

Chapter 14

How to modify NEURON itself

NEURON's extensive library of functions and graphical tools has been developed with an eye to providing those features that are most widely applicable in empirically-based neural modeling. Since they are necessarily rather generic, it is sometimes desirable to change these features or add new ones in order to meet the special needs of individual projects. This is particularly important where the graphical user interface is concerned, given the well-established role of the GUI in enhancing software utility. Here we show how to create new GUI tools and add new functions to NEURON.

A word about graphics terminology

Computer graphics literature is full of technical jargon, of which we will use only the most minute part. Think of a *scene* as being like a sheet of paper that has its own coordinate system (*scene coordinates*), and a *view* as a rectangular area on that sheet. *Model* (in the graphical sense) is just a synonym for "scene," and "model coordinates" are the same as "scene coordinates." The *screen* is simply the computer's display (CRT, flat panel, etc.), and *screen coordinates* represent actual physical locations on the display. Finally, when something is "mapped to the screen," the visible consequence is that a view of it appears on the display.

Graphical interface programming

In **Chapter 6** we noted that an iterative process of incremental revision and testing can be an effective strategy for software development. This is especially true when the task is to create a tool that interacts with the user through a graphical interface. It is rarely clear at the outset what manipulations of objects on the screen are feasible with reasonable programming effort, have obvious or easily learned meanings on first approach, and allow users to straightforwardly and rapidly reach their goals. The approach outlined in this section has proven useful in creating every one of the tools in the NEURONMainMenu suite.

As a concrete example, suppose we want to make a tool for graphically specifying the parameters of the Boltzmann function

$$y(x) = \frac{1}{e^{4k(d-x)}} \quad \text{Eq. 14.1}$$

The half maximum of this curve is located at $(d, 0.5)$, and the slope at this point is k .

The graphical metaphor for setting the value of d seems straightforward: just use the cursor to drag the $(d, 0.5)$ point on the curve horizontally. By putting a small open square mark centered at $(d, 0.5)$ on the curve, we can suggest that this is a user-selectable "control point."

How to set the value of k is less obvious. It seems quite natural to use the cursor's x coordinate for midpoint control, but simultaneously using its y coordinate to control the slope turns out to be neither intuitive nor convenient. One alternative could be to use a second control point at the 80% y value that can be dragged back and forth horizontally; this limits the minimum slope, but it might be adequate. Another possibility is to place the slope control point at some fixed distance from the midpoint. In this case the function is always drawn so that the slope control point is on the line between the current cursor location and the midpoint. Finally, perhaps it would be better to allow the user to click on any point of the curve and drag, and have the GUI tool redraw the curve so that it follows the cursor.

But we're getting ahead of ourselves. We can always experiment with the various styles of setting the slope parameter later--the burning question now is how to even get started.

Clearly we need a canvas on which to plot the curve, and a way to get mouse button events and cursor coordinates when one clicks in the canvas and drags the cursor to a new location. We can put a `Graph` on the screen with

```
objref g
g = new Graph()
```

Now we can do some exploratory tests of how to deal with mouse events. Let's specify a procedure to handle a mouse event

```
g.menu_tool("A new mouse handler", "handle_mouse")
```

This creates a new radio style menu item labeled "A new mouse handler" that appears at the end of the primary graph menu. If this menu item is selected, then future mouse button presses in the `Graph`'s canvas will call the procedure named `handle_mouse()` with four arguments. These arguments specify the state of the left mouse button, cursor coordinates, and whether the Shift, Control, and/or Alt key is being pressed. The `handle_mouse()` procedure does not have to be defined when `menu_tool()` is executed, but it certainly must be defined by the time it is supposed to be called. As a quick test, we can use this procedure

```
proc handle_mouse() {
    print $1, $3, $3, $4
}
```

for a concrete demonstration of mouse event handling. This is an example of how simple tests can provide a far better understanding of the meaning of a method, than we could ever gain from just reading the reference manual.

General issues

We can go quite far by trying things out one at a time and verifying that the hoc code is working the way we want. However, experience has shown that sooner or later it is becomes necessary to grapple with the following issues.

- How to encapsulate a tool so that its variables do not conflict with other tools. This tool defines the variables k and d , and it will probably also demand that we invent a lot of other names for functions and variables. Keeping all of these names localized to a particular context so that it doesn't matter if they are used in other contexts is one of the benefits of object-oriented programming.
- How to allow many instances of the same type of tool. If this tool is ever used as a part of some larger tool, e.g. a channel builder that describes steady state as a function of voltage, there will be a need for separate pairs of Boltzmann parameters for every channel state transition. This is another benefit that one gets almost for free with object-oriented programming.
- How to save a tool so the user doesn't lose data that was specified with it. The values of k and d for one of our objects may represent a great deal of thought and effort. Even the small matter of repeatedly dragging a window to a desired location and resizing it quickly becomes tedious. The user should be able to save the tool, with its present state, in a session file so that it can be re-instantiated by retrieving the session file.
- How to destroy the tool when its window is closed. We want our tool to exist only by sufferance of the user. It should be destroyed if and only if the user presses the Close button. Objects in hoc are reference counted, which normally means that an object stays in existence only as long as there is some object reference (`objref`) that points to it (see **Creating and destroying an object in Chapter 13**). Think of a reference as a label on the object: when the last label is removed the object is destroyed. If the reference issue is not properly handled, two opposite problems result. First, closing the window would not reclaim the memory used by the tool. This isn't so bad in our situation, but it could be very confusing for tools that manage, for example, a point process or even an entire cell! The second problem is that a tool instance and its window could be inadvertently destroyed even if the user hadn't pressed the window's Close button. This can happen if creating a new instance of the tool reuses the only reference to an already existing instance of the tool.

Although it may seem tedious at first, the effort of starting a tool development project with a pattern that addresses all these issues is quickly repaid. We create such a pattern in the following pages, starting by defining a new class that will be used to encapsulate the tool data and code. For the sake of concreteness, we will assume that the hoc code for our tool will be saved to a file named `bp.hoc`.

A pattern for defining a GUI tool template

The name we have chosen for this class is `BoltzmannParameters` (Listing 14.1). The overall style of the pattern is to first declare the names of things used in the template, and then to declare procedures and functions. The `public g` statement declares that `g` is

accessible to the outside world, and `objref g` declares that its type is "reference to an object" (see **Chapter 13**). When a new instance of the `BoltzmannParameters` class is created, the `init()` procedure in this template will bring up a new `Graph`; this is merely so that creating a new `BoltzmannParameters` object will produce a visible result (Fig. 14.1).

The two statements that follow the template definition create an instance of the class. This facilitates the edit-run-diagnose cycle by ensuring that, when NEURON executes this code, we don't need to type anything into the interpreter in order to see the result of our file modifications.

As an aside, we must note that mistyping and other trivial errors are common. This is yet another reason for developing a program in stages, making only a few additions at a time. Building a tool requires many iterations of incrementally editing and adding to the `hoc` code that defines it, and then launching NEURON to test it. In a windowed desktop environment, this can be facilitated by using two windows: one for keeping the file open in an editor, and the other for running NEURON after an editor save (e.g. by double clicking on the name of the `hoc` file).

```

begintemplate BoltzmannParameters
  public g
  objref g

  proc init() {
    g = new Graph()
  }
endtemplate BoltzmannParameters

// the following lines facilitate debugging
objref b
b = new BoltzmannParameters()

```

Listing 14.1. Initial version of `bp.hoc` creates an object and produces a visible result (Fig. 14.1).

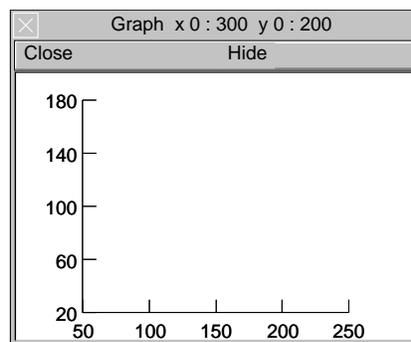


Fig. 14.1. The graph window generated by the code in Listing 14.1.

After executing the code in Listing 14.1, we can perform additional tests by entering a few simple commands into the interpreter at the `oc>` prompt:

```

oc>b
  BoltzmannParameters[0]
oc>b.g
  Graph[0]
oc>b.init()
init not a public member of BoltzmannParameters

```

The `init()` procedure is automatically called whenever a new instance of the template is created (see **Initializing variables in an object** in **Chapter 13**). In most cases it doesn't make sense to call `init()` again, so it is usually not declared public.

Enclosing the GUI tool in a single window

We quickly discover that the relationship between the window on the screen and the object `BoltzmannParameters[0]` is inappropriate. Pressing the Close button destroys the window but doesn't get rid of the `BoltzmannParameters[0]` object, because the latter is still referenced by `b`. Furthermore, the `BoltzmannParameters[0]` object is not protected from outside interference--redeclaring `b` or making it reference another object will drop `BoltzmannParameters[0]`'s reference count to 0 and destroy it. For a tool, this should only happen when one closes the window.

The current situation is that `b` references our tool instance, `b.g` references the `Graph` in our tool, but the window doesn't reference anything. We need a way for the window to reference the tool. To make this happen, we augment our pattern by enclosing the `Graph` in a `VBox` which itself references the tool, and making sure that this `VBox` is the only reference to the tool. The pattern now looks like Listing 14.2.

```

begintemplate BoltzmannParameters
  public g, box
  objref g, box, this

  proc init() {
    box = new VBox()
    box.ref(this)
    box.intercept(1)
    g = new Graph()
    box.intercept(0)
    box.map()
  }
endtemplate BoltzmannParameters

// the following lines facilitate debugging
objref tmpobj
proc makeBoltzmannParameters() {
  tmpobj = new BoltzmannParameters()
  objref tmpobj
}
makeBoltzmannParameters()

```

Listing 14.2. `bp.hoc` revised for proper management of reference count.

Between the `box.intercept(1)` and `box.intercept(0)` statements, anything that would create a window becomes arranged vertically in a `VBox` (horizontally if it was an `HBox`). When this is declared as an object reference, it always refers to the object that declared it. The `box.ref(this)` statement has the effect that, when the last window

created by the box is closed, the reference count of the tool is decremented. If this makes the reference count equal to 0, the tool is destroyed. When boxes are nested, only the outermost (the one with an actual window) requires this idiom. The code at the end of this revised `bp.hoc` creates an instance of our tool, and then immediately redeclares `tmpobj` in order to guarantee that the only remaining reference to the tool is the one associated with the `Vbox`.

The pattern is now in a form that can be integrated into the standard NEURONMainMenu / Tools / Miscellaneous menu. We can do this with

```
NEURONMainMenu[0].miscellaneous_add("BoltzmannParameters", \
    "makeBoltzmannParameters()")
```

Those who recall our injunction against writing code that invokes unique object names (see **Object references vs. object names** in **Chapter 13**) may be somewhat uncomfortable at the appearance of the object name `NEURONMainMenu[0]` in this statement. Such excellent retention of the lore of `hoc` is to be commended. Rest assured, however, that there is never more than one instance of the `NEURONMainMenu` class, and when such an object exists, it is *always* called `NEURONMainMenu[0]`. Hence this object name is *truly* unique, making this a situation in which it is safe to inscribe the proscribed.

Our `init()` procedure is collecting a lot of odds and ends; readability would be better served by separating the different idioms into procedures, as shown in Listing 14.3. Also note the changes to `map()`, which add the useful feature of giving the window a title that reflects the name of the tool instance. These changes produce the window shown in Fig. 14.2.

```
strdef tstr

proc init() {
    build()
    map()
}

proc build() {
    box = new VBox()
    box.ref(this)
    box.intercept(1)
    g = new Graph()
    box.intercept(0)
}

proc map() {
    sprintf(tstr, "%s", this)
    box.map(tstr)
}
```

Listing 14.3. Revision of `init()` to enhance clarity.

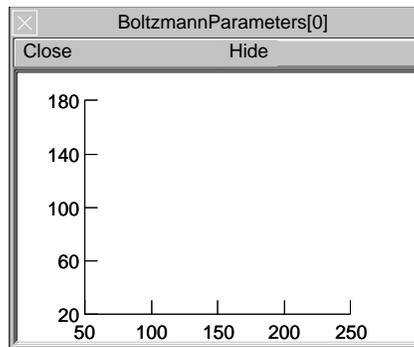


Fig. 14.2. Changes in the `map()` procedure (Listing 14.3) turn the window title into the name of the tool instance (compare with Fig. 14.1).

Saving the window to a session

The last enhancement to our basic tool pattern provides a way to save the tool in a session file so it can be recreated properly when the session file is loaded. This enhancement involves several changes to `bp.hoc` (see Listing 14.4). Before examining these, we must point out that everything in this code is independent of the future implementation of the particulars of what we want our tool to do. It just gives us a starting framework to which we can add specific functionality.

The principal change to `bp.hoc` is the addition of a `save()` procedure that can print executable `hoc` code to the session file. Most of the statements in `save()` are of the form `box.save("string")`. Since our tool has a `Graph`, it is a good idea to save its size. This can be done explicitly by building an appropriate statement, but a simple idiom is to call its `save_name()` method (see `g.save_name("ocbox.g", 1)`) since this will also save any new views of the graph that were generated by the user. This must be done at the top level, so the `Graph` reference has to be public.

A related change to `bp.hoc` is the addition of `box.save("save()")` to the `build()` procedure. This designates `save()` as the procedure that will be called when the tool is saved to a session file.

Changes to `init()` and `map()` are necessitated by the fact that we want the tool to be properly mapped to the screen regardless of whether we have created it for the first time, or instead are reconstituting it from a session file. If we are creating a tool *de novo*, it should appear in a default location with default values for its various user-specifiable parameters. However, if we are recreating it from a session file, it should appear in the same location and with the same parameter values as when we saved it.

Here "top level" refers to the interpreter dealing with global variables rather than variables that are only visible inside templates. Put another way, one can say the interpreter is executing at the top level unless it is executing code that is declared in a template or class. When the interpreter is not at the top level, it is often useful to temporarily get back to the top level with `execute("statement")`.

```

begintemplate BoltzmannParameters
  public g, box, map
  objref g, box, this
  strdef tstr

  proc init() {
    build()
    if (numarg() == 0) {
      map()
    }
  }

  proc build() {
    box = new VBox()
    box.save("save()")
    box.ref(this)
    box.intercept(1)
    g = new Graph()
    box.intercept(0)
  }

  proc map() {
    sprintf(tstr, "%s", this)
    if (numarg() == 0) {
      box.map(tstr)
    } else {
      box.map(tstr, $2, $3, $4, $5)
    }
  }

  proc save() {
    box.save("\load_file(\"bp.hoc\", \"BoltzmannParameters\")\n}\n{")
    box.save("ocbox_ = new BoltzmannParameters(1)")
    box.save("\n{object_push(ocbox_)}")
    // insert tool-dependent statements here
    box.save("{object_pop()}\n{")
    g.save_name("ocbox_.g", 1)
  }
endtemplate BoltzmannParameters

// the following lines facilitate debugging
objref tmpobj
proc makeBoltzmannParameters() {
  tmpobj = new BoltzmannParameters()
  objref tmpobj
}
makeBoltzmannParameters()

```

Listing 14.4. This refinement of `bp.hoc` contains a tool that can be saved and retrieved from a session file. Despite its specific name, this "BoltzmannParameters" template is a *completely generic starting point* that can be customized to implement specific GUI tools.

Making this happen involves three related changes to `bp.hoc`. First, the `init()` procedure has been modified so that calling it without an argument will cause the window to be mapped immediately with its defaults. Otherwise, the mapping is deferred, so that code in the session file can specify its location etc.. The second change is to the `map()` procedure: when called with no arguments, it makes the tool appear in its default position. Alternatively `map()` can be called from the session file with arguments that specify where the tool will be drawn on the screen. This brings us to the third change,

which is to add `map` to the `public` statement at the top of the template (otherwise `map()` couldn't be called from the session file).

```
load_file("nrngui.hoc")
objectvar save_window_, rvp_
objectvar scene_vector_[3]
objectvar ocbox_, ocbox_list_, scene_, scene_list_
{ocbox_list_ = new List() scene_list_ = new List()}
{pwmman_place(0,0,0)}

//Begin BoltzmannParameters[0]
{
load_file("bp.hoc", "BoltzmannParameters")
}
ocbox_ = new BoltzmannParameters(1)
{object_push(ocbox_)}
{object_pop()}
}
save_window_ = ocbox_.g
save_window_.size(0,300,0,200)
scene_vector_[2] = save_window_
ocbox_.g = save_window_
save_window_.save_name("ocbox_.g")
}
ocbox_.map("BoltzmannParameters[0]", 211, 211, 315, 251.1)
}
objref ocbox_
//End BoltzmannParameters[0]

objectvar scene_vector_[1]
{doNotify()}
```

Listing 14.5. The contents of the session file created by executing Listing 14.4 and then saving the tool to a session.

Saving this tool to a session file by itself produces the text shown in Listing 14.5. NEURON's internal code for saving session files first prints several statements that, among other things, declare object reference variables used by various kinds of windows. The only statement that concerns us here is the declaration of `ocbox_`, which is used by boxes to map themselves to the screen as windows.

Next it prints a comment that contains the title of the window, and then it calls our `save()` procedure, which prints the lines in the `ses` file up to `save_window_`. The first of these strings is

```
load_file("bp.hoc", "BoltzmannParameters")
```

which makes sure that the code for our template is loaded, and the next is

```
ocbox_ = new BoltzmannParameters(1)
```

Although `ses` files use the deprecated `objectvar`, the preferred keyword for declaring an object reference is `objref` (see **Declaring an object reference** in Chapter 13)

which uses the `ocbox_` object reference to create an instance of our tool. Note the use of an argument to prevent the `init()` procedure from mapping the tool to the screen. The `object_push()` causes succeeding statements to be executed in the context of the object. This allows access to any of the variables or functions in the object, *even if they are not public*--which is necessary in order to save the state of our object. The graph reference must be public because `object_pop()` returns to the previous context, which is normally the top level of the interpreter.

After the `save()` procedure returns, the `map()` method is called for `ocbox_` with the first argument being the title of the window, followed by four more size arguments which specify the screen location of the window. Since we commandeered `ocbox_` as the reference for our tool, our own `map()` is called instead, and we replace whatever title was used by the current authoritative instance name. Our `map()` also uses the four placement arguments if they exist (they don't if `map()` is called from `init()`).

Tool-specific development

Plotting

This tool is supposed to display a Boltzmann function, so we clearly need to define the function, declare the parameters we will be managing, and plot the function. Any names we invent will be nicely segregated so they can't conflict with names at the top level of the interpreter or names in any other tool. Thus we can safely use the natural names that first come to mind.

The Boltzmann function can go anywhere in the template.

```
func b() {
    return 1/(1 + exp(-4*k*(d - $1)))
}
```

The first argument is referred to as `$1` in a function body. The parameters are used here, and this is sufficient to declare them, but their default values would be 0. To give them better defaults we add

```
k = 1
d = 0
```

to the body of `init()`.

Plotting the function raises the issue of the proper domain and the number of points. We could get sophisticated here, plotting points at x values that nicely follow the curve, but instead we will just plot 100 line segments over a domain defined by the current graph scene size.

```

proc pl() {local i, x, x1, x2
  g.erase_all
  x1 = g.size(1)
  x2 = g.size(2)
  g.begin_line
  for i=0, 100 {
    x = x1 + i*(x2 - x1)
    g.line(x, b(x))
  }
  g.flush
}

```

This procedure starts with `erase_all` because it will eventually be called whenever `k` or `d` is changed. For now we get a drawing on the graph by calling `pl()` at the end of the `map()` procedure.

Making these changes and running `bp.hoc` produces the error message

```

nrniv: syntax error in bp.hoc near line 51
  return 1/(1 + exp(-4*k*(d - $1))
                                ^

```

This is a parse error, which means that something is wrong with the syntax of the statement (see **Error handling in Chapter 12: hoc, NEURON's interpreter**). The parser often fails a few tokens after the actual error, but in this case the carat is right on target: this statement is missing a closing parenthesis.

Adding the parenthesis and trying again, we find that `begin_line` was misspelled. This error occurred at run time, so the call stack was printed, which gives very helpful information about the location of the error.

```

begin_line not a public member of Graph
nrniv: Graph begin_line in bp.hoc near line 77
  makeBoltzmannParameters()
                                ^

BoltzmannParameters[0].pl()
BoltzmannParameters[0].map()
BoltzmannParameters[0].init()
makeBoltzmannParameters()

```

Another try gives

```

nrniv: exp result out of range in bp.hoc near line 77
  makeBoltzmannParameters()
                                ^

```

Testing `exp(-10000)` shows there is no underflow problem with this function, and `exp(700)` is below the overflow limit. Taking this into account requires a slight elaboration of `b()`

```

func b() { local x
  x = -4*k*(d - $1)
  if (x > 700) { return 0 }
  return 1/(1 + exp(x))
}

```

Now we no longer get an "out of range" message, but the `Graph` still appears to be empty, looking identical to Fig. 14.2. Invoking the item `View = plot` from the `Graph`'s secondary menu turns this into a featureless grey rectangle (Fig. 14.3).

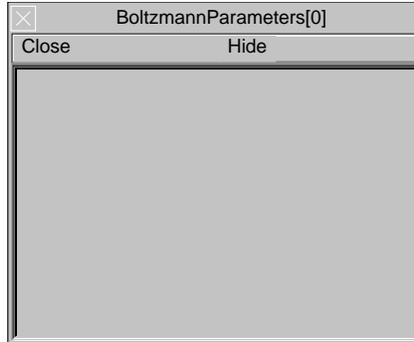


Fig. 14.3. After the numeric overflow problem is eliminated, the `Graph` seems to be empty. However, `View = plot` turns the window contents grey, indicating that something else is wrong.

This is puzzling at first, but after a moment's reflection we try adding

```
print $1, 1/(1 + exp(x))
```

just before the `return` statement in `func b()`, and run the code again. This diagnostic test results in NEURON printing just one pair of values

```
25 3.720076e-44
```

which is a clue to what should have been obvious: instead of one mistake, there are two.

The first error was in the way the desired x value was calculated in `proc pl()`. This should have read

```
x = x1 + i*(x2 - x1)/100
```

The second error was to accept the default dimensions of the `Graph`, which have x running from 25 to 275 so that `b()` is too small to be of interest. This can be fixed by adding

```
g.size(-5, 5, 0, 1)
```

after creating the `Graph` in the `build()` procedure.

Now the `Graph` produced by this template looks pretty good (Fig. 14.4), but the plotted curve has a slope of -1 at $x = 0$. This is just carelessness; maybe we fell into thinking that exponentials always have negative arguments, as if everything is a time constant. Let us strike a mutual pact never to make another mistake.

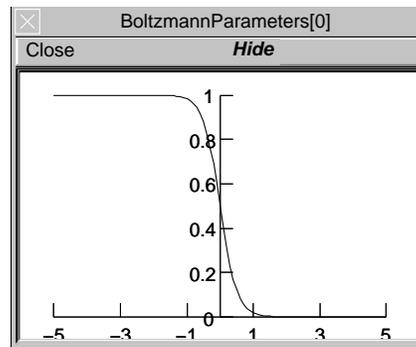


Fig. 14.4. The Graph after fixing the range of x values and specifying appropriate dimensions.

Handling events

The following lines, placed in the `build()` procedure after the Graph has been created, specifies how mouse events are handled.

```
g.menu_tool("Adjust", "adjust")
g.exec_menu("Adjust")
```

The `exec_menu()` selects the menu item `Adjust` as the current tool for mouse event handling. That is, `press`, `drag`, or `release` of the left mouse button while the mouse cursor is over the Graph will cause a procedure named `adjust()` to be called with arguments that specify the type of event (`press`, `drag`, or `release`), the scene coordinates of the cursor, and the state of the `Shift`, `Control`, and `Alt` keys.

```
proc adjust() {
  if ($1 == 2) { // left mouse button pressed
    adjust_ = 0
    // $2 and $3 are scene coords of mouse cursor
    if (ptdist($2, $3, d, 0.5) < 100) {
      adjust_ = 1
    } else if (ptdist($2, $3, $2, b($2)) < 100) {
      adjust_ = 2
    }
  }
  if (adjust_ == 1) {
    d = $2
    pl()
  }
  if (adjust_ == 2) {
    // keep y value within function domain
    if ($3 > 0.99) $3 = 0.99
    if ($3 < 0.01) $3 = 0.01
    // avoid singularity at x == d
    if ($2 > d || $2 < d) {
      // change k so that curve passes through
      // cursor location
      k = log(1/$3 - 1)/(4 * (d - $2))
      pl()
    }
  }
}
```

```
func ptdist() {
    return 1
}
```

Listing 14.6. The handler for mouse events is `proc adjust()`.

The `adjust()` procedure needs to decide whether d or k is the parameter of interest. If the mouse cursor is within 10 pixels of the current $(d, 0.5)$, then on every event we'll set the value of d to the cursor's x coordinate. Otherwise, if the cursor is within 10 pixels of any other point on the curve, we'll set the value of k so that the curve passes through the cursor's location. Just to get things working, for now we defer the calculation of pixel distance between two points, using instead a `ptdist()` function that always returns 1 so that we always end up setting d . We compare to 100, instead of 10, because we won't bother taking the square root when we calculate the distance.

Running `bp.hoc`, placing the mouse cursor over the Graph canvas, and dragging it back and forth with the mouse button pressed shows satisfying tracking of the curve with the cursor (Fig. 14.5 A), so there is no need to worry yet about performance.

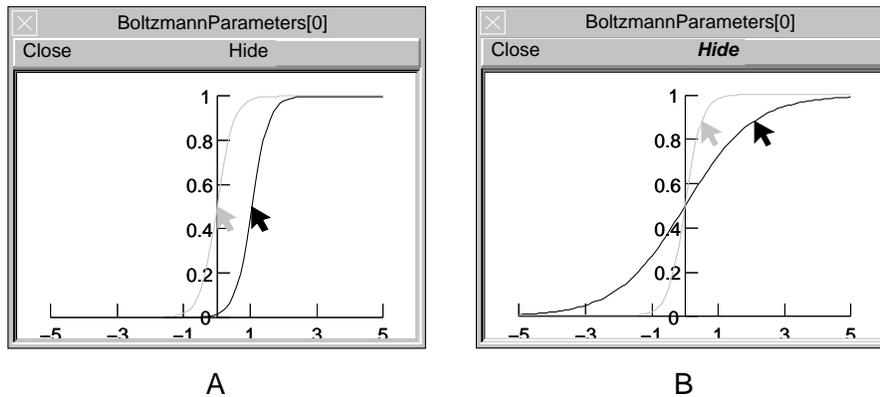


Fig. 14.5. A. The code in Listing 14.6 enables click and drag to shift the curve from side to side. B. Forcing `adjust_` to 2 allows us to test the use of this tool to set k .

Testing the ability of `adjust()` to change k is easily done by temporarily forcing `adjust_` to have the value 2 on the press event. In considering the inverse of the Boltzmann function, there are limits on the range $0 < y < 1$ and a singularity at $x == d$. This first attempt at doing something reasonable is likely to be sufficient but we won't know until we try it. In fact, it works much better than adding another control point for k (Fig. 14.5 B).

Some kind of mark, e.g. a little open square, is needed to indicate the control point at $(d, 0.5)$. To do this we put

```
g.mark(d, 0.5, "s", 8)
```

just before `g.flush` in the `pl()` procedure. A size of 8 points is perhaps a bit large for a control point (Fig. 14.6). Later we will add a visual readout of the values of d and k .

So far, it has been most natural to specify plotting and event handling in terms of scene coordinates. However, calculating proximity between something plotted on a Graph and the location of the mouse cursor is best done in pixel or screen coordinates,

since that avoids the problems caused by disparities between scene x and y scales. This requires revisions to `ptdist()`, where the information needed to transform between scene and screen coordinates will be obtained by a sequence of calls to `view_info()`. To recall the details of `view_info()`'s various arguments, we'll have to refer to the documentation of the `Graph` class and its methods in the Programmer's Reference.

Since each `Graph` can be displayed in more than one view simultaneously, the transformation between scene coordinates and screen coordinates depends on which view we are interested in. Of course, the view we want is the one that contains the mouse cursor, and this is the purpose of the first call to `view_info()`.

```
func ptdist() {local i, x1, y1, x2, y2
  i = g.view_info() // i is the view in which
                  // the mouse cursor is located
  // $1..$4 are scene (x,y) of mouse cursor
  // and corresponding point on curve, respectively
  x1 = g.view_info(i, 13, $1)
  y1 = g.view_info(i, 14, $2)
  x2 = g.view_info(i, 13, $3)
  y2 = g.view_info(i, 14, $4)
  return (x1 - x2)^2 + (y1 - y2)^2
}
```

A little testing shows us that it is difficult to get the cursor near enough to the curve when the slope is large, because even a slight offset in the x coordinate causes a large jump of $b(x)$ away from the cursor's y coordinate. We can fix this by adding a clause to `adjust()` that detects whether the horizontal distance between the cursor and the curve is small. Thus

```
} else if (ptdist($2, $3, $2, b($2)) < 100) {
```

becomes

```
} else if (ptdist($2, $3, $2, b($2)) < 100 \
           || abs($2 - b_inv($3)) < 10 {
```

(the backslash at the end of a line is a statement continuation character). Implementation of `b_inv()`, the inverse to the Boltzmann function, is more than 90% argument and parameter testing.

```
func b_inv() {local x
  if ($1 >= 1) {
    x = 700
  } else if ($1 <= 0) {
    x = -700
  } else {
    x = log(1/$1 - 1)
  }
  if (k == 0) {
    return 1e9
  }
  return d - x/(4*k)
}
```

Finishing up

As mentioned earlier, it would be nice to have a direct indication of the values of the parameters. To do this, we add a horizontal panel to the bottom of the `vBox` and make the field editor buttons call the plot function `pl()` whenever the user changes the value. Also, when a `Graph` menu tool is created within the scope of an open `xpanel`, the tool selector appears as a radio button in the panel. The code fragment to do this, in context in the `build()` procedure, is

```
g = new Graph()
g.size(-5,5,0,1)
xpanel("", 1)
  xpvalue("k", &k, 1, "pl()")
  xpvalue("d", &d, 1, "pl()")
  g.menu_tool("Adjust", "adjust")
xpanel()
g.exec_menu("Adjust")
```

We use `xpvalue()` and pointers, instead of `xvalue()` and variable names, because field editors assume variable names are at the top level of the interpreter. This is in contrast to action statements, such as `pl()` in this instance, which are executed in the context of the object. Figure 14.6 shows what the tool now looks like.

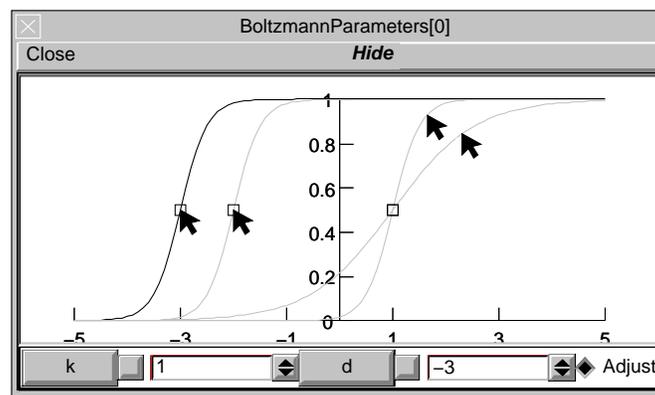


Fig. 14.6. Final appearance of the "Boltzmann Parameters" tool. The parameter values displayed are for the leftmost curve. Other curves with arrows suggest the process of selection and dragging the mouse.

The last touch is to change the `save()` procedure so that it saves the tool-specific state. By default, scene coordinates are saved instead of view coordinates, so it is a good idea in this case to make them equivalent. This is done by adding these tool-dependent statements to `save()`

```
// insert tool-dependent statements here
sprintf(tstr, "{k=%g d=%g}", k, d)
box.save(tstr)
g.exec_menu("Scene=View")
```

The first two statements, which incidentally take advantage of the `tstr` string variable that already exists, write the assignments of k and d to the session file, while the third one takes care of making scene and view coordinates equivalent.

The entire code for the final implementation of our tool is shown in Listing 14.7.

```

begintemplate BoltzmannParameters
public g, box, map
objref g, box, this
strdef tstr

proc init() {
    k = 1
    d = 0
    build()
    if (numarg() == 0) {
        map()
    }
}

proc build() {
    box = new VBox()
    box.save("save()")
    box.ref(this)
    box.intercept(1)
    g = new Graph()
    g.size(-5,5,0,1)
    xpanel("", 1)
        xvalue("k", &k, 1, "pl()")
        xvalue("d", &d, 1, "pl()")
    g.menu_tool("Adjust", "adjust")
    xpanel()
    g.exec_menu("Adjust")
    box.intercept(0)
}

proc map() {
    sprintf(tstr, "%s", this)
    if (numarg() == 0) {
        box.map(tstr)
    } else {
        box.map(tstr, $2, $3, $4, $5)
    }
    pl()
}

proc save() {
    box.save("load_file(\"bp.hoc\", \"BoltzmannParameters\")\n\n{")
    box.save("ocbox_ = new BoltzmannParameters(1)")
    box.save("{\n{object_push(ocbox_)}")
    // insert tool-dependent statements here
    sprintf(tstr, "{k=%g d=%g}", k, d)
    box.save(tstr)
    g.exec_menu("Scene=View")
    // end of tool-dependent statements
    box.save("{object_pop()}\n{")
    g.save_name("ocbox_.g", 1)
}

func b() { local x
    x = 4*k*(d - $1)
    if (x > 700) { return 0 }
    return 1/(1 + exp(x))
}

```

```

proc pl() { local i, x, x1, x2
  g.erase_all
  x1 = g.size(1)
  x2 = g.size(2)
  g.beginline
  for i=0, 100 {
    x = x1 + i*(x2 - x1)/100
    g.line(x, b(x))
  }
  g.mark(d, 0.5, "s", 8)
  g.flush
}

proc adjust() {
  if ($1 == 2) { // left mouse button pressed
    adjust_ = 0
    // $2 and $3 are scene coords of mouse cursor
    if (ptdist($2, $3, d, 0.5) < 100) {
      adjust_ = 1
    } else if (ptdist($2, $3, $2, b($2)) < 100 \
      || abs($2 - b_inv($3)) < 10) {
      adjust_ = 2
    }
  }
  if (adjust_ == 1) {
    d = $2
    pl()
  }
  if (adjust_ == 2) {
    if ($3 > 0.99) $3 = 0.99
    if ($3 < 0.01) $3 = 0.01
    if ($2 > d || $2 < d) {
      k = log(1/$3 - 1)/(4 * (d - $2))
      pl()
    }
  }
}

func ptdist() {local i, x1, y1, x2, y2
  i = g.view_info() // i is the view in which
                  // the mouse cursor is located
  // $1..$4 are scene (x,y) of mouse cursor
  // and corresponding point on the curve, respectively
  x1 = g.view_info(i, 13, $1)
  y1 = g.view_info(i, 14, $2)
  x2 = g.view_info(i, 13, $3)
  y2 = g.view_info(i, 14, $4)
  return (x1 - x2)^2 + (y1 - y2)^2
}

```

```
func b_inv() {local x
  if ($1 >= 1) {
    x = 700
  } else if ($1 <= 0) {
    x = -700
  } else {
    x = log(1/$1 - 1)
  }
  if (k == 0) {
    return 1e9
  }
  return d - x/(4*k)
}
endtemplate BoltzmannParameters
```

Listing 14.7. Complete source code for the BoltzmannParameters tool.

Chapter 14 Index

E

execute() 7

G

good programming style

 iterative development 1, 4

Graph class

 beginline() 11

 erase_all() 11

 exec_menu() 13

 flush() 14

 mark() 14

 menu_tool() 2

 save_name() 7

 size() 12

 view_info() 15

GUI

 graphics terminology 1

 model 1

 model coordinates 1

 scene 1

 scene coordinates 1

 making scene and view coordinates equivalent 16

 vs. screen coordinates 14

 screen 1

 mapping to the screen 1

 screen coordinates 1

 tools

 Close button 3, 5

 view 1

GUI tool development

 general issues

- allowing multiple instances 3, 5, 6
- destroying 3, 5, 6
- encapsulating 3
- saving and retrieving 3, 7, 16
- generic starting point 8
- mapping to the screen 7, 8
- window title 6

H

- HBox 5

- hoc

- top level of the interpreter 7

M

- mouse

- events 2
 - cursor coordinates 2, 13
 - handling 2, 13

N

- NEURON Main Menu GUI

- Tools

- Miscellaneous 6

- NEURONMainMenu class

- miscellaneous_add() 6

- NEURONMainMenu object

- is always NEURONMainMenu[0] 6

O

- object

- reference count 6

- object reference

- this 5

S

- session file

- object_pop() 10

object_push() 10

ocbox_ 9

V

VBox 5

VBox class

intercept() 5

map() 5, 6, 8

mapping to the screen

 window title 6

ref() 5

save() 7

X

xpanel() 16

xpvalue() 16

xvalue() 16