Chapter 11

Modeling networks

NEURON was initially developed to handle neuronal models in which complex membrane properties and extended geometry play important roles [Hines, 1989 #61; , 1993 #104; , 1995 #22]. However, as the research interests of experimental and theoretical neuroscientists evolved, NEURON has been revised accordingly. Since the early 1990s it has been used to model networks of biological neurons (e.g. [Destexhe, 1993 #166][Lytton, 1997 #260][Sohal, 2000 #487]). This work stimulated the development of powerful strategies that increase the convenience and efficiency of creating, managing, and exercising such models [Destexhe, 1994 #267][Lytton, 1996 #206][Hines, 2000 #323]. Increasing research activity on networks of spiking neurons (e.g. [Maas, 1999 #610][Riecke, 1997 #556]) prompted further enhancements to NEURON, such as inclusion of an event delivery system and development of the `NetCon` (network connection) class (see **Chapter 10**).

Consequently, since the latter 1990s, NEURON has been capable of efficient simulations of networks that may include biophysical model neurons and/or artificial spiking neurons. Biophysical model neurons are built around representations of the biophysical mechanisms that are involved in neuronal function, so they have sections, density mechanisms, and synapses (see **Chapter 5**). A synapse onto a biophysical model cell is a point process with a `NET_RECEIVE` block that affects membrane current (e.g. `ExpSyn`) or a second messenger (see **Chapter 10**). The membrane potential of a biophysical model cell is governed by complex, interacting nonlinear mechanisms, and spatial nonuniformities may also be present, so numerical integration is required to advance the solution in time.

What could be more oxymoronic than "real model neuron"?

As we discussed in **Chapter 10**, artificial spiking neurons are actually point processes with a `NET_RECEIVE` block that calls `net_event()` (e.g. `IntFire1`). An artificial neuron has a "membrane state variable" with very simple dynamics, and space is not a factor, so the time course of the integration state is known analytically and it is relatively easy to compute when the next spike will occur. Since artificial neurons do not need numerical integration, they can be used in discrete event simulations that run several orders of magnitude faster than simulations involving biophysical model cells. Their simplicity also makes it very easy to work with them. Consequently, artificial spiking neurons are particularly useful for prototyping network models.

In this chapter we present an example of how to build network models by combining the strengths of the GUI and `hoc` programming. The GUI tools for creating and managing network models are most appropriate for exploratory simulations of small nets. Once you have set up and tested a small network with the GUI, a click of a button creates

a hoc file that contains reusable cell class definitions and procedures. This eliminates the laborious, error-prone task of writing "boilerplate" code. Instead, you can just combine NEURON's automatically generated code with your own hoc programming to quickly construct large scale nets with complex architectures. Of course, network models can be constructed entirely by writing hoc code, and NEURON's WWW site contains links to a tutorial for doing just that (Gillies and Sterratt, 2004). However, by taking advantage of GUI shortcuts, you'll save valuable time that can be used to do more research with your models.

Building a simple network with the GUI

Regardless of whether you use the GUI or write hoc code, creating and using a network model involves these basic steps:

1. Define the types of cells.
2. Create each cell in the network.
3. Connect the cells.
4. Set up instrumentation for adjusting model parameters and recording and/or displaying simulation results.
5. Set up controls for running simulations.

We will demonstrate this process by constructing a network model that can be used to examine the contributions of synaptic, cellular, and network properties to the emergence of synchronous and/or correlated firing patterns.

Conceptual model

The conceptual model is a fully connected network, i.e. each cell projects to all other cells, but not to itself (Fig. 11.1 left). All conduction delays and synaptic latencies are identical.

The cells are spontaneously active integrate and fire neurons, similar to those that we discussed in **Chapter 10**. All cells have the same time constant and firing threshold, but in isolation each has its own natural interspike interval (ISI), and the ISIs of the population are distributed uniformly over a fixed range (Fig. 11.1 right).

Figure 11.2 illustrates the dynamics of these cells. Each spike is followed by a "post-spike" hyperpolarization of the membrane state variable *m*, which then decays monoexponentially toward a suprathreshold level. When *m* reaches threshold (1), it triggers another spike and the cycle repeats. A synaptic input hyperpolarizes the cell and prolongs the ISI in which it occurred, shifting subsequent spikes to later times. Each input produces the same hyperpolarization of *m*, regardless of where in the ISI it falls. Even so, the shift of the spike train depends on the timing of the input. If it arrives shortly after a spike, the additional hyperpolarization decays quickly and the spike train shifts by only a

small amount (Fig. 11.2 left). An input that arrives late in the ISI can cause a much larger shift in the subsequent spike train (Fig. 11.2 right).

Our task is to create a model that will allow us to examine how synaptic weight, membrane time constant and natural firing frequency, number of cells and conduction latency interact to produce synchronized or correlated spiking in this network.

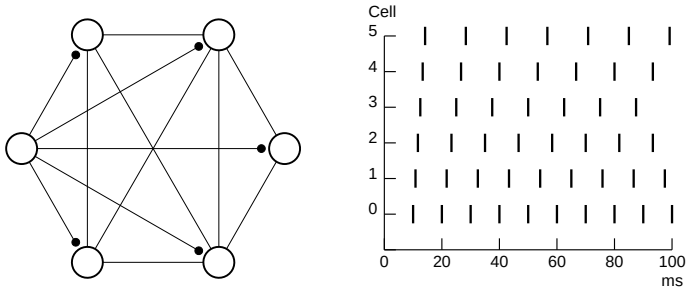


Figure 11.1. Left: An example of a fully connected net. Thin lines indicate reciprocal connections between each pair of cells, and thick lines mark projections from one cell to its targets. Right: When disconnected from each other, every cell has its own natural firing frequency.

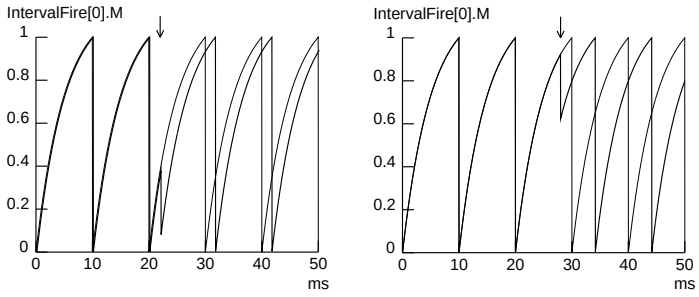


Figure 11.2. Time course of the membrane state variable *m* in the absence (thin traces) and presence (thick traces) of an inhibitory input. Notice that *m* follows a monoexponential "depolarizing" time course which carries it toward a suprathreshold level. When *m* reaches 1, a spike is triggered and *m* is reset to 0 ("post-spike hyperpolarization"). An inhibitory synaptic event causes the same hyperpolarizing shift of *m* no matter where in the ISI it arrives, but its effect on later spike times depends on its relative position in the ISI. Left: Inhibitory events that occur early in the ISI decay quickly, so following spikes are shifted to slightly later times. Right: An inhibitory event that occurs late in the ISI has a longer lasting effect and causes a greater delay of the subsequent spike train.

Adding a new artificial spiking cell to NEURON

Before we start to build this network, we need to add a new kind of artificial spiking cell to NEURON. Our model will use cells whose membrane state variable *m* is governed by the equation

$$\tau \frac{dm}{dt} + m = m_{\infty} \qquad \text{Eq. 11.3}$$

where $m_\infty > 1$ and is set to a value that produces spontaneous firing with the desired ISI.

An input event with weight *w* adds instantaneously to *m*, and if *m* reaches or exceeds the threshold value of 1, the cell "fires," producing an output event and returning *m* to 0. We will call this the `IntervalFire` model, and the NMODL code for it is shown in Listing 11.1. `IntervalFire` has essentially the same dynamics as `IntFire1`, but because its membrane state relaxes toward a suprathreshold value, it uses a `firetime()` function to compute the time of the next spike (see discussions of `IntFire1` and `IntFire2` in **Chapter 10**).

```
NEURON {
  ARTIFICIAL_CELL IntervalFire
  RANGE tau, m, invl
}

PARAMETER {
  tau = 5 (ms)   <1e-9,1e9>
  invl = 10 (ms) <1e-9,1e9>
}

ASSIGNED {
  m
  minf
  t0(ms)
}

INITIAL {
  minf = 1/(1 - exp(-invl/tau)) : so natural spike interval is invl
  m = 0
  t0 = t
  net_send(firetime(), 1)
}

NET_RECEIVE (w) {
  m = M()
  t0 = t
  if (flag == 0) {
    m = m + w
    if (m > 1) {
      m = 0
      net_event(t)
    }
    net_move(t+firetime())
  } else {
    net_event(t)
    m = 0
    net_send(firetime(), 1)
  }
}

FUNCTION firetime()(ms) { : m < 1 and minf > 1
  firetime = tau*log((minf-m)/(minf - 1))
}
```



```
FUNCTION M() {
  M = minf + (m - minf)*exp(-(t - t0)/tau)
}
```

Listing 11.1. NMODL implementation of `IntervalFire`. Figures 11.1 (right) and 11.3 illustrate its operation.

Creating a prototype net with the GUI

After we compile the code in Listing 11.1 (see **Chapter 9**), when we launch `nrngui` these lines should appear at the end of NEURON's startup message

```
Additional mechanisms from files
 invlfire.mod
```

to reassure us that what was defined in `invlfire.mod`--i.e. the `IntervalFire` cell class--is now available. We are ready to use the GUI to build and test a prototype net.

1. Define the types of cells

This involves using the existing cell classes to create the types of cells that we will employ in our network. Our network contains artificial spiking cells, so we need an ArtCellGUI tool, which we get by clicking on Build / NetWork Cell / Artificial Cell in the NEURON Main Menu toolbar (Fig. 11.3).

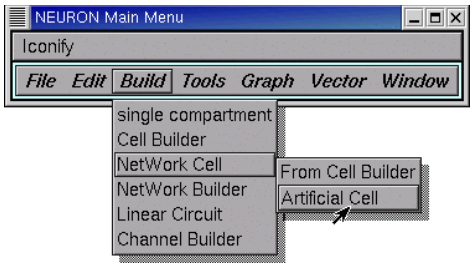


Figure 11.3. Using the NEURON Main Menu to bring up an ArtCellGUI tool.

The gray area in the lower left corner of the ArtCellGUI tool displays a list of the types of artificial spiking cells that will be available to the NetWork Builder. It starts out empty because we haven't done anything yet (Fig. 11.4). To remedy this, click on New and scroll down to select IntervalFire (Fig. 11.5 left), and then release the mouse button. The Artificial Cell types list now contains a new item called IntervalFire, and the right panel of the ArtCellGUI tool shows the user-settable parameters for this cell type (Fig. 11.5 right). These default values are fine for our initial exploratory simulations, so we'll leave them as is.

However, there is one small change that will make it easier to use the NetWork Builder: IntervalFire is a big word, and the NetWork Builder's canvas is relatively small. So let's give our cell type a short, unique name, like IF (see Figures 11.6 and 7).

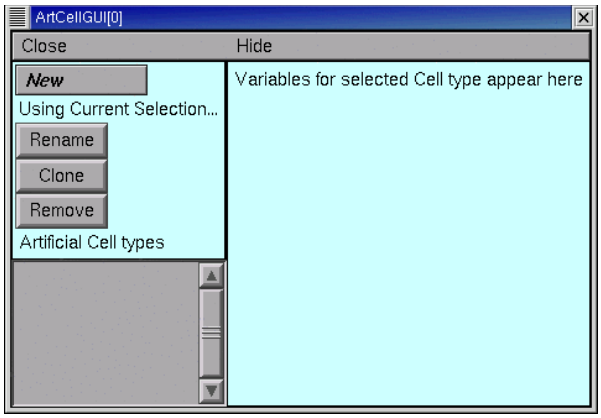


Figure 11.4. The ArtCellGUI tool starts with an empty Artificial Cell types list.

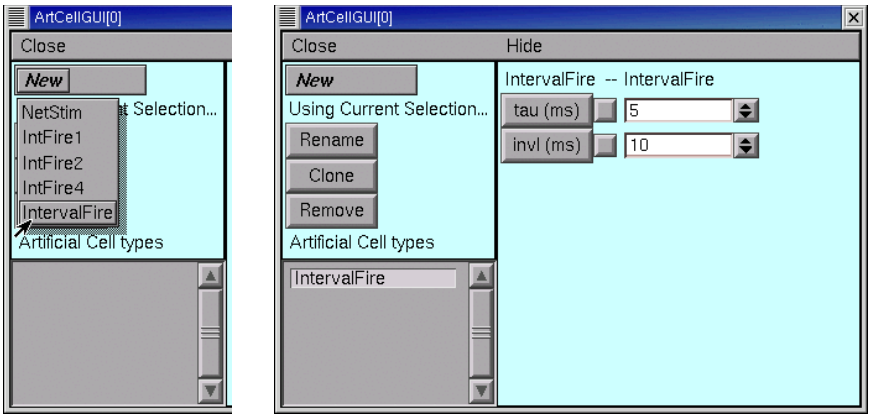
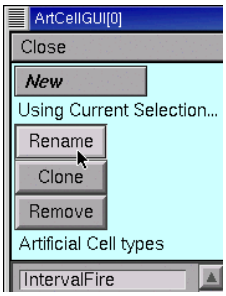


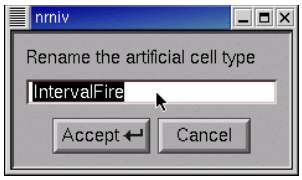
Figure 11.5. Click on New / IntervalFire to add it to the Artificial Cell types list.

Figure 11.6. Changing the name of one of the Artificial Cell types.

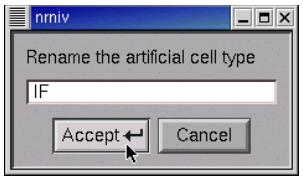
To change the name of one of the Artificial Cell types, select it (if it isn't already selected) and then click on the Rename button.



This pops up a window with a string editor field. Click in the field . . .



. . . change the name to IF, and then click the Accept button.



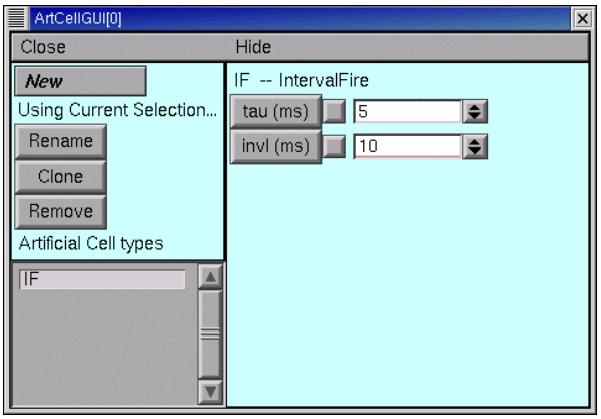


Figure 11.7. The ArtCellGUI tool after renaming the cell type. The right panel shows that IF is based on the IntervalFire class.

Now that we have configured the ArtCellGUI tool, it would be a good idea to save everything to a session file with NEURON Main Menu / File / save session (also see Fig. 1.23 and **Save the model cell** in **Chapter 1**). If you like, you may hide the ArtCellGUI tool by clicking on Hide just above the drag bar, but don't close it--the NetWork Builder will need it to exist.

2. Create each cell in the network

Having specified the cell types that will be used in the network, we are ready to use the NetWork Builder to create each cell in the network and connect them to each other. Actually, we'll just be creating the specification of each cell in the net; no cells are really created and there is no network until the Create button in the NetWork Builder is ON.

To get a NetWork Builder, click on NEURON Main Menu / Build / NetWork Builder (Fig. 11.8).

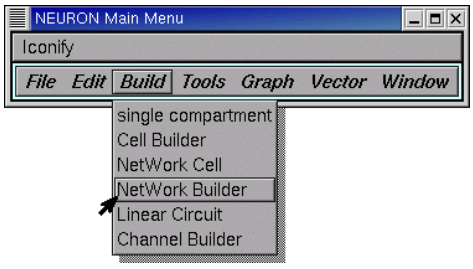


Figure 11.8. Bringing up a NetWork Builder.

The NetWork Builder's drag bar reveals that this tool is an instance of the `NetGui` class (see Fig. 11.9).

The right panel of a NetWork Builder is a canvas for laying out the network. The "palette" for this canvas is a menu of the cell types that were created with the ArtCellGUI tool. These names appear along the upper left edge of the canvas (for this example, a limited palette indeed: IF is the only cell type). Context-dependent hints are displayed at the top of the canvas.

The left panel of a NetWork Builder contains a set of buttons that control its operation. When a NetWork Builder is first created, its Locate radio button is automatically ON. This means that the NetWork Builder is ready for us to create new cells. We do this by merely following the hint (Fig. 11.10). Notice that the cell names are generated by concatenating the base name (name of the cell type) with a number that starts at 0 and increases by 1 for each new cell. We'll say more about this below in **7. A word about cell names**.

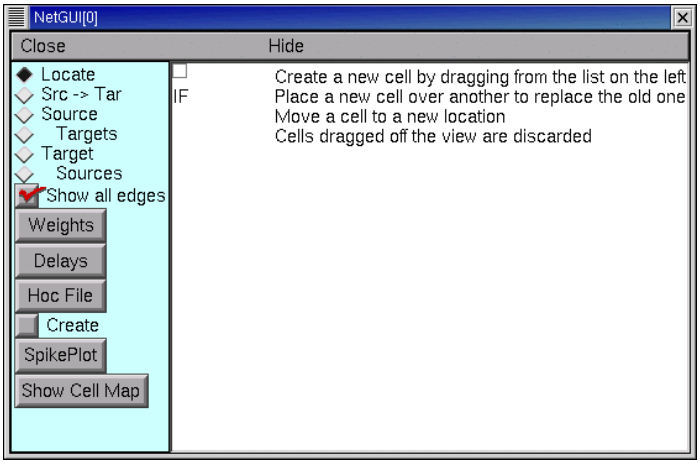
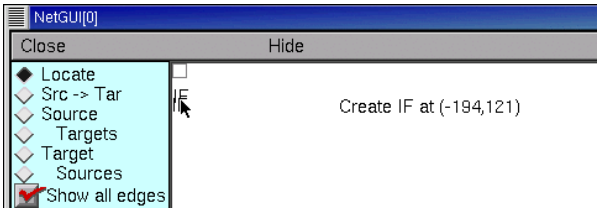


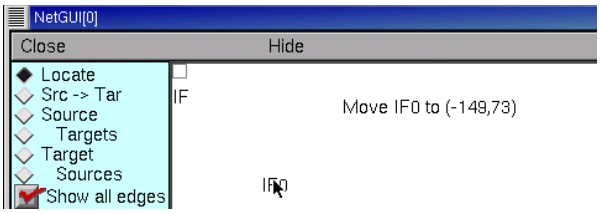
Figure 11.9. A new NetWork Builder.

Figure 11.10. Creating new cells in the NetWork Builder.

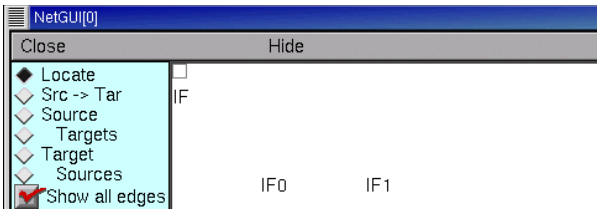
To create a new cell, click on one of the items in the palette (in this example, the only item is IF) and hold the mouse button down . . .



while dragging the new cell to a convenient location on the canvas. Release the mouse button, and you will see a new cell labeled IF0.



After you create a second IF cell, the NetWork Builder should look like this.



If you released the mouse button while the cursor was still close to one of the palette items, the new cell will be hard to select since palette item selection takes precedence over selection of a cell. If this happens, just select Translate in the canvas's secondary menu (the canvas is just a modified graph!) and then left click on the canvas and drag it to the right (if you have a three button mouse, or a mouse with a scroll wheel, don't bother with the canvas's menu--just click on the middle button or scroll wheel and drag the canvas). This will pull the cell out from under the palette items, which never move from their position along the left edge of the canvas. Finally, click on one of the radio buttons (Locate, Src -> Tar, etc.) and continue working with the NetWork Builder.

3. Connect the cells

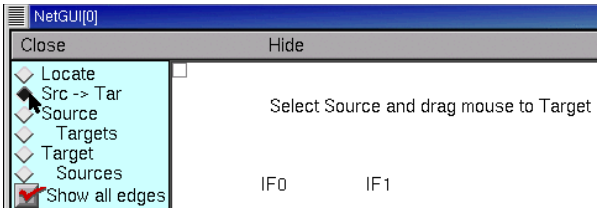
Connecting the cells entails two closely related tasks: setting up the network's architecture, and specifying the delays and weights of these connections.

Setting up network architecture

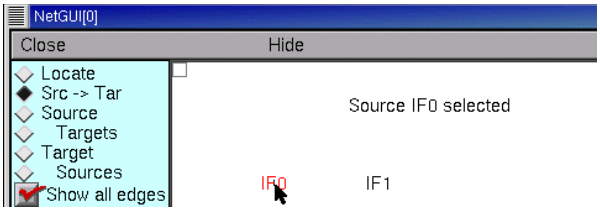
To set up the architecture, we click on the Src -> Tar radio button, read the new hint in the canvas, and do what it says (Fig. 11.11).

Figure 11.11. Setting up network architecture.

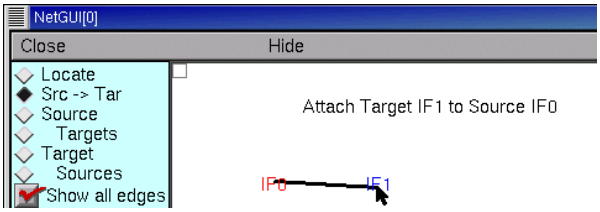
Clicking on the Src -> Tar button brings out a new hint.



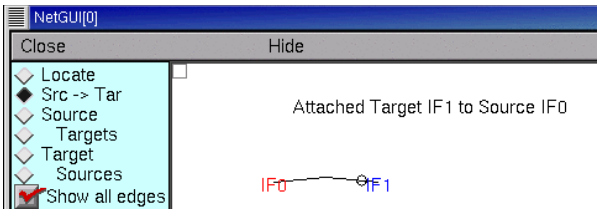
So we click on IF0 and hold the mouse button down while dragging the cursor toward IF1. A thin "rubber band" line will stretch from IF0 to the cursor.



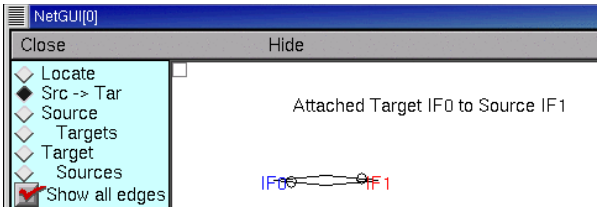
When the cursor is on top of IF1, the rubber band becomes a thick black line, and the hint changes to the message shown here.



To complete the attachment, we just release the mouse button. The projection ("edge") from IF0 to IF1 will appear as a thin line with a slight bend near its midpoint. The O marks the target end of this connection.



Making the reciprocal connection requires only that we click on IF1, drag to IF0, and release the mouse button.



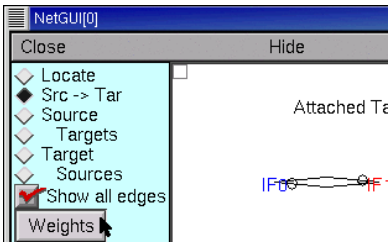
This is a good time to save everything to a session file.

Specifying delays and weights

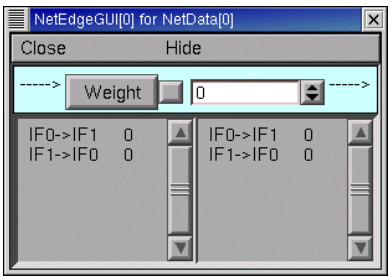
The default initial value of all synaptic weights is 0, i.e. a presynaptic cell will have no effect on its postsynaptic targets. The NetWork Builder has a special tool that we can use to change the weights to what we want (Fig. 11.12).

Figure 11.12. Setting the synaptic weights.

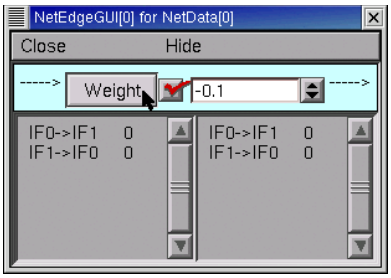
Clicking on the Weights button in the NetWork Builder . . .



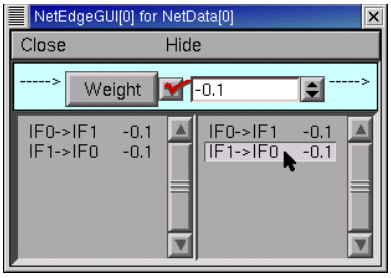
. . . brings up a tool for specifying synaptic weights. The top of this tool has a numeric field with its associated spinner and button (labeled Weight). The value in the numeric field can be set in the usual ways (direct entry, using the spinner, etc.), but note the arrows, which suggest other possibilities.



The bottom of the weights tool contains two panels that list the weights of all synaptic connections (aka "edges" in network theory). Clicking on a connection in the left list copies from the connection to the numeric field, and clicking on a connection in the right list copies from the numeric field to the connection.



For this example, let's make both synapses have a weight of -0.1 (mild inhibition). First we change Weight to -0.1 . . .



. . . and then we click on IF0->IF1 and IF1->IF0 in the right panel. We're finished when the weights tool looks like this.

Now we can close this window. If we need it again, clicking on the NetWork Builder's Weights button will bring it back.

All delays are 1 ms by default, which is fine for our purposes. If we wanted to change this to something else, we would click on the NetWork Builder's Delays button (see Fig. 11.9) to bring up a tool for setting delays. The delay tool works just like the weight tool.

At this point, the ArtCellGUI tool plus the NetWork Builder together constitute a complete specification of our network model. We should definitely save another session file before doing anything else!

Now we have a decision to make. We could use the NetWork Builder to create a hoc file that, when executed, would create an instance of our network model. A better choice is to use the GUI to test our model. If there are any problems with what we have done so far, this is a good time to find out and make the necessary corrections.

However, before we can run tests, there must first be something to test. We have a network specification, but no network. As we pointed out earlier in **2. Create each cell in the network**, the network doesn't really exist yet. Clicking on the Create button in the NetWork Builder fixes that (Fig. 11.13).

4. Set up instrumentation

We want to see what our network does, and to explore how its behavior is affected by model parameters. Clicking on the SpikePlot button in the NetWork Builder brings up a tool that will show the input and output spike trains (Fig. 11.14).

We already know how to adjust model parameters. With the NetWork Builder we can change synaptic weights and delays, and the IF cells' properties can be changed with the ArtCellGUI tool. Suddenly, we realize that both IF cells will have the same time constant and firing rate. No problem--our goal is to combine the strengths of the GUI and hoc. We will take care of this later, by combining the hoc code that the NetWork Builder generates with our own hoc code. Using a few lines of hoc, we can easily assign unique firing

rates across the entire population of IF cells. And if we insisted on sticking with GUI tools to the bitter end, we could just bring up a PointProcessGroupManager (NEURON Main Menu / Tools / Point Processes / Managers / Point Group), which would allow us to control the attributes of each cell in our network individually.

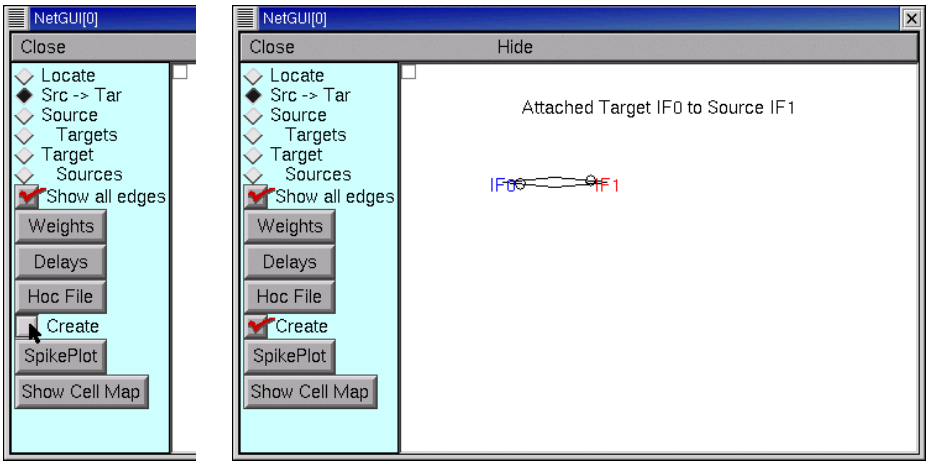


Figure 11.13. Left: Toggling the Create button ON causes the network specification to be executed. Right: Once Create is ON, the representation of the network is available for NEURON's computational engine to use in a simulation.

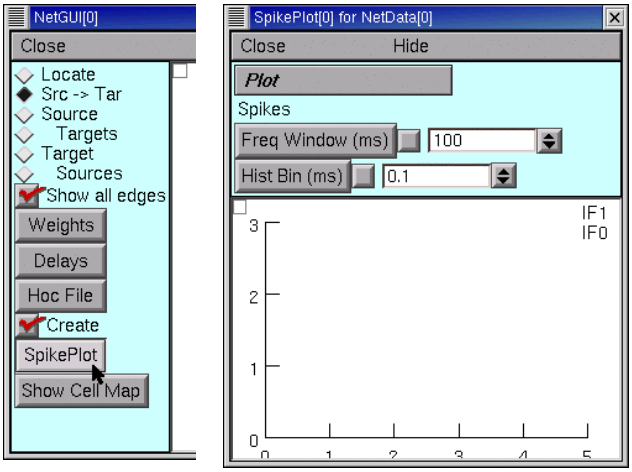


Figure 11.14. The NetWork Builder's SpikePlot button (left) brings up a tool for displaying and analyzing spike trains (right).

5. Set up controls for running simulations

At a minimum, we need a RunControl panel (NEURON Main Menu / Tools / RunControl, as shown in **5. Set up controls for running the simulation** in **Chapter 1**). Also, since our network contains only artificial spiking neurons, we can use adaptive

integration to achieve extremely fast, discrete event simulations. We'll need a VariableTimeStep panel (NEURON Main Menu / Tools / VariableStepControl (Fig. 11.15)), which makes it easy to choose between fixed time step or adaptive integration (Fig. 11.16).

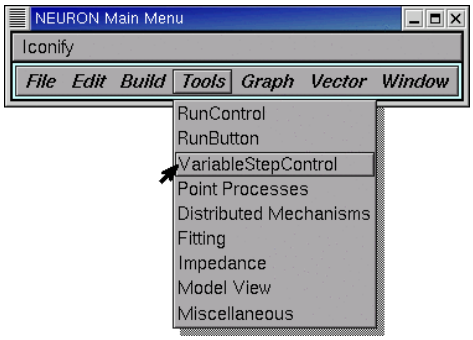


Figure 11.15. Bringing up a VariableTimeStep panel.

Figure 11.16. Toggling adaptive integration ON and OFF.

The VariableTimeStep panel's Use variable dt checkbox is empty, which means that adaptive integration is off.

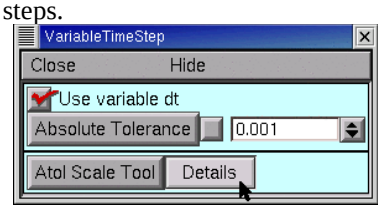
To turn adaptive integration ON, we click on the Use variable dt checkbox.

The check mark in the Use variable dt checkbox tells us that adaptive integration is ON. Clicking on this checkbox again will turn it back OFF so that fixed time steps are used.

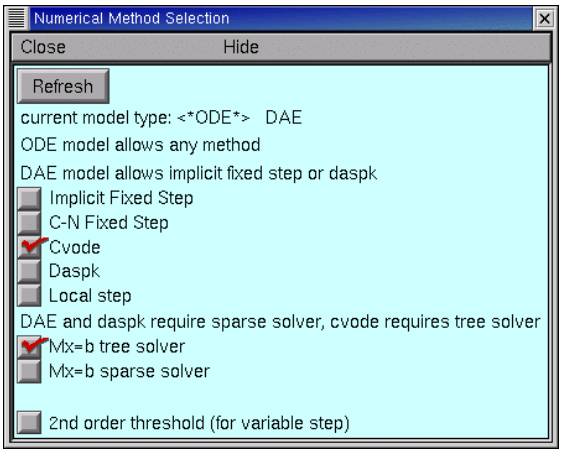
Adaptive integration can use either global or local time steps, each of which has its own particular strengths and weaknesses (see **Adaptive integrators** in **Chapter 7**). The VariableTimeStep panel's default setting is to use global time steps, which is best for models of single cells or perfectly synchronous networks. Our toy network has two identical cells connected by identical synapses, so we would expect them to fire synchronously. However, when we build our net with hoc code, the cells will all have different natural firing frequencies, and who can tell in advance that they will achieve perfect synchrony? Besides, this is a tutorial, so let's use local time steps (Fig. 11.17).

Figure 11.17. Toggling between global and local time steps.

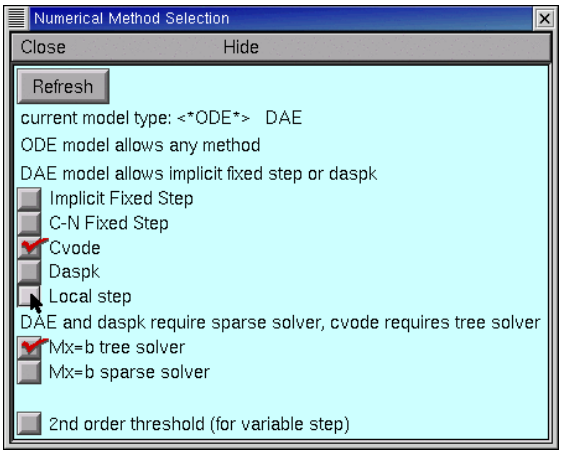
To specify whether to use global or local time steps, we first click on the VariableTimeStep panel's Details button.



We are concerned with the Local step checkbox, which is empty. To activate the use of local variable time steps . . .

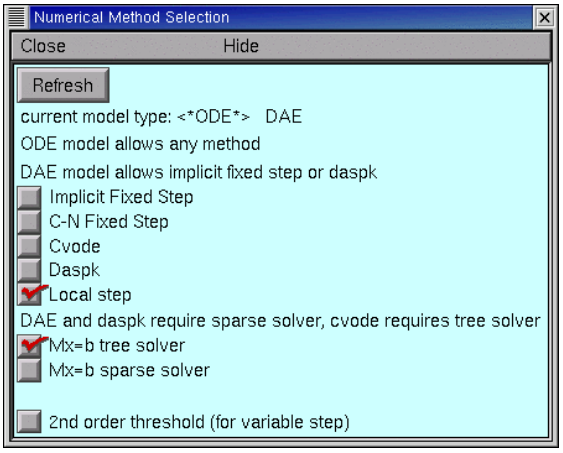


. . . we just click on the Local step checkbox . . .



. . . and now each cell in our network will advance with its own time step. If we want to restore global time steps, we can just click on the Cvode button.

Now we can close this panel; should we need it again, we only have to click on the VariableTimeStep panel's Details button.



After rearrangement, the various windows we have created should look something like Fig. 11.18. The tools we used to specify the network are on the left, simulation controls are in the middle, and the display of simulation results is on the right. Quick, save it to a session file!

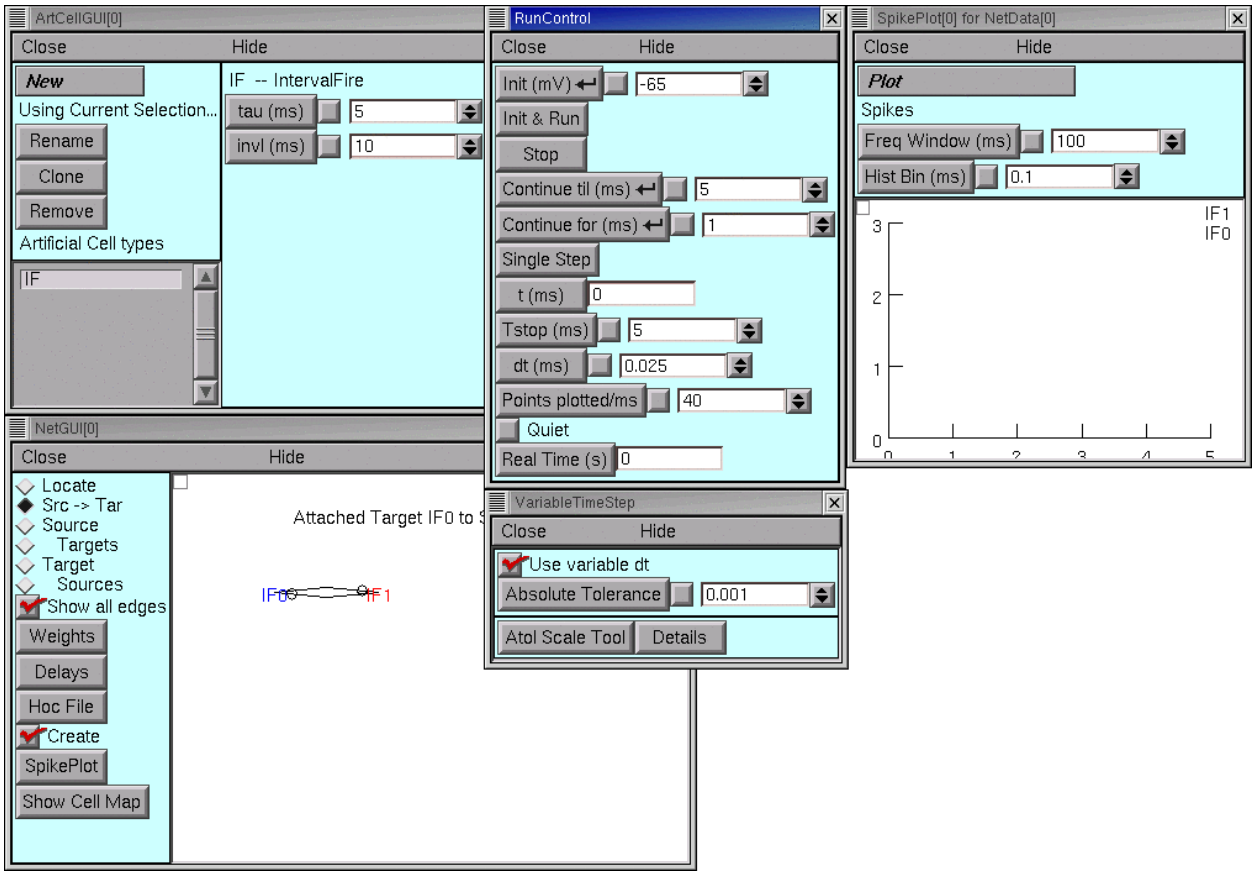


Figure 11.18. The completed model with controls for running simulations and displaying results.

6. Run a simulation

This is almost too easy. Clicking on Init & Run in the RunControl panel, we see--nothing! Well, almost nothing. The t field in the RunControl panel shows us that time advanced from 0 to 5 ms, but there were no spikes. A glance at the ArtCellGUI tool tells us why: invl is 5 ms, which means that our cells won't fire their first spikes for another 5 ms. Let's change Tstop to 200 ms so we'll get a lot of spikes, and try again. This time we're successful (Fig. 11.19).

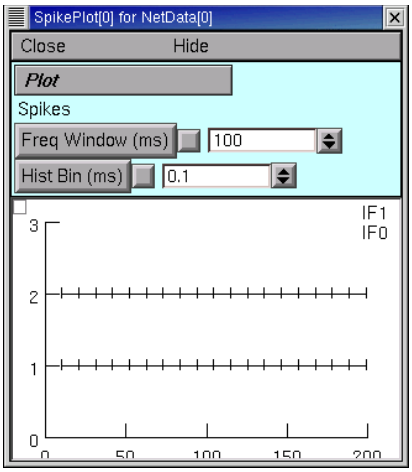


Figure 11.19. The SpikePlot shows the spike trains generated by the cells in our network model. Note that rasters correspond to cell names from top to bottom, and that the raster for cell i is plotted along the line $y = i + 1$.

7. Caveats and other comments

Changing the properties of an existing network

As we saw in this example, the ArtCellGUI tool is used to specify what artificial spiking cell types are available to a Network Builder. The same ArtCellGUI tool can be used to adjust the parameters of those cells, and such changes take effect immediately, even if the network already exists (i.e. even if the Network Builder's Create button is ON).

The NetReadyCellGUI tool (NEURON Main Menu / Build / Network Cell / From Cell Builder) is used to configure biophysical model cell types for use with a Network Builder. In fact, we would use a separate NetReadyCellGUI instance for each different type of biophysical model cell we wanted to use in the net. The NetReadyCellGUI tool has its own CellBuilder for specifying topology, geometry, and biophysical properties, plus a SynapseTypes tool for adding synaptic mechanisms to the cell (see the tutorial at `http://www.neuron.yale.edu/neuron/docs/netbuild/main.html`). However, changes made with a NetReadyCellGUI tool do *not* affect an existing network; instead, it is necessary to save a session file, exit NEURON, restart and reload the session file.

What about changes to the network itself? Any changes whatsoever can be made in the Network Builder, as long as its Create button is OFF. Once it is ON, some changes are possible (e.g. adding new cells and synaptic connections to an existing network), but additional actions may be required (a pre-existing SpikePlot will not show spike trains from new cells), and there is a risk of introducing a mismatch between one's conceptual model and what is actually in the computer. The best policy is to toggle Create OFF (see Fig. 11.20), make whatever changes are needed, save everything to a session file, exit NEURON, and then restart and load the new session file.

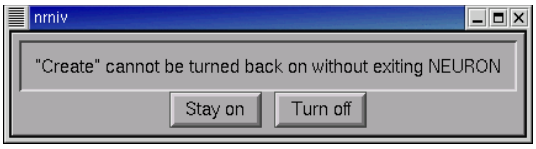


Figure 11.20. Trying to turn Create OFF brings up this window, which offers the opportunity to change one's mind. Select Turn off if it is necessary to make substantial changes to an existing network in the NetWork Builder.

A word about cell names

As we mentioned above in **2. Create each cell in the network**, the cell names that appear in the NetWork Builder are generated automatically by concatenating the name of the cell type with a sequence of numbers that starts at 0 and increases by 1 for each additional cell. But that's only part of the story. These are really only short "nicknames," a stratagem for preventing the NetWork Builder and its associated tools from being cluttered with long character strings.

This is fine as long as the NetWork Builder does everything we want. But suppose we need to use one of NEURON's other GUI tools, or we have to write some hoc code that refers to one of our model's cells? For example, we might have a network that includes a biophysical model neuron, and we want to see the time course of somatic membrane potential. In that case, it is absolutely necessary to know the actual cell names.

That's where the NetWork Builder's Cell Map comes in. Clicking on Show Cell Map brings up a small window that often needs to be widened by clicking and dragging on its left or right margin (Fig. 11.21). Now we realize that, when we used the ArtCellGUI tool to create an IF cell "type," we were actually specifying a new cell class whose name is a concatenation of our "type" (IF), an underscore character, and the name of the root class (the name of the class that we based IF on, which was IntervalFire).

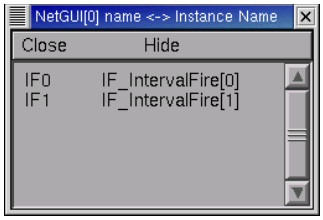


Figure 11.21. The Cell Map for our toy network. See text for details.

Combining the GUI and programming

Creating a hoc file from the NetWork Builder

Having tested our prototype model, we are now ready to write a hoc file that can be mined for reusable code. Clicking on the Hoc File button in the NetWork Builder brings up a tool that looks much like what we used to specify file name and location when

saving a session file. Once we're satisfied with our choices, clicking on this tool's "Open" button writes the hoc file (yes, the button should say Close). This file, which we will call prototype.hoc, is presented in Listing 11.2, and executing it would recreate the toy network that we just built with the NetWork Builder.

```
// NetGUI default section. Artificial cells, if any, are located here.
  create acell_home_
  access acell_home_

//Network cell templates
//Artificial cells
//   IF_IntervalFire


begintemplate IF_IntervalFire
public pp, connect2target, x, y, z, position, is_art
external acell_home_
objref pp
proc init() {
  acell_home_ pp = new IntervalFire(.5)
}
func is_art() { return 1 }
proc connect2target() { $o2 = new NetCon(pp, $o1) }
proc position(){x=$1  y=$2  z=$3}
endtemplate IF_IntervalFire

//Network specification interface

objref cells, nclist, netcon
{cells = new List()  nclist = new List()}

func cell_append() {cells.append($o1)  $o1.position($2,$3,$4)
   return cells.count - 1
}

func nc_append() {//srcindex, tarcelindex, synindex
  if ($3 >= 0) {
    cells.object($1).connect2target(cells.object($2).synlist.object($3), \
                                    netcon)
    netcon.weight = $4   netcon.delay = $5
  }else{
    cells.object($1).connect2target(cells.object($2).pp,netcon)
    netcon.weight = $4   netcon.delay = $5
  }
  nclist.append(netcon)
  return nclist.count - 1
}

//Network instantiation

  /* IF0 */  cell_append(new IF_IntervalFire(),    -149,   73, 0)
  /* IF1 */  cell_append(new IF_IntervalFire(),    -67,    73, 0)
  /* IF1 -> IF0    */  nc_append(1, 0, -1,  -0.1,1)
  /* IF0 -> IF1    */  nc_append(0, 1, -1,  -0.1,1)
```

Listing 11.2. Clicking on the Hoc File button in the NetWork Builder produces a file which we have called prototype.hoc.

A quick glance over the entire listing reveals that `prototype.hoc` is organized into several parts, which are introduced by one or more lines of descriptive comments. Let us consider each of these in turn, to see how it works and think about what we might reuse to make a network of any size we like.

NetGUI default section

The first part of the file creates `acell_home_` and make this the default section. What is a *section* doing in a model that contains artificial spiking cells? Remember that artificial spiking cells are basically point processes (see **Artificial spiking cells** in **Chapter 10**), and just like other point processes, they must be attached to a section. Suddenly the meaning of the comment `Artificial cells, if any, are located here` becomes clear: `acell_home_` is merely a "host" for artificial spiking cells. It has no biophysical mechanisms of its own, so it introduces negligible computational overhead.

Network cell templates

The NetWork Builder and its associated tools make extensive use of object-oriented programming. Each cell in the network is an instance of a cell class, and this is where the templates that declare these classes are located (templates and other aspects of object-oriented programming in NEURON are discussed in **Chapter 13**).

The comments that precede the templates contain a list of the cell class names. Our toy network uses only one cell class, so `prototype.hoc` contains only one template, which defines the `IF_IntervalFire` class. When biophysical model cells are present, they are declared first. Thus, if we had a NetWork Builder whose palette contained a biophysical model cell type called pyr, and an artificial spiking cell type S that was derived from the `NetStim` class, the corresponding cell classes would be called `pyr_Cell` and `S_NetStim`, and the header in the exported hoc file would read

```
//Network cell templates
//   pyr_Cell
//Artificial cells
//   S_NetStim
```

Functions and procedures with the same names as those contained in the `IF_IntervalFire` template will be found in every cell class used by a NetWork Builder (although some of their internal details may differ). The first of these is `init()`, which is executed automatically whenever a new instance of the `IF_IntervalFire` class is created. This in turn creates a new instance of the `IntervalFire` class that will be associated with the `acell_home_` section. As an aside, we should mention that this is an example of how the functionality of a basic object class can be enhanced by wrapping it inside a template in order to define a new class with additional features, i.e. an example of emulating inheritance in `hoc` (see **Polymorphism and Inheritance** in **Chapter 13**).

The remaining `func`s and `proc`s are public so they can be called from outside the template. If we ever need to determine which elements in a network are artificial spiking cells and which are biophysical model cells, `is_art()` is clearly the way to do it. The next is `connect2target()`, which looks useful for setting up network connections, but

it turns out that the hoc code we write ourselves won't call this directly (see **Network specification interface below**). The last is position() which can be used to specify unique xyz coordinates for each instance of this cell. The coordinates themselves are public (accessible from outside the template--see **Chapter 13** for more about accessing variables, funcs and procs declared in a template). Position may seem an arcane attribute for an artificial spiking neuron, but it is helpful for algorithmically creating networks in which connectivity or synaptic weight are functions of location or distance between cells.

Network specification interface

These are the variables and functions that we will actually call from our own hoc code. These are intended to offer us a uniform, compact and convenient syntax for setting up our own network. That is, they serve as a "programming interface" between the code we write and the lower level code that accomplishes our ultimate aims.

The purpose of the first two lines in this part of prototype.hoc is evident if we keep in mind that the NetWork Builder implements a network model with objects, some of which represent cells while others represent the connections between them. The List class is the programmer's workhorse for managing collections of objects, so it is reasonable that our network model will be packaged into two Lists called cells and nclist.

The functions that add new elements to these Lists are cell_append() and nc_append(), respectively. The first argument to cell_append() is an objref that points to a new cell that is to be added to the list, and the remaining arguments are the xyz coordinates that are to be assigned to that cell. The nc_append() function uses an if . . . else to deal properly with either biophysical model cells or artificial spiking cells. In either case, its first two arguments are integers that indicate which elements in cells are the objrefs that correspond to the pre- and postsynaptic cells, and the last two arguments are the synaptic weight and delay. If the postsynaptic cell is a biophysical model cell, one or more synaptic mechanisms will have been attached to it (see the tutorial at http://www.neuron.yale.edu/neuron/docs/netbuild/main.html). In this case, the third argument to nc_append() is a nonnegative integer that specifies which synaptic mechanism is to be the target of the new NetCon. If instead the postsynaptic cell is an artificial spiking cell, the argument is just -1.

Network instantiation

So far everything has been quite generic, in the sense that we can use it to create cells and assemble them into whatever network architecture we desire. In other words, the code up to this point is exactly the reusable code that we needed. The statements in the "network instantiation" group are just a concrete example of how to use it to spawn a particular number of cells and link them with a specific network of connections. Let's make a copy of prototype.hoc, call it netdefs.hoc, and then insert // at the beginning of each of last four lines of netdefs.hoc so they persist as a reminder of how to call cell_append() and nc_append() but won't be executed. We are now ready to use netdefs.hoc to help us build our own networks.

Exploiting the reusable code

Where should we begin? A good way to start is by imagining the overall organization of the entire program at the "big picture" level. We'll need the GUI library, the class definitions and other code in `netdefs.hoc`, code to specify the network model itself, and code that sets up controls for adjusting model parameters, running simulations, and displaying simulation results. Following our recommended practices of modular programming and separating model specification from user interface (see **Elementary project management** in **Chapter 6**), we turn this informal outline into an `init.hoc` file that pulls all these pieces together (Listing 11.3).

```
load_file("nrngui.hoc")
load_file("netdefs.hoc")  // code from NetWork Builder-generated hoc file
load_file("makenet.hoc")  // specifies network
load_file("rig.hoc")  // for adjusting model params and running simulations
```

Listing 11.3. The `init.hoc` for our own network program.

For now, we can comment out the last two lines with `//` so we can test `netdefs.hoc` by using NEURON to execute `init.hoc`. and then typing a few commands at the `oc>` prompt (user entries are **`Courier bold`** while the interpreter's output is plain `Courier`).

```
Additional mechanisms from files
 invlfire.mod
   1
   1
oc>objref foo
oc>foo = new IF_IntervalFire()
oc>foo
   IF_IntervalFire[0]
oc>
```

So far so good. We are ready to apply the strategy of iterative program development (see **Iterative program development** in **Chapter 6**) to fill in the details.

The first detail is how to create a network of a specific size. If we call the number of cells `ncell`, then this loop

```
for i=0, ncell-1 {
  cell_append(new IF_IntervalFire(), i, 0, 0)
}
```

will make them for us, and this nested loop

```
for i=0, ncell-1 for j=0, ncell-1 if (i != j) {
  nc_append(i, j, -1, 0, 1)
}
```

will attach them to each other. A first stab at embedding both of these in a procedure which takes a single argument that specifies the size of the net is


```
proc createnet() { local i, j
  ncell = $1
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    nc_append(i, j, -1, 0, 1)
  }
}
```

and that's what we put in the first version of `makenet.hoc`.

We can test this by uncommenting the `load_file("makenet.hoc")` line in `init.hoc`, using NEURON to execute `init.hoc`., and then typing a few commands at the `oc>` prompt.

```
oc>createnet(2)
oc>ncell
   2
oc>print cells, nclist
   List[8] List[9]
oc>print cells.count, nclist.count
   2 2
oc>for i=0,1 print cells.object(i), nclist.object(i)
IF_IntervalFire[0] NetCon[0]
IF_IntervalFire[1] NetCon[1]
oc>
```

So it works. But almost immediately a wish list of improvements comes to mind. In order to try networks of different sizes, we'll be calling `createnet()` more than once during a single session. As it stands, repeated calls to `createnet()` just tack more and more new cells and connections onto the ends of the `cells` and `nclist` lists. Also, `createnet()` should be protected from nonsense arguments (a network should have at least two cells).

We can add these fixes by changing `ncell = $1` to

```
if ($1<2) { $1 = 2 }
ncell = $1
nclist.remove_all()
cells.remove_all()
```

The first line ensures our net will have two or more cells. The last two lines use the `List` class's `remove_all()` to purge `cells` and `nclist`. Of course we check this

```
oc>createnet(1)
oc>ncell
   2
oc>createnet(2)
oc>ncell
   2
oc>createnet(3)
oc>ncell
   3
oc>
```

which is exactly what should happen.

What else should go into `makenet.hoc`? How about procedures that make it easy to change the properties of the cells and connections? For example, this

```
proc delay() { local i
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}
```

lets us set all synaptic delays to the same value by calling `delay()` with an appropriate argument. Similar `proc`s can take care of weights and cellular time constants. Setting ISIs seems more complicated at first, but after a few false starts we come up with

```
proc interval() { local i, x, dx
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}
```

This assigns the `low` ISI to the first cell in `cells`, the `high` ISI to the last cell in `cells`, and evenly spaced intermediate values to the other cells.

Does that mean the first cell is the fastest spiker, and the last is the slowest? Only if we are careful about the argument sequence when we call `interval()`. For that matter, what prevents us from calling `interval()` with one or both arguments < 0? Come to think of it, some of our other `proc`s might also benefit by being protected from nonsense arguments. For example, we might protect against negative delays by changing

```
del = $1
```

in `proc delay()` to

```
if ($1<0) $1=0
del = $1
```

and we could insert similar argument-trapping code into other `proc`s as necessary.

However, it makes more sense to try to identify a common task that can be split out into a separate function that can be called by any `proc` that needs it. It may help to tabulate the vulnerable variables and their restrictions.

Variable	Restriction
`ncell`	≥ 2
`tau`	> 0
`low` ISI	> 0
`high` ISI	≥ `low` ISI
`del`	≥ 0

Most of these are "greater than or equal to" restrictions, the two holdouts being `tau` and `low` ISI. After a moment we realize that there are practical lower limits to these

variables--say 0.1 ms for `tau` and 1 ms for `low` ISI--so "greater than or equal to" restrictions can be applied to all.

The final version of `makenet.hoc` (Listing 11.4) contains all of these refinements. The statements at the very end create a network by calling our revised `proc`s.

```
/*
returns value >= $2
for bulletproofing procs against nonsense arguments
*/

func ge() {
  if ($1<$2) {
    $1=$2
  }
  return $1
}

////////// create a network //////////

// argument is desired number of cells

proc createnet() { local i, j
  $1 = ge($1,2) // force net to have at least two cells
  ncell = $1
  // so we can make a new net without having to exit and restart
  nclist.remove_all()
  cells.remove_all()
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    // let weight be 0; we'll give it a nonzero value elsewhere
    nc_append(i, j, -1, 0, 1)
  }
  objref netcon  // leave no loose ends (see nc_append())
}

////////// specify parameters //////////

// call this settau() to avoid conflict with scalar tau

proc settau() { local i
  $1 = ge($1,0.1)  // min tau is 0.1 ms
  tau = $1
  for i=0, cells.count-1 {
    cells.object(i).pp.tau = $1
  }
}
```



```
// args are low and high

proc interval() { local i, x, dx
  $1 = ge($1,1)  // min low ISI is 1 ms
  $2 = ge($2,$1)
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}

proc weight() { local i
  w = $1
  for i=0, nclist.count-1 {
    nclist.object(i).weight = $1
  }
}

proc delay() { local i
  $1 = ge($1,0)  // min del is 0 ms
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}

////////// actually make net and set parameters //////////

createnet(2)
settau(10)
interval(10, 11)
weight(0)
delay(1)
```

Listing 11.4. Final implementation of `makenet.hoc`.

Time for more tests!

```
oc>del
0
oc>{delay(-1) print del}
0
oc>{delay(3) print del}
3
oc>createnet(4)
oc>ncell
   4
oc>del
3
oc>
```

Of course we can and should test the other `proc`s, especially `interval()`.

Our attention now shifts to creating the user interface for adjusting model parameters, controlling simulations, and displaying results. To evoke the metaphor of an experimental rig, this is placed in a file called `rig.hoc`.

An initial implementation of `rig.hoc` might look like this

```
load_file("runctl.ses")  // RunControl and VariableTimeStep

xpanel("Model parameters")
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(500,400)
```

In the spirit of taking advantage of every shortcut the GUI offers, the first statement loads a session file that recreates a RunControl and a VariableTimeStep panel configured for the desired simulation duration (Tstop = 500 ms) and integration method (adaptive integration with local time steps). The other statements set up a panel with numeric fields and controls for displaying and adjusting model parameters. This implementation of `rig.hoc` lacks two important features: a graph that displays spike trains, and the ability to change the number of cells in the network.

To prepare to record and plot spike trains, we can insert the following code right after the `load_file()` statement:

```
objref netcon, vec, spikes, nil, graster

proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()
```

For each cell in the net, this creates a new `Vector`, uses the `NetCon` class's `record()` method to record the time of that cell's spikes into the `Vector`, and appends the `Vector` to a `List`. After the end of the `for` loop that iterates over the cells, the `netcon` and `vec` `objref`s point to the last `NetCon` and `Vector` that were created, exposing them to possible interference if we ever do anything that reuses these `objref` names. The `objref netcon, vec` statement breaks the link between them and the objects, thereby preventing such undesirable effects.

The last two statements in `preprasterplot()` create a `Graph` and place it at a desired location on the screen. How can we tell what the numeric values should be for the

arguments in the `graster.view()` statement? By creating a graph (NEURON Main Menu / Graph / Voltage axis will do), dragging it to the desired location, saving it to a session file all by itself, and then stealing the argument list from that session file's `save_window_.view()` statement--being careful to change the third and fourth arguments so that the x and y axes span the correct range of values. No cut and try guesswork for us! While we're at it, we might as well use the same strategy to fix the location for our model parameter panel, but now we only need the fifth and sixth arguments to `view()`, which are the screen coordinates where the `Graph` is positioned. For my monitor, this means the second `xpanel` statement becomes `xpanel(300,370)`.

Running a new test, we find that our user interface looks like Fig. 11.22. Everything is in the right place, and time advances when we click on Init & Run, but no rasters are plotted.

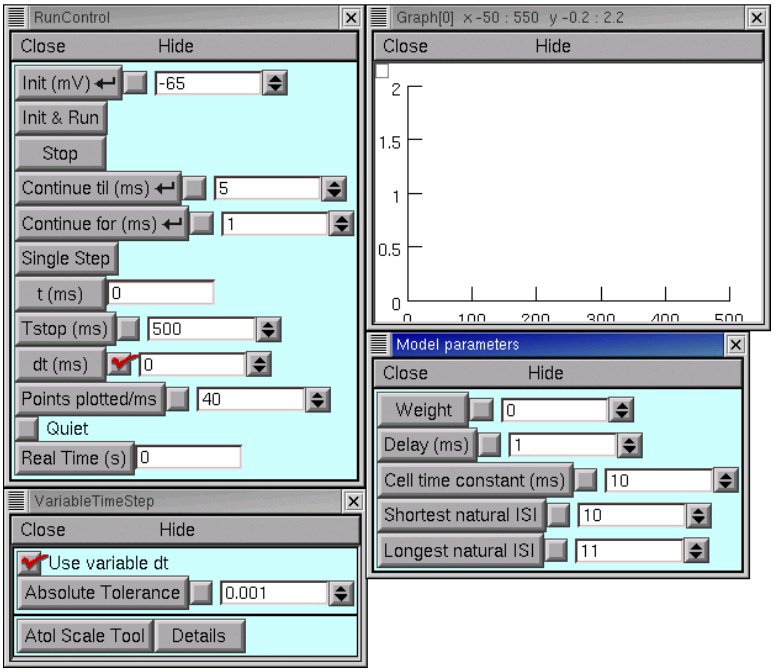


Figure 11.22. The user interface after the first revision to `rig.hoc`, in which we added `preprasterplot()`.

For each cell we need to draw a sequence of short vertical lines on `graster` whose x coordinates are the times at which that cell fired. To help us tell one cell's spikes from another's, the vertical placement of their rasters should correspond to their ordinal position in `cells`. We can do this by inserting the following code into `rig.hoc`, right after the call to `preprasterplot()`. The first thing that `proc showraster()` does is to clear any previous rasters off the `Graph`. Then, for each cell in turn, it uses three `Vector` class methods in succession: `c()` to create a `Vector` that has as many elements as the number of spikes that the cell fired, `fill()` to fill those elements with an integer that is one more than the ordinal position of that cell in `cells`, and `mark()` to mark the firing times.


```
objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}
```

Testing once again, we run a simulation and then type `showraster()` at the `oc>` prompt, and sure enough, there are the spikes. We change the longest natural ISI to 20 ms, run another simulation, and type `showraster()` once more, and it works again.

All this typing is tedious. Why not customize the `run()` procedure so that it automatically calls `showraster()` after each simulation? Adding this

```
proc run() {
  stdinit()
  continuerun(tstop)
  showraster()
}
```

to the end of `rig.hoc` does the job (see **An outline of the standard run system** in **Chapter 7: How to control simulations**).

Another test and we are overcome with satisfaction--it works. Then we change Tstop to 200 ms, run a simulation, and are disappointed that the raster plot's x axis does not rescale to match the new Tstop. One simple fix for this is to use a custom `init()` procedure that sets the raster plot to the correct size during initialization (see **Default initialization in the standard run system: stdinit() and init()** in **Chapter 8**). So we insert this

```
proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

right after our custom `run()`. Notice that this also rescales the y axis, which will be helpful when we finally add the ability to change the number of cells in the network.

Success upon success! It works!

We can finally get around to changing the number of cells. Let's think this out carefully before doing anything. We'll need a new control in the `xpanel`, to show how many cells there are and let us specify a new number. That's easy--just put this line

```
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
```


right after `xpanel("Model parameters")` so that when we change the value of `ncell`, we automatically call a new procedure called `recreate()` that will throw away the old cells and their connections, and create a new set of each.

But what goes in `recreate()`? We'll want the new cells and connections to have the same properties as the old ones. And we'll have to replace the old raster plot with a new one, complete with all the `NetCons` and `Vectors` that it uses to record spikes. So `recreate()` should be

```
proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}
```

A good place for this is right before the `xpanel`'s code.

So now we have completed `rig.hoc` (see Listing 11.5). The parameter panel has all the right buttons (Fig. 11.23) so it is easy to explore the effects of parameter changes (Fig. 11.24). How to develop an understanding of what accounts for these effects is beyond the scope of this chapter, but we can offer one hint: run some simulations of a net containing only 2 or 3 cells, using fixed time steps, and plot their membrane state variables (actually their `M` functions).

```
////////// user interface //////////

load_file("runctl.ses")  // RunControl and VariableTimeStep

// prepare to record and display spike trains
objref netcon, vec, spikes, nil, graster

proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()
```



```
objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}

// destroys existing net and makes a new one
// also spawns a new spike train raster plot
// called only if we need a different number of cells

proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}

xpanel("Model parameters")
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(300,370)

////////// custom run() and init() //////////

proc run() {
  stdinit()
  continuerun(tstop)
  showraster()  // show results at the end of each simulation
}

proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())  // rescale x and y axes
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Listing 11.5. Complete implementation of `rig.hoc`.

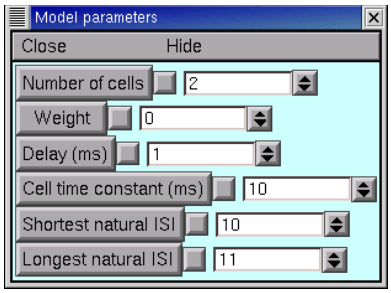
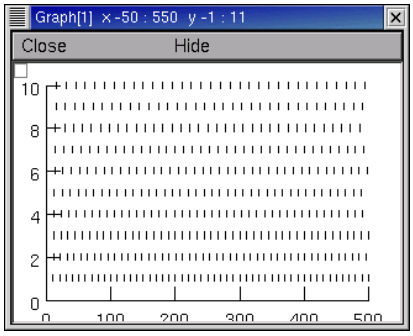


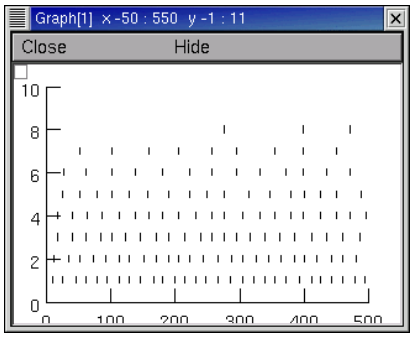
Figure 11.23. The parameter panel after addition of a control for changing the number of cells.

Figure 11.24. Simulations of a fully connected network with 10 cells whose natural ISIs are spaced uniformly over the range 10-15 ms. The rasters are arranged with ISIs in descending order from top to bottom.

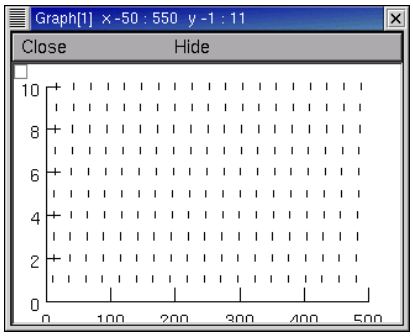
A: With all synaptic weights 0, cell firing is asynchronous and uncorrelated.



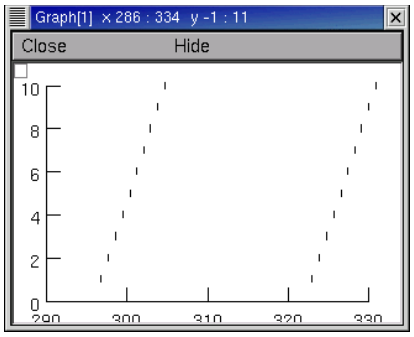
B: Mild inhibitory coupling (weight -0.2) with a delay of 1 ms silences the slowest cells and reduces the firing rates of the others. There is a suggestion of spike clustering, but no obvious synchrony or strong correlation.



C: Increasing synaptic delay to 8 ms allows the slowest cells to escape from inhibition and results in strong correlation.



D: Close examination reveals that spikes are not synchronous, but lag progressively across the population with increasing natural ISI.



References

Web site retrieved 11/8/2004. NEURON Tutorial by Andrew Gillies and David Sterratt. http://www.anc.ed.ac.uk/school/neuron/

