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 these features are necessarily rather generic, it is sometimes desirable to have to change them or add new ones in order to meet the special needs of individual projects. Here we show how to create new GUI tools and add new functions to NEURON.

Graphical interface programming

The importance of the graphical user interface to the utility of software has been repeatedly demonstrated in recent years. Creating a tool that interacts with the user through a graphical interface is usually a highly iterative process. 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 strategy 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 that allows us to graphically specify the parameters of the Boltzmann function

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

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 function horizontally. By placing a small open square mark centered at (d, 0.5) on the function, 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 how small the slope can be, but it might be adequate. Another possibility is to place the slope control point a fixed distance away 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

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

click on any point of the curve and redraw the curve so that it follows the cursor wherever the cursor is dragged.

Experimenting with the various styles of setting the slope parameter can be done 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 get a Graph onto the screen with

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

and test the mouse input handling capability by specifying what procedure handles a mouse event with

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

This creates a new radio style menu item at the end of the graph menu with the label "A new mouse handler". If this menu item is selected, then future mouse button presses in the graph canvas will cause the procedure named `handle_mouse()` to be called 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 the `menu_tool()` function 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 input handling. Using tests like this is an excellent way to get feedback about the meaning of a method that is far more specific than reading the reference manual can provide.

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 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 a larger channel builder tool to help describe the steady state as a function of voltage, there will be a need for separate pairs of Boltzmann parameters for every

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

channel state transition. Here is another benefit one almost gets for free with object oriented programming.

- How to save a tool so the user doesn't lose the data that was specified with the tool. 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 its desired location and resizing it tends to become unbearably tedious. The user should be able to save the window in a session file so that the tool is re–instantiated any time the session file is retrieved.
- 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. This normally means that an object stays in existence as long as there is some object reference variable (`objref`) that references it. If an object is no longer referenced, it is destroyed and the memory holding its data is available for any other purpose. Think of a reference as a label on the object: when the last label is removed the object is destroyed. This causes two opposite problems for graphical interface tools if special provision is not made to handle the reference issue. First, the memory used by the tool would not be reclaimed when the window is closed. This isn't so bad in our situation, but can be very confusing for tools that manage a point process or even an entire cell if those items stay in existence. The second problem is that a tool instance can be inadvertently destroyed along with its window without the user's pressing the Close button on the window. This can happen if the only reference to the tool instance is reused when a new instance is created.

Although it may seem tedious at first, the small effort of starting a tool development project with a pattern that addresses all these issues is quickly repaid. An initial version of the pattern begins by defining a new class that will be used to encapsulate the tool data and code.

A pattern for defining a template

The name we have chosen for this class is `BoltzmannParameters` (Listing 14.1). The overall style of the pattern is to declare the names of things used in the template first, 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."

Following the template definition are two statements that 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. The purpose of creating the `Graph` at this point is merely to cause the object to do something visible on the screen when it is created (Fig. 14.1).

As an aside we must note that typographic and other kinds of trivial errors are common, so it is a good idea to develop a program in stages, making only a few additions at a time. Building a tool requires many iterations of incrementally adding to and editing the `hoc` code that defines it and then launching NEURON to test it. In a windowed

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

desktop environment such as MSWindows, MacOS, or Xwindows, this cycle 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). For the sake of concreteness, we will assume that the hoc code for our tool will be saved to a file named bp.hoc.

```
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).

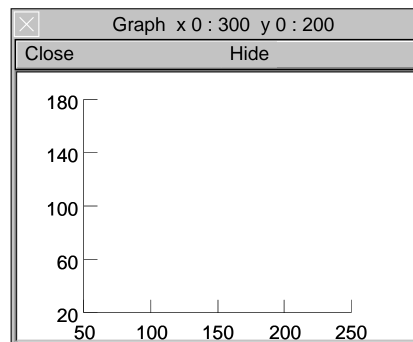

Fig. 14.1. The graph window generated by the code in Listing 14.1.

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. In most cases it doesn't make sense to call it again, so it is usually not declared public.

Enclosing the GUI tool in a single window

Unfortunately the relationship between the object BoltzmannParameters[0] and the window on the screen is easily shown to be inappropriate. Pressing the Close button destroys the window but doesn't get rid of the BoltzmannParameters[0] object

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

because the latter is still referenced by b. Furthermore, redeclaring b or making it reference another object will destroy BoltzmannParameters[0] because its reference count is reduced to 0. 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. We can augment our pattern so that this happens by enclosing the Graph in a VBox which references the tool itself, and making sure that this VBox is the only reference to the tool. The pattern now looks like:

```
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 (or horizontally if it is 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 with

```
add_miscellaneous("BoltzmannParameters", \
                  "makeBoltzmannParameters()")
```

Our init() procedure is collecting a lot of odds and ends; readability would be better served by separating the different idioms into procedures. Listing 14.3 shows the organization we prefer. Also note the changes to map(), which add the useful feature of

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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

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

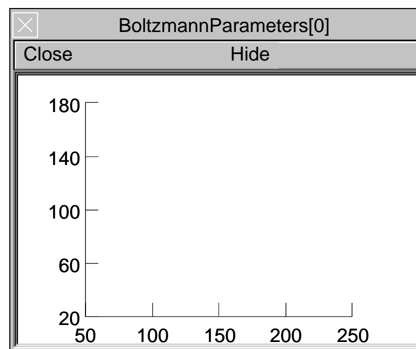

Fig. 14.2. Changes in the `map()` procedure (Listing 14.3) turn the window title into the name of the tool instance (compare with Fig. 14.1).

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 new `save()` procedure that can print executable `hoc` code lines 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

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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.

```
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() {
     sprint(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` generates a tool that can be saved and retrieved from a session file.

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 a tool is created *de novo*, it should appear in a default location with default initializations of its various user–

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

specifiable parameters. However, if it is being recreated from a session file, we want it to appear in the same location and with the same parameter values as when it was saved.

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 will make 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).

```
objectvar save_window_, rvp_
objectvar scene_vector_[3]
objectvar ocbox_, ocbox_list_, scene_, scene_list_
{ocbox_list_ = new List()  scene_list_ = new List()}

//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)
ocbox_.g = save_window_
save_window_.save_name("ocbox_.g")
}
ocbox_.map("BoltzmannParameters[0]", 403, 30, 314.25, 249.75)
}
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 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. The subsequent lines up to `save_window_` are exactly the strings printed by the `save()` procedure of `bp.hoc`. The first of these strings is

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

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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

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

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. The graph reference must be public because `object_pop()` returns to the previous context, which is normally the top level of the interpreter.

> "Top level" refers to the interpreter dealing with global variables rather than variables that are only visible inside templates. Strictly, I suppose 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 but inside a template, you may often find it useful to temporarily get back to the top level with `execute("statement")`.

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` procedure is called instead and we replace whatever title was used by the current authoritative instance name. Our `map()` procedure 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 an obvious first step is to define the function, declare the parameters we will be managing, and plot the function. Since any names we invent are nicely segregated so they can't conflict with names at the top level of the interpreter or names in any other tool, 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 the `init()` procedure.

Plotting the function raises the issue of the proper domain and the number of points. We could get sophisticated here with regard to plotting points at *x* values that nicely

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

follow the curve, but instead we will just plot 100 line segments with 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

```
c:\NRN\BIN\Neuron.exe: parse error in
  C:\WINDOWS\Desktop\b.hoc near line 20
    return 1/(1 + exp(-4*k*(d - $1))
                                     ^
```

because the Boltzmann function was missing a closing parenthesis. Parse errors mean something is wrong with the syntax of the statement, and most languages give better details about what might be invalid. The carat indicates the point where the parser failed, which is usually usually an important clue, but failure may not occur until several tokens after the actual mistake. One common parse error in apparently well–formed statements is the wrong type of a name, e.g. specifying a string where a scalar variable is required.

Adding the parenthesis and trying again, we find that `beginline` 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. Generally, run time messages are more pertinent to that actual problem than are syntax error messages, although logic errors can be very difficult to diagnose. Of no help at all is the line number, which refers to the last line that was parsed (`makeBoltzmannParameters()`).

```
begin_line not a public member of Graph
c:\NRN\BIN\Neuron.exe: Graph begin_line in
  C:\WINDOWS\Desktop\b.hoc near line 75
    makeBoltzmannParameters()
                              ^
        BoltzmannParameters[0].pl()
      BoltzmannParameters[0].map()
    BoltzmannParameters[0].init()
  makeBoltzmannParameters()
```

Another try gives

```
c:\NRN\BIN\Neuron.exe: exp result out of range in
C:\WINDOWS\Desktop\b.hoc near line 75
makeBoltzmannParameters()
                          ^
```

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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

```
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 empty, looking identical to Fig. 14.2. Invoking the Graph menu item "View = plot" turns this into a featureless grey rectangle (Fig. 14.3).

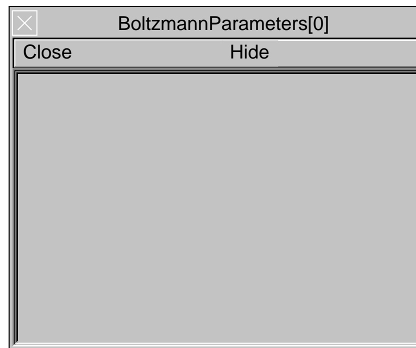

Fig. 14.3. After the numeric overflow problem is eliminated, the graph seems to be empty. However, View = plot turns the window contents grey, indicating that something else is wrong.

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()`, which 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 the line

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

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

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

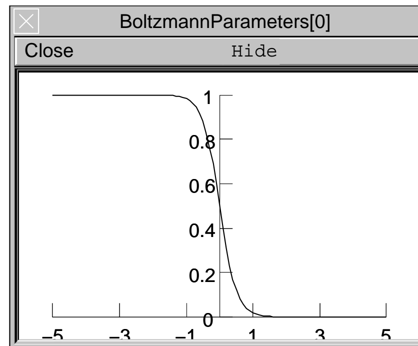

Fig. 14.4. The graph after fixing the range of *x* values and specifying appropriate dimensions.

Now the graph produced by this template looks pretty good, but it has a slope of −1. 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.

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 control, shift, 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
```

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

```
            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* on every mouse event so that the curve will pass 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 `b.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.

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, I'm very pleased––much better than another control point for *k* (Fig. 14.5 B).

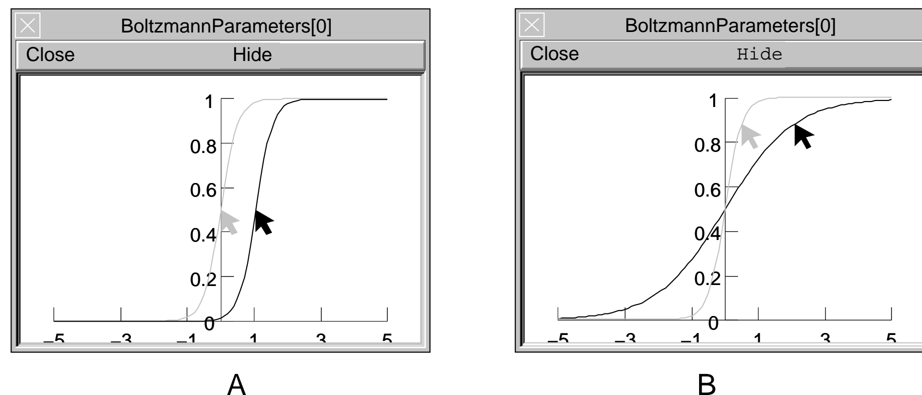

Fig. 14.5. A. The code in Listing 14.6 enables click and drag to shift the curve from side to side. B. Forcing `adjust_` to 2 allows us to test the use of this tool to set *k*.

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)
```

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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, plotting and handling events has been natural in model 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 model x and y scales. This requires revisions to `ptdist()`, where the information needed to transform between model and screen coordinates will be obtained by a sequence of calls to the `view_info()` function. I can certainly never remember the details of the various arguments for `view_info()` and need to refer to the help page.

Since each Graph can be displayed in more than one view simultaneously, the transformation between model 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 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 {
```

Note that a 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
   }
```


Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

```
    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.

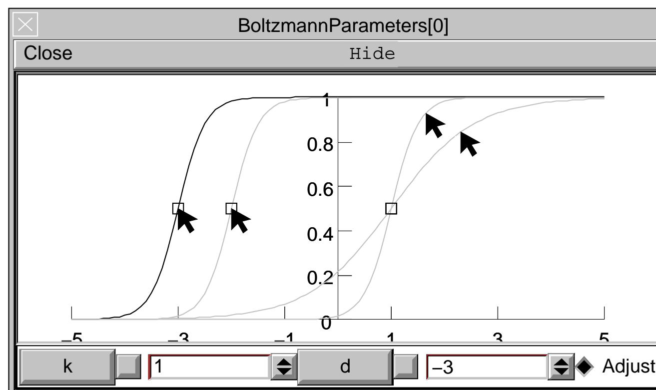

Fig. 14.6. Final appearance of the "Boltzmann Parameters" tool. The parameter values displayed are for the leftmost curve. Other curves with arrows suggest the process of selection and dragging the mouse.

The last touch is to change the `save()` procedure so that it saves the tool−specific state. Since model scene coordinates are saved by default instead of view coordinates, it is also 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
sprint(tstr, "{k=%g d=%g}", k, d)
box.save(tstr)
g.exec_menu("Scene=View")
```

Copyright © 2001−2003 N.T. Carnevale and M.L. Hines, all rights reserved

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)
      xpvalue("k", &k, 1, "pl()")
      xpvalue("d", &d, 1, "pl()")
      g.menu_tool("Adjust", "adjust")
    xpanel()
    g.exec_menu("Adjust")
    box.intercept(0)
  }

  proc map() {
    sprint(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
    sprint(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))
```

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

```
}

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
}
```

Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved

```
    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.

 Copyright © 2001–2003 N.T. Carnevale and M.L. Hines, all rights reserved