How is the orientation-selectivity conferred on the network? Can you improve the selectivity just by changing some parameters of the commands?

The cells in the two V1 planes contain some elements that are not used in the simulation. The spike generators of the cells connect to nothing and there is no input to the inhibitory channels, *inh_syn*. As an exercise in using the commands discussed above,

4. Modify the *ret_V1.g* script by adding synapses between the retinal cells and the inhibitory synapses of the V1 cells in order to improve the quality of the orientation selectivity.

5. Modify the *ret_V1.g* script to generate connections between V1 cells. Can you use the feedback connections to further improve the quality of the orientation selectivity?