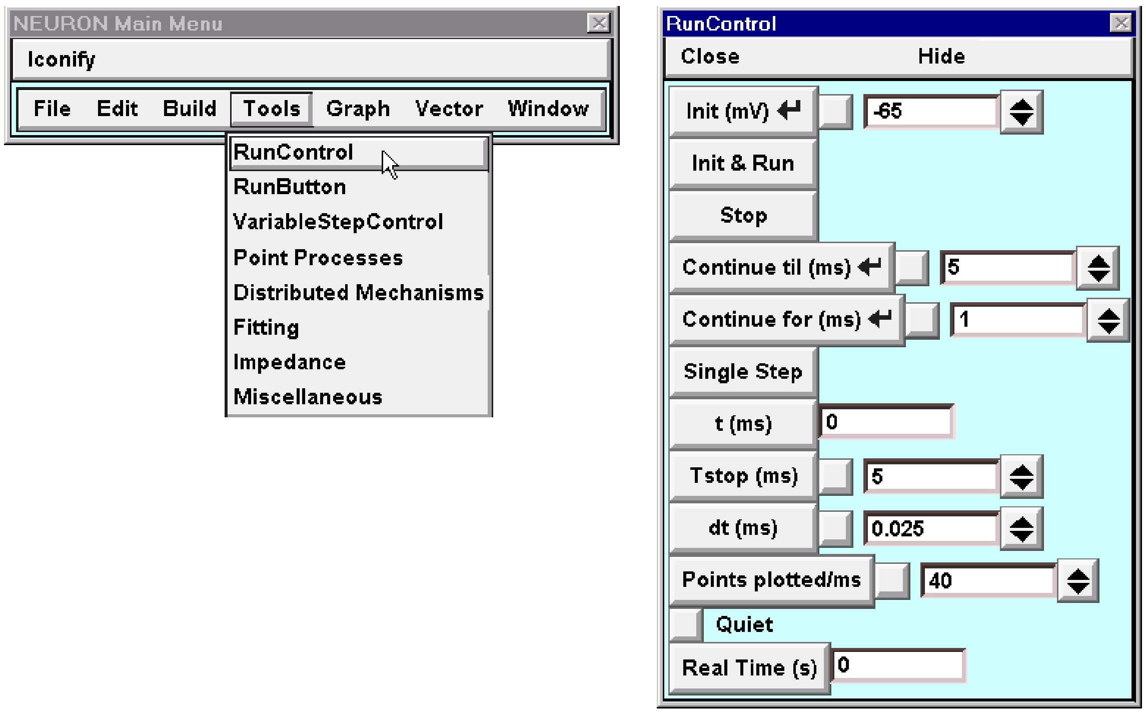
Chapter 7

How to control simulations

Simulation control with the GUI

The RunControl panel (Fig. 7.1 right) has several buttons and value editors (boxes that contain numbers) that provide a basic set of controls for initializing, starting, and stopping simulations. The actions listed in Table 7.1 are "defaults," i.e. the standard behavior of the tool. These actions are all customizable, because the RunControl works by calling procedures that are defined in `hoc` (see below) so you can always create a new procedure with the same name that substitutes for the default code.



Init (mV) -65
Continue til (ms) 5
Continue for (ms) 1
t (ms) 0
Tstop (ms) 5
dt (ms) 0.025
Points plotted/ms 40
Real Time (s) 0

Fig. 7.1. Left: NEURON Main Menu / Tools / RunControl brings up a panel with controls for running simulations. Right: The RunControl panel allows a great deal of control over the execution of simulations. See text for details.

In learning to use the RunControl panel it may help to keep in mind that adjacent controls have related functions. The three buttons at the top (Init, Init & Run, and Stop) perform the most common operations: initializing, starting, and stopping simulations. The next three (Continue til (ms), Continue for (ms), and Single Step) are particularly helpful for exploratory dissection of the time sequence of events in dynamically complex simulations.

Graphs created from the NEURON Main Menu respond appropriately to all of these controls. Init erases unsaved traces from graphs whose x axis shows time, and makes all other graphs (e.g. variables vs. anatomical location, phase plane plots) show initial values, whereas Init & Run, Continue til, Continue for, and Single Step cause graphs to be updated at intervals governed by Points plotted/ms and Quiet.

Table 7.1. Functions of the RunControl panel

Button	**Action**
Init (mV)	Sets time to 0, changes V_m throughout the model to the value displayed in the adjacent value editor, initializes ionic concentrations, and sets biophysical mechanisms (e.g. ionic conductances, pumps) to their corresponding steady−state values.
Init & Run	Same as the Init button, but then launches a simulation that runs until t equals Tstop (see below). Graphs constructed from the NEURON Main Menu are updated at a rate specified by Points plotted/ms and Quiet (see below).
Stop	Stops a simulation at the end of a step.
Continue til (ms)	Continues a simulation until t ≥ the value displayed in the adjacent value editor. Graphs are updated according to Points plotted/ms (see below).
Continue for (ms)	Continues a simulation for the amount of time displayed in the adjacent value editor. Graphs are updated according to Points plotted/ms (see below).
Single Step	Continues a simulation for one step and plots. A step is 1 / (Points plotted/ms) milliseconds and consists of 1 / (dt · Points plotted/ms) calls to `fadvance()`.
t (ms)	No action. The adjacent numeric field shows model time during the course of a simulation.
Tstop (ms)	No action. Adjacent field is used to specify stop time for Init & Run.
dt (ms)	No action. Adjacent field shows the fundamental integration time step used by `fadvance()`. Values entered into this field editor are automatically rounded down so that an integral multiple of `fadvance`s make up a Single Step.
Points plotted/ms	No action. Adjacent field is used to specify the number of times per millisecond at which graphs are updated. Notice that reducing dt does not by itself increase the number of points plotted. If 1 / (Points plotted/ms) is not an integral multiple of dt, then dt is rounded down to the nearest integral fraction of 1 / (Points plotted/ms).
Quiet	When checked, turns off graph updates during a simulation. This can speed things up considerably, e.g. when using the Multiple Run Fitter in the presence of a shape movie plot under MSWindows.
Real Time (s)	No action. Adjacent field shows a running display of computation time, with a resolution of 1 second.

The standard run system

The Init & Run button of the RunControl panel is probably the user's first contact with the standard run system. The standard run system for version 5.4 is implemented in the file

`nrn-5.4/share/lib/hoc/stdrun.hoc` (UNIX/Linux)

or

`c:\nrn54\lib\hoc\stdrun.hoc` (MSWindows)

which is interpreted with a number of other files when

```
load_file("nrngui.hoc")
```

is executed or the `nrngui` script or icon is launched. This system is a considerable elaboration over the minimal "oscilloscope level" simulation

```
proc run() {
   finitialize(-65)
   fcurrent()
   while (t < 5) {
      fadvance()
   }
}
```

which integrates a cell specification from `t` = 0 to `t` = 5 ms. The elaborations consist of various parameters and hooks for starting and stopping the simulation and obtaining information during the simulation run. Tools that involve the analysis of simulation results, e.g. optimization tools such as the Multiple Run Fitter, assume the existence of a `run()` procedure to carry out their evaluation of the difference between simulation result and data.

Understanding a few aspects of the standard run system is necessary in order to be able to write functions or objects that can work in the presence of this framework, or at least do not vitiate it. It is generally much easier to work with and reuse components of this system than attempt to recreate a great deal of existing functionality. Most users have come to count on existing features that allow plotting of any variable during a run, or easy switching between integration methods.

Be sure to load replacements of standard functions *after* the standard library. Otherwise the library version will overwrite your version instead of the other way around.

NEURON's standard run system was designed with the realization that research requirements are quite varied, so no generic implementation will suffice in all cases. Therefore an attempt was made to divide the run process into as many elements as seemed reasonable in order to make it easy for the user to replace any one of them. In most cases a replacement procedure requires only one or two specific code statements directed toward maintaining its standard function. The standard run system has proven to be usable without changes in a wide variety of situations, with the exception of the `init()` procedure for initialization (this is discussed extensively in **Chapter 8**). Nevertheless, certain problems can only be overcome by writing `hoc` code, or even low level C code, so it is helpful to have a tour of the sequence of events that leads to an actual time step advance.

An outline of the standard run system

The chain of execution follows the outline

```
run()
   stdinit()
      init()
         finitialize()
   continuerun() or steprun()
      step()
         advance()
            fadvance()
```

Each of these routines is very compact except for `continuerun()`, which employs rarely used graphical interface functions to optimize both simulation speed and graph line drawing so that the lines seem to be drawn in real time as the simulation progresses. Let’s start with `fadvance()` and work up from there.

fadvance()

For now it suffices that `fadvance()` integrates all equations of the system from `t` to `t+dt` and then replaces the value of `t` by `t+dt`; we will examine the details of this later. The value of `dt` is either set by the user when the default fixed step integration method is used, or chosen by the integrator if the variable step method is used.

advance()

The `advance()` routine

```
proc advance() {
   fadvance()
}
```

provides the hook for doing any desired calculations before and/or after each time step. With the default fixed step method, anything is allowed. That is, we may change any state or any parameter, including `dt`. Each advance takes place as though it starts from a new initial condition without any previous history. Things are not so easy with the variable time step methods. Although it is safe to evaluate any variable and save it in an array or write it to a file, changing a parameter or state is not allowed unless we execute `cvode.re_init()` after the change. This is because CVODE saves state and derivative information from previous steps and assumes that all coefficients and states are differentiable up to its current order of accuracy. Changing a parameter or state constitutes a new set of equations, which constitutes a new problem. The only way that time−varying parameters may be simulated with variable step methods is in the context of a model description.

step()

advance() is called by the step() procedure, which is implemented as

```
proc step() {local i
   if (using_cvode_) {
      advance()
   } else for i=1,nstep_steprun {
      advance()
   }
   Plot()
}
```

The idea behind this function is that numerical accuracy may require a smaller time step than needed for plotting. That is, the interval between plots (call it Dt) is an integral multiple of the underlying fadvance() time step dt. This integral multiple is calculated in a setdt() function which reduces dt if necessary to ensure that the Dt steps lie on a dt boundary. The RunControl panel has a field editor labeled Points plotted/ms which displays the value of the variable steps_per_ms. This value, along with dt, is used to calculate nstep_steprun and perhaps modify dt whenever either changes by calling setdt(). One can see that when CVODE is active, a step is just a single advance. At the end of a step, the Plot() procedure iterates over all the Graphs in the various plot lists that need to be updated during a simulation run. The purpose of these lists is detailed later in this chapter; adding to one of these lists an object that can carry out certain specific methods may be a more attractive way of recording results of a simulation run than replacing proc step(), since objects can automatically add and remove themselves from these lists.

steprun() and continuerun()

The step() procedure is called by the continuerun() and steprun() procedures. steprun() is

```
proc steprun() {
   step()
   flushPlot()
}
```

which implements the action for the Single Step button of the RunControl. It ensures that all the plot lists are flushed so that any deferred graph updates are performed.

continuerun() is called directly as an action by the Continue til and Continue for buttons in the RunControl. The actions are continuerun(runStopAt) and continuerun(t+runStopIn) respectively. continuerun() is quite complex, and it is doubtful that anyone will want to replace it with something more complicated. It takes a single argument which is the time at which the integration should stop.

Before every step(), continuerun() checks to see if the stoprun variable is nonzero; if so it immediately breaks out of its loop. continuerun() sets stoprun to 0 on entry; stoprun is set nonzero if the user presses the Stop button on the RunControl. stoprun is a global variable in C so it can be checked by any C or C++ class that can carry out multiple runs and needs to properly clean up and return, e.g. optimization routines such as the praxis optimizer. In designing any class that manages a family of runs, one must decide what to do when the user presses Stop. If `stoprun` becomes nonzero but the class ignores it, the current simulation run will end and the next run in the family will start.

`continuerun()` uses the stopwatch to count the seconds in a variable called `realtime` while it is executing, and this value is displayed in the Real Time field editor. The resolution of the stopwatch is one second, and after each second the plots are flushed with a special method that avoids redrawing the portions of lines that are already plotted, all field editors are updated if the values they are watching have changed, and any outstanding events are handled (otherwise pressing the Stop button would have no effect). Actually, to give more rapid response to events, the `doEvents()` function is called at every step for the first two seconds and less often after that to avoid overhead if steps are very fast.

"On one side hung a very large oil−painting so thoroughly besmoked, and every way defaced, that in the unequal cross−lights by which you viewed it, it was only by diligent study and a series of systematic visits to it, and careful inquiry of the neighbors, that you could any way arrive at an understanding of its purpose. Such unaccountable masses of shades and shadows, that at first you almost thought some ambitious young artist, in the time of the New England hags, had endeavored to delineate chaos bewitched. But by dint of much and earnest contemplation, and oft repeated ponderings, and especially by throwing open the little window towards the back of the entry, you at last come to the conclusion that such an idea, however wild, might not be altogether unwarranted."

When `continuerun()` has reached its stopping time, a full flush of all the plots is done. Plots are flushed at intermediate times only if the variable `stdrun_quiet` is 0; this variable is toggled by the Quiet checkbox in the RunControl. Drawing plots on the screen is expensive and considerable speedup can often be seen if plotting is deferred to the end of a run. However, it often seems worth the penalty to view the progress of a simulation.

run()

The `run()` procedure

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

is invoked as an action by the Init & Run button to initialize the system and integrate up to the value shown in the Tstop field editor of the RunControl. The initialization process is discussed at length in **Chapter 8**, but we should note that `stdinit()`

```
proc stdinit() {
   realtime=0
   startsw()
   setdt()
   init()
   initPlot()
}
```

calls `init()`

```
proc init() {
   finitialize(v_init)
   fcurrent()
}
```

which is generally the only function in the system that needs to be replaced in order to implement complex initialization strategies.

Details of `fadvance()`

The `fadvance()` function is implemented in `nrn.../src/nrnoc/fadvance.c`. In one form or another, `fadvance()` has always been the workhorse of the NEURON simulator, dating back to before NEURON's progenitor CABLE and even prior to the `hoc` interpreter, when all PDP8 FOCAL (FOrmula CALculator) functions had to begin with the letter `f`. One could easily do without an `finitialize()` function, since the interpreter overhead for computing steady states is small compared to the computational effort of taking `tstop/dt` steps to do a simulation. But fast integration is most naturally carried out in compiled code, which is on the order of a hundred times faster than the interpreter.

"From the chocks it hangs in a slight festoon over the bows, and is then passed inside the boat again; and some ten or twenty fathoms (called box−line) being coiled upon the box in the bows, it continues its way to the gunwale still a little further aft, and is then attached to the short−warp −−the rope which is immediately connected with the harpoon; but previous to that connexion, the short−warp goes through sundry mystifications too tedious to detail."

Extending NEURON's numerical methods and simulation domain has been an incremental process carried out over several years. It may help to understand the current structure of `fadvance()` if we first consider how it evolved. The order of additions was CVODE (variable order, variable time step integrator), `NetCon` (event delivery system), `LinearMechanism` (overlay of algebraic equations onto the Jacobian), and DASPK (differential algebraic solver). Each major increase in functionality reused as much of the existing functions and program structure as possible, but a few functions needed small changes so they could support both the old and new methods. These increases in functionality also had to be usable with the least amount of effort on the part of the user. For example, turning variable time step integration on or off can be done by clicking on a checkbox in the NEURON Main Menu / Tools / VariableStepControl panel.

Our dissection of `fadvance()` follows its evolution by

- reviewing the details of what happens during classical fixed time step integration, i.e. the fully implicit (backward Euler) and Crank−Nicholson methods. Topics examined include the strategies that account for NEURON's reputation for speed:
    1. exploiting the tree topology of the branched nerve equations. Tree topologies require exactly the same number of add/multiply/divide operations as a single unbranched cable.
    2. using a staggered time step to avoid Newton iterations of HH−like nonlinear channels. This gives the second order Crank−Nicholson method the same performance per time step as the first order implicit method.

3. using rate tables involving the value of `dt`. This optimizes the analytic integration of channel states by trivial assignment statements like `m=m+mexp*(minf-m)`.

- discussing the variable time step, variable order ordinary differential equation solvers.
- walking through the operation of the local variable time step method to learn how it works and how it handles discrete events.

Many of these items are closely related to each other, so we must occasionally mention later additions to complete the discussion of earlier ones.

The fixed step methods: implicit Euler and Crank−Nicholson

It is easiest to understand the reasons for the particular sequence of actions if we focus on the second order correct Crank−Nicholson method (CVODE is inactive and the global variable `secondorder` has the value of 2). Assume that, on entry to `fadvance()`, the value of `t` is `tentry`, the voltages are second order correct at `tentry`, and the gating states are second order correct at `tentry` + `dt`/2. This latter assumption may seem odd, but we will learn how it helps accelerate integration.

When the Crank−Nicholson method is chosen, the purpose of `fadvance()` is to integrate the voltages and states such that, on exit from `fadvance()`,

`t` = `tentry` + `dt` (call this `texit`)

`v` and concentrations are second order correct at `texit`

gating states are second order correct at `texit` + `dt`/2

and as a side effect

ionic currents are second order correct at `texit` − `dt`/2

Notice that these exit conditions satisfy the entry conditions for a subsequent call to `fadvance()`.

One might object that the entry assertions are not satisfied at `t` = 0 since the gating states are second order correct at time 0, not time `dt`/2. We'll discuss this in detail, however second order correctness refers to the integrated error over a specific time interval `Dt` as more and more `dt` steps are used. The local error over a single `dt` step for second order correctness is proportional to $\mathtt{dt}^3$ and for first order correctness it is $\mathtt{dt}^2$. So as long as d*state*/d*t* = 0 at *t* = 0, as it must be in the steady state, the error associated with using *state*(*t* = 0) as the value of *state*(*t* = `dt`/2) is itself proportional to $\mathtt{dt}^2$ and is a once−only error which does not accumulate for each `dt` time step. If non−steady state initializations are performed, then the gating states should be adjusted to their values according to *state* = *state* + d*state*/d*t* · `dt`/2.

For the default implicit and Crank−Nicholson methods, the sequence of operations carried out by `fadvance()` is

1. Check to see if any voltages or other variables that are sources for `NetCon` objects have reached threshold. Deliver any discrete events whose delivery time is earlier than `tentry+dt/2`. With fixed step methods, events necessarily lie on time step boundaries, so this certainly delivers all events outstanding at time `tentry`. The

function that carries this out (`NetCvode::deliver_net_events()` in `nrn.../src/nrncvode/netcvode.cpp`) first appends the value of `state` at `tentry` to the corresponding `Vector` according to the list defined by the `cvode.record(&state, vec, tvec)` statements. This list is most useful with the local variable step method; indeed, the `cvode.record` method is the only good way of keeping the proper association between local step state value and local `t`. Of course, `cvode.record` also works with the fixed step methods. As of version 5.4, `Vector`s that are played or recorded at specific times are handled as a sequence of discrete events.

2. When `Vector.play()` is treated as an interpolated (continuous) function, values are interpolated at time = `tentry+dt/2`. The syntax `Vector.play(&var)`, which has no specific time `Vector` or declared play interval, cannot be used by variable step methods and is therefore deprecated. However, in case you find it in old code, we mention that `Vector.play(&var)` makes `var` receive its value from the *next* `Vector` element; thus the first `fadvance()` after `finitialize()` will assign `Vector.x[1]` to `var`.

3. The matrix equation for voltage is set up with the global variable `t = tentry+dt/2`. This is done by calling the function `setup_tree_matrix()` in `nrn.../src/nrnoc/treeset.c`. Prior to version 5, NEURON was limited, as the names of this function and file imply, to coupled voltage equations with the topology of a tree structure, i.e. each voltage node had at most one parent node. This is not only well−matched to neuronal structure, but also has the attractive property that solution of linear equations with this structure by Gaussian elimination takes exactly the same number of arithmetic operations as if the equations had the topology of an unbranched cable with the same number of nodes. It is the tree structure which makes the simulation time proportional to the number of voltage nodes. Speed suffers when the topology is not equivalent to a tree, e.g. when gap junctions, linear circuits, or the extracellular mechanism is present. Completely general graph structures have a worst case Gaussian elimination time which is proportional to the cube of the number of voltage nodes (see **Chapter X**).

   The purpose of the `setup_tree_matrix()` function is to create the algebraic equation for each node. In abstract terms we are setting the problem up as a matrix equation in the form

$$\mathbf{M}\ \mathbf{v}(t_{entry} + \Delta t) = \text{r.h.s.} \qquad \text{Eq. 7.1a}$$

   ("r.h.s." = right hand side) for the implicit method, or

$$\mathbf{M}\ \mathbf{v}(t_{entry} + \frac{\Delta t}{2}) = \text{r.h.s.} \qquad \text{Eq. 7.1b}$$

   for the Crank−Nicholson method. Tree structures are very similar to tridiagonal cable equations. For unbranched cables the most straightforward description of the spatially discrete cable equation has a row structure

$$b_i v_{i-1} + d_i v_i + a_i v_{i+1} = \text{r.h.s.}_i \qquad \text{Eq. 7.2}$$

and each coefficient and variable in the row is kept in a node structure (*b*, *d*, and *a* are the subdiagonal ("*b*elow"), *d*iagonal, and supradiagonal ("*a*bove") elements of **M**). Generalization to a tree preserves the association of *b, d, v,* and r.h.s. in the node equation. The only change is that `Node.a` (see next paragraph) refers to the matrix element in the parent node equation.

Setup of the matrix equations begins by first checking a flag to see if any diameters or section lengths have changed, and if so, recalculating the two connection coefficients between a node and its parent. These connection coefficients are both stored in the node. `Node[i].b` is the resistance between node `i` and its parent divided by the area of the node. `Node[i].a` is the same thing but divided by the area of the parent node. Next, the *d* and r.h.s. elements of all nodes are set to zero in preparation for incrementally adding conductance and current contributions to them. The *a* and *b* elements of the matrix generally do not change during a simulation. Fortunately, they are not destroyed during Gaussian elimination and so only need to be computed when the morphology changes.

At this point the membrane current and conductance contributions to the node equations are added to r.h.s. and *d* respectively. This is done by calling the `nrn_cur` functions of every mechanism in every node (pointers to these functions are kept in the `memb_func[type].current` structure). These functions are the model description translation of the `BREAKPOINT` block. Recall that the most common usage of the `BREAKPOINT` block in a model description is to calculate channel currents from the values of `STATE` variables and membrane potential `v` (see **Chapter 9**). In the translation of a `BREAKPOINT` block, the `SOLVE` statement information, which tells how to integrate the `STATE` variables, is segregated into a `nrn_state` function (see step 6 below), and the remaining statements are used to construct a `nrn_current` function which takes voltage as an argument. The `nrn_current` function is called twice by the `nrn_cur` function, once with an argument of `v` + 0.001 and then with an argument of `v`, in order to calculate the numerical derivative `di/dv` as well as the current. The `nrn_cur` function then adds the `di/dv` value to the diagonal element `Node.d` and the value of `-i` to the right hand side element `Node.rhs`. The form of this expression follows from the current conservation equation evaluated at $t + \Delta t$

$$C \frac{\Delta v_i}{\Delta t} + \frac{di_i}{dv_i} \Delta v_i - \sum_j \frac{\Delta v_j - \Delta v_i}{area_i \, r_{ij}} = - i_i(v_i(t)) + \sum_j \frac{v_j - v_i}{area_i \, r_{ij}} \qquad \text{Eq. 7.3}$$

where

$$i_i(v_i(t+\Delta t)) = i_i(v_i(t)) + \Delta v_i \frac{di_i}{dv_i} \qquad \text{Eq. 7.4}$$

All terms that are proportional to Δv go into the matrix (left) side of Eq. 7.1, and all constant terms or product terms of $v(t)$ go into the right hand side. If Δv_j refers to the parent of node i, the coefficient $1/area_i\ r_{ij}$ is the ith node's b element (see Eq. 7.2); if Δv_j refers to a child, the coefficient is the child node's a element.

4. The `nrn_solve()` function in `nrn.../src/nrnoc/solve.c` is called to solve the voltage node equations. Normally these equations are tree−structured, which allows use of triangularization and back substitution functions that are specifically crafted to minimize pointer arithmetic overhead by taking advantage of the details of our `Node` structure in `nrn.../src/nrnoc/section.h`. This step executes approximately twice as fast as the more general sparse matrix Gaussian elimination package necessary for non−tree structures. However this has less significance than it appears since Gaussian elimination of tree structures takes much less than half the time required to set up equations containing channel currents. On exit from `nrn_solve()` the r.h.s. field of the `Node` structures contains the values of Δv.

Nowadays, voltage clamp models are best implemented as linear mechanisms. Voltage and current states in such a model are computed simultaneously with the membrane potential, so the issues associated with staggered time steps do not arise.

If `secondorder` is 2 then the currents are updated with a call to `second_order_current`, which uses `di_ion/dv` along with Δv to compute the second order correct ionic currents at `tentry+dt`/2. Therefore when `fadvance()` returns and `t` is `tentry + dt`, the ionic currents are second order correct at `t − dt`/2. Note that individual currents associated with particular channel mechanisms and available to the interpreter as `ASSIGNED` variables are not updated to be second order correct. That is, individual model description current variables are approximated by `g(texit – dt/2)*(v(texit-dt) – erev)`. Without special attention to this problem, model descriptions of voltage clamp currents that are appropriate for the internal use made of them during `fadvance()` would be complete nonsense when plotted, since they do not take into account the large change between `v(texit-dt)` and `v(texit)`. For this reason, particularly stiff models such as voltage clamps are careful to recalculate the current variable within the block called by the `BREAKPOINT`'s `SOLVE` statement (see step 6 below), which occurs when the voltage values are at `texit`.

For fixed step methods, one should always compare plots of individual model current and conductance variables with their time courses computed with smaller `dt`. In some cases it may be useful for plotting to introduce a `FUNCTION` into the channel model which uses the present values of `t`, `v`, and `STATE`s to return the consistent first order values of those currents. Equivalently, one could call `fcurrent()` on return from `fadvance()` (`fcurrent()` carries out step 3) to reevaluate the currents and conductances at the present values of `t`, `v`, and `STATE`s.

With the variable step methods (see below), all variables have their appropriate values at `texit`. One of the most significant benefits of the variable time step methods is the ease of plotting current and conductance variables at the accuracy of the underlying computation.

5. The voltages are updated using the equation `v` = `v` + r.h.s. for the implicit method and `v` = `v` + 2 r.h.s. for the Crank−Nicholson method. The global variable `t` is set to `tentry` + `dt`.
6. `nonvint()` is called, which integrates all states EXCEPT the voltages. This is done by executing the `nrn_state` function for every mechanism in every segment of every section (pointers to these functions are kept in the `memb_func[type].state` structure). These functions are the model description translation of the `SOLVE` statement in the `BREAKPOINT` block. Since `v` is now at `tentry` + `dt`, or the midpoint of the integration interval from `tentry` + `dt` + `dt`/2, second order correct integration schemes that treat `v` as a constant in the integration interval remain second order correct. Specifically, the analytic integration of Hodgkin−Huxley−like channel gating states, e.g.

$$m(t+\frac{\Delta t}{2}) = m(t-\frac{\Delta t}{2}) + (1-e^{-\Delta t\,/\,tau\,(v(t))})\,(m_\infty(v(t)) - m(t-\frac{\Delta t}{2})) \qquad \text{Eq. 7.5}$$

where $v(t)$ is assumed constant, is second order correct for smooth functions of v. It should be remembered, however, that the calculation of m is only first order correct with the fixed step implicit method since the value of v itself is only first order correct.

When fixed step methods were used exclusively, it was common practice to factor the integration statement into the form

```
m = m + mexp(v)*(minf(v)-m)
```

where `mexp` and `minf` were calculated with fast interpolated table lookup. However, since the `mexp` table is dependent on the value of `dt`, this no longer works with variable step methods. Of course, `minf` and `mtau` could still be stored in tables, but the speedup is marginal, and in these days of fast floating point processors, `minf` and `mtau` have to be quite complicated to justify the use of tables.

7. All the variables being recorded due to `Vector.record(&variable)` statements (i.e. without an associated sampling interval or `Vector` of recording times) are stored in the `Vector` elements associated with time `tentry` + `dt`. Starting with version 5.4, sampling times specified by a sampling interval or `Vector` of recording times are handled by the discrete event system.

Adaptive integrators

Our chief aim here is to see how adaptive integration operates in the context of a simulation, and in particular how it fits in with the event delivery system. Mathematical aspects of adaptive integration are discussed more thoroughly in **Chapter 4**.

Adaptive integrators adjust the time step and order of integration so that the local error for each state is less than a user−specified tolerance. For a given `dt` they are three times slower than the fixed step methods, because calculating the local error involves a lot of overhead and it is no longer is possible to use `dt`−dependent rate tables or avoid Newton iterations. However, the time step can be so large during interspike intervals that total run time is often almost an order of magnitude faster than with fixed step methods yielding the same accuracy. From the user’s perspective, a potentially more important advantage of adaptive integration is that it eliminates the need for trial and error adjustments of `dt` in order to achieve satisfactory accuracy; instead, one merely specifies the local step accuracy and the integrator does the rest.

In models that involve asynchronous events, adaptive integration can improve simulation accuracy by guaranteeing that all events occur at their specified times rather than being forced to a `dt` step boundary (see below). Furthermore, variables are consistent at time `t` on return from `fadvance()`, so there is no need to wonder whether to plot a variable at `t`, `t+dt`/2, or `t−dt`/2 (see step 4 under **The fixed step methods: implicit Euler and Crank−Nicholson** above).

Adaptive integration was first added to NEURON starting with CVODE [Cohen, 1994 #512][Cohen, 1996 #722] for global time steps in version 4.0, and this was extended to local time steps in version 4.1. The original CVODE required modifications in order to work with models that involved `at_time()` events, which were used to implement abrupt changes of a parameter or a state. A strategy for dealing with an event that occurs at t_{event} is to stop integration at t_{event}, change the parameters or states that are modified by the event, calculate a new initial condition at t_{event}, and then resume integration. However, the CVODE integrator had no provision for stopping at a specified time, so it needed custom revisions. DASPK [Brown, 1994 #675], which was subsequently added to deal with models in which some states are determined by algebraic equations (e.g. extracellular fields or linear circuit elements), had a specifiable stop time beyond which the integrator would not proceed, so it had a very different way of handling `at_time()`. It would have been nice if DASPK could simply have replaced CVODE, but DASPK did not directly support the interpolation operation needed by the local step method, and it has even more overhead per step than CVODE. Therefore a significant amount of code was required to provide the logical machinery that would make all these different pieces of the NEURON simulation environment work properly with each other, while at the same time allowing users to easily switch between the various integrators. The later addition of an delivery system to NEURON greatly increased the complexity of the code that ties all these pieces together.

This complexity has been much reduced in the most recent releases of NEURON by replacing CVODE and DASPK with CVODES and IDA of the SUNDIALS package (available from `http://www.llnl.gov/CASC/sundials/`). CVODES [Hindmarsh, 2002 #721] is similar to CVODE but accepts a `tstop` beyond which the solution will not proceed, and IDA is a new Initial value Differential Algebraic solver version of DASPK which now does support the interpolation operation. However, for historical reasons the class that is used to manage adaptive integration in NEURON is called `CVode`, and in this book we often use the term "CVODE" as a generic reference to any of NEURON's adaptive integrators.

The normal CVODE integration step consists of a prediction followed by a correction. Generating the prediction involves an evaluation of $\boldsymbol{f}(\mathbf{y}, t)$ (see Eq. 4.28a and 4.29a) which consumes most of the computational effort in an integration step. When CVODE returns, all STATEs have the correct values at the new time, but the ASSIGNED variables (which include currents) still have their "predicted" values. Correcting the ASSIGNED variables requires another evaluation of $\boldsymbol{f}(\mathbf{y}, t)$, but this nearly doubles the total computational overhead per integration step. For many purposes the uncorrected values are sufficiently accurate, and tightening the error tolerance takes care of most cases when it is not. Future releases of NEURON will apply the correction by default but may offer users the option of disabling the ASSIGNED variable correction with the extra call to $\boldsymbol{f}(\mathbf{y}, t)$ after a CVODE step.

Adaptive integrators and discrete events

Now we are ready to consider what constitutes an `fadvance()` when adaptive integration is used. We will focus on local variable step integration, in which an independent CVODE method is created for each cell. The process of global time step integration has only one CVODE method for the entire model, and is just a degenerate case of what happens with local time steps.

In local time step integration there is a queue of event times and a queue of cell times. The event times are the times at which events are to be delivered, and the cell times are the current times of each cell in the model. When `fadvance()` is called, it checks these queues and deals with whichever is earliest: the earliest event or the earliest cell. If there is a tie, the event is handled before the cell is. After an event is handled, it is discarded. When a cell is handled, its old time is discarded and it is assigned a new time that is put back into the cell time queue.

Each cell has three variables, called `t0`, `t_`, and `tn`, that are related to the progress of the simulation in time. `t_` is the current time of the cell, and it determines the position of the cell in the cell queue; the significance of `t0` and `tn` will become clear shortly. When `fadvance()` is called, it can take one of the following three actions, and when it returns these variables are left in one of the configurations shown in Figure 7.2. For the purpose of illustration, we assume that before `fadvance()` is called, the cell starts with `t0`, `t_`, and `tn` as depicted in the top row of this figure.

1. **Initialize**: perform an initialization at some time `t` and then return. The cell's `STATE`s and currents are consistent at `t`, and its `t0`, `t_`, and `tn` are all equal to `t`.
2. **Advance**: perform a normal integration step to some new time `t` and then return. This involves computing values for the `STATE`s and currents at some new time `t`, updating `t0` to the old `tn`, and making `t_` and `tn` equal to the new `t`.

3. **Interpolate**: return just before the time $\texttt{t}_{\texttt{event}}$ of the next event. On exit from fadvance(), t_ lies between t0 and tn with a value equal to $\texttt{t}_{\texttt{event}}$. STATE values at t_ are calculated from their values at tn, t0, and prior solution points according to CVODE's interpolation formulas (this is much less costly than a numeric integration step). If an integration step carries tn past the time of an event, or if a new event arrives with $\texttt{t}_{\texttt{event}} < \texttt{t\_}$, interpolation will be applied so that t_ retreats to $\texttt{t}_{\texttt{event}}$. However, a cell can't retreat to a time earlier than its t0. If there are multiple cells, the largest t0 is the "least event time," i.e. the time before which *no* cell can retreat.

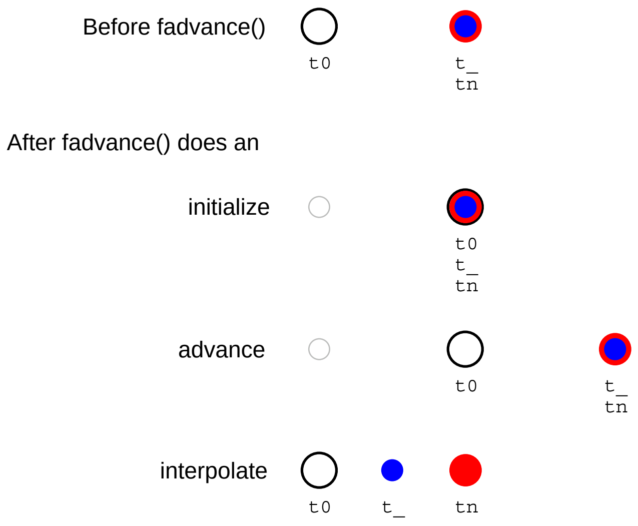


Fig. 7.2. After fadvance() returns, the relative positions of t0 (black open circle), t_ (blue dot), and tn (red filled circle) in time depends on whether fadvance() performed an initialization, a normal integration step, or an interpolation to just before the next event. The small grey circle after initialize and advance marks the former location of t0. Time increases toward the right in each row.

Note that the STATE and current values at the new tn are "tentative" because if there is an event in the [t0, tn] interval, a new initialization may be required that forces the solution into a new trajectory. The values at t0 are "real" in the sense that a cell cannot retreat past t0.

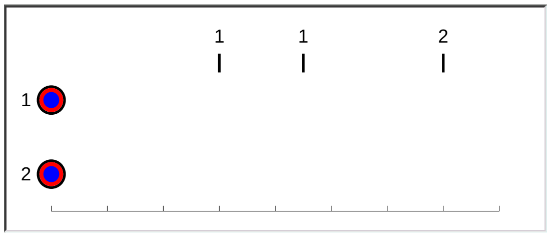
If multiple events occur at the same time, they are all handled. If more than one of these requires an initialization, the initialization is deferred until after all simultaneous events are handled. Thus if there are 4 events at the same time and 3 of them require initialization, each event will be handled but there will be only one initialization, which is performed after all four have been handled.

To make this more concrete, let's walk through a hypothetical simulation of a small network model using the local variable time step method. This model has two neurons called 1 and 2. A NetCon delivers events to an excitatory synapse on cell 1, and cell 1 projects via another NetCon to a synapse on cell 2. In the following discussion the "step" number refers to how many times we have called `fadvance()`, the "action" is what `fadvance()` did, and the "outcome" is a figure that shows the relative position in time of events and each cell's `t0`, `t_`, and `tn`.

Step, action, and outcome

Comments

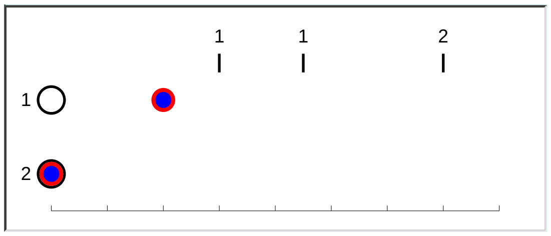
0. Initialize the model



Initializing the model causes both cells to be initialized to `t` = 0 ms; notice that `t0` = `t_` = `tn` = 0 ms. Also three events are placed in the event queue, two for cell 1 and one for cell 2, at the indicated times.

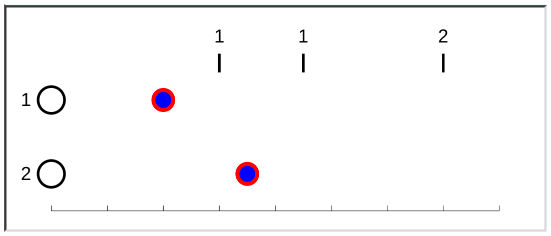
There are no events at `t` = 0 ms . . .

1. Advance cell 1



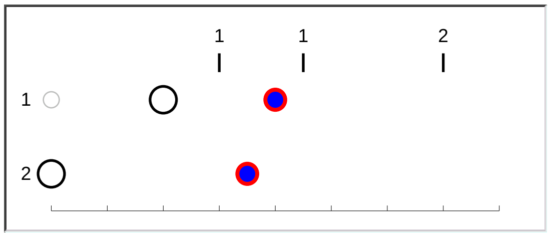
. . . so the first `fadvance()` *advances* one of the cells. For the sake of illustration, we'll say it advances cell 1. This makes 2 the earliest cell.

2. Advance cell 2



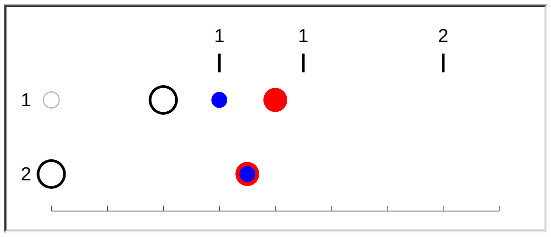
Cell 2's `t_` and `tn` move past the earliest event, but that's OK because the event isn't for cell 2. Cell 1 is now earliest.

3. Interpolate cell 1 −−
integration phase



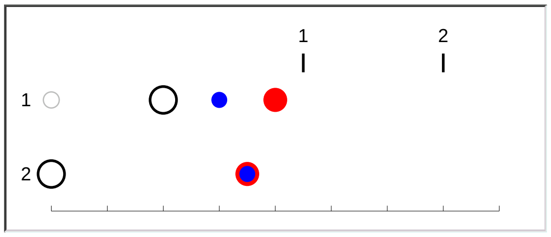
Cell 2's `t_` and `tn` move to a new time. Notice how `t0` follows behind `tn`, jumping from its prior location (marked by the small "ghost" circle) to the prior location of `tn`. But also notice that `t_` has moved past an event for cell 1. Before `fadvance()` can return . . .

3 continued: Interpolate cell 1 ––
retreat phase



. . . `t_` must retreat to the event time, and cell 1's `STATE`s at `t_` are then calculated by interpolation. `fadvance()` may now return, and we are ready to handle the event.

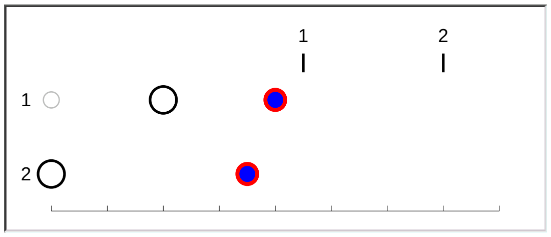
Handle the event



Handling the event removes it from the event queue.

Cell 1 is still earliest. For the sake of illustration, let's say the event we just handled didn't do anything to cell 1 that forces initialization . . .

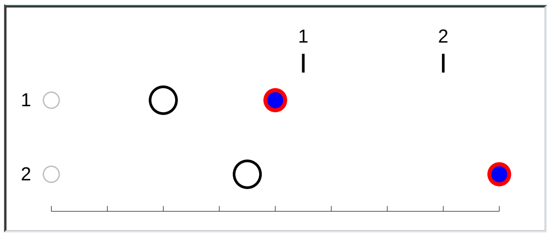
4. Interpolate cell 1



. . . so cell 1's trajectory isn't affected. There are no events between its current time `t_` and `tn`, so `t_` can be moved right up to `tn`, as shown here. Technically speaking this is an "interpolation" but no real calculations are involved.

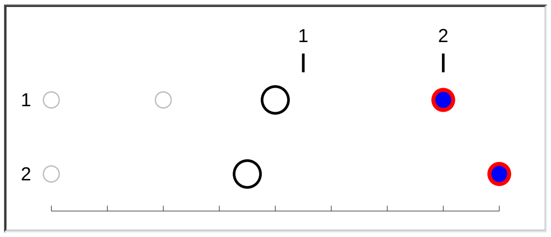
The earliest cell is now cell 2.

5. Advance cell 2



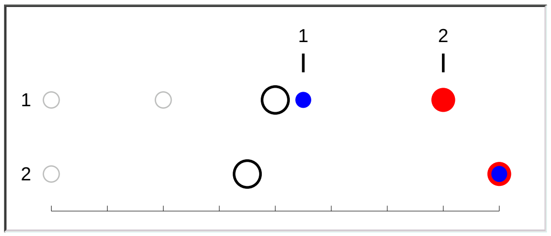
Although cell 2's `t_` and `tn` have moved past several events, the earliest event doesn't pertain to it, so `fadvance()` only does an advance rather than an interpolate.

6. Interpolate cell 1 ––
integration phase



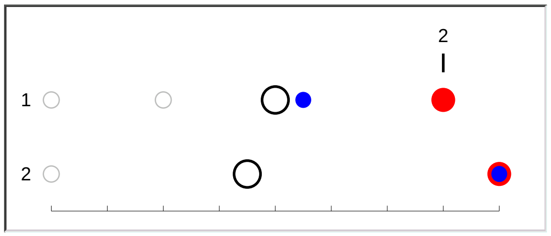
We have seen this before.

6 continued: Interpolate cell 1 −−
retreat phase



It is now time to deal with the event . . .

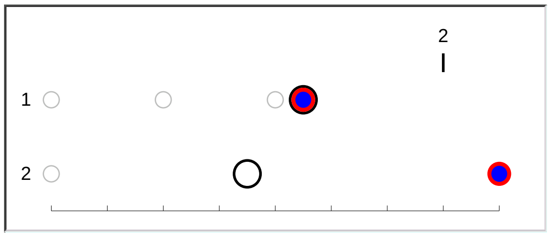
Handle the event



. . . which removes it from the queue.

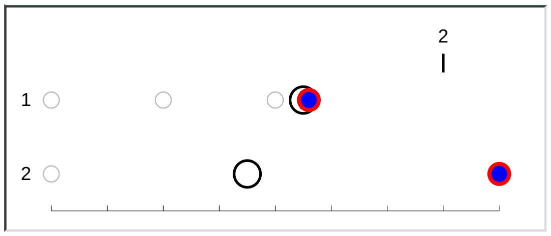
And it’s also time to introduce a little excitement. Unlike the first event, which didn’t affect cell 1’s trajectory, we’ll stipulate that this one was delivered to the excitatory synaptic mechanism on cell 1 by a `NetCon` with a strong positive weight, causing an abrupt change in one of the that mechanism’s parameters. This means the next `fadvance()` has to *initialize* cell 1.

7. Initialize cell 1



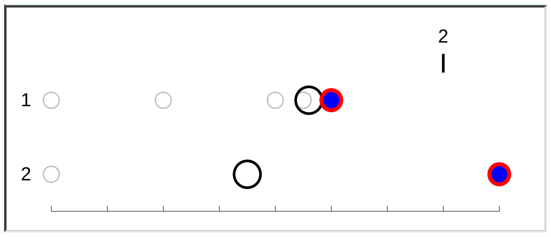
Notice that cell 1’s `t0`, `t_` and `tn` are exactly at the handled event time.

8. Advance cell 1

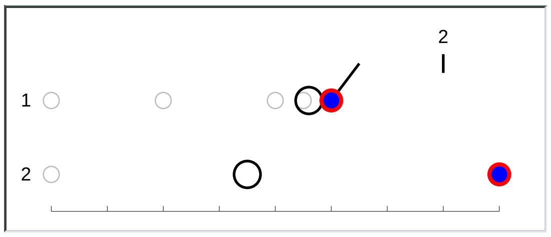


The strong synaptic input drives this cell toward firing threshold. Since its membrane potential is changing rapidly, `fadvance()` must perform short advances to satisfy the error criterion.

9. Advance cell 1

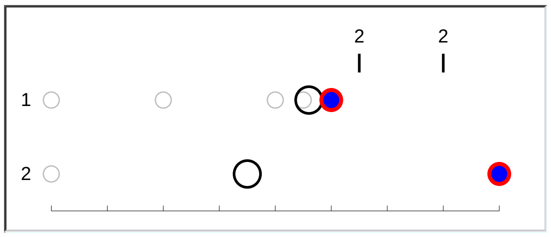


Cell 1 generates a spike event . . .



That last `fadvance()` took cell 1 over the threshold of the `NetCon` that monitors its membrane potential.

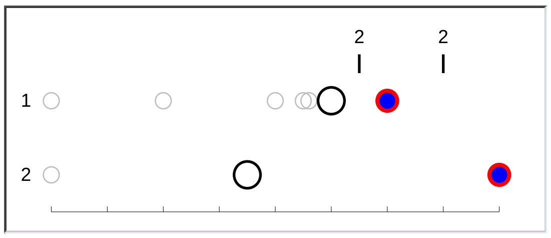
. . . which is inserted into the event queue



The spike event will be delivered to the synapse on cell 2 at the new time indicated in this figure.

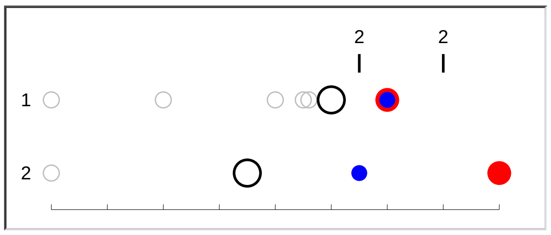
Cell 1 is the earliest cell now . . .

10. Advance cell 1



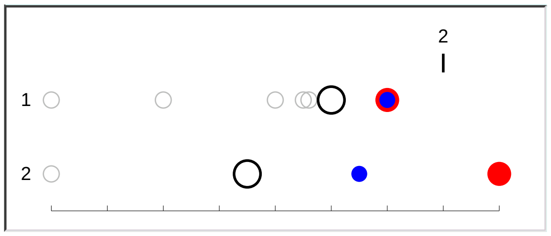
. . . and again. But it has moved past the spike event for cell 2, so that event becomes the next thing to deal with.

11. Interpolate cell 2



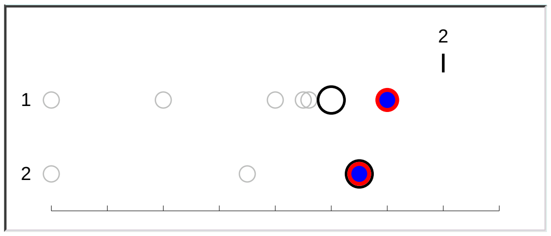
Cell 2 must retreat to the time of its event.

Handle the event



The event disappears from the event queue.

12. Initialize cell 2



The event caused an abrupt change in a variable in cell 2's synapse, so `fadvance()` must initialize this cell.

From the standpoint of users, this is all easier done than said, thanks to the behind−the−scenes coordination of adaptive integration and discrete events in NEURON.

Incorporating `Graphs` and new objects into the plotting system

Objects that need to be notified at every step of a simulation are appended to one of six lists. The first four lists are referenced by `graphList[n_graph_lists]` and their normal contents are `Graph` objects that plot variables requested by each `Graph`'s `addexpr` or `addvar` statement. Variables are plotted as line drawings in which the abscissa is related to `t` and the ordinate is the magnitude of the variable. `Graphs` are added to these four lists when one of the buttons of the NEURON Main Menu / Graph menu titled Voltage axis, Current axis, State axis, and Phase Plane is pressed.

For	each variable is plotted vs.
`graphList[0]`	`t`
`graphList[1]`	`t-0.5*dt`
`graphList[2]`	`t+0.5*dt`
`graphList[3]`	an arbitrary function of `t` called an x−expression

The most useful of these lists is `graphList[0]`, which is recommended for all line drawings. `graphList[1]` and `[2]` are useful only to provide second order correct plots of ionic currents and state variables, respectively, when the Crank−Nicholson method has been selected through the variable `secondorder=2`. The offset is meaningless when the default first order method is used (`secondorder=1`) because first order accuracy holds at all instants in the interval `[t-0.5*dt, t+0.5*dt]`. When the variable time step methods are chosen, all variables are computed at the same `t` so the offset is 0 and the `[1]` and `[2]` `graphList` lists are identical to `graphList[0]`.

The remaining two lists whose object elements are notified at every step are called `flush_list` and `fast_flush_list`. The first is for `Graph`s that plot `Vector`s that may change every time step. These do the `Vector movies` and `Space Plots` requested from a `Shape plot`. The `fast_flush_list` is for `Shape plots` or `Hinton plots` in which it is not necessary to redraw an entire cell or pattern because only a few rectangles change color during each step.

Plots are initialized by a call from `stdinit()` to `initPlot()`. The `initPlot()` procedure first removes any objects in the graph or flush lists for which there is no view on the screen by checking the return value of the `view_count()` method of the objects, and then calls the `begin()` method for all objects in the graph lists. Finally, it calls the `Plot()` and `flushPlot()` procedures to get the right things plotted at `t=0`.

The `Plot()` procedure is called at the end of each step. `Plot()` calls `plot(t)` for the `graphList` objects (actually the previously discussed offsets may be used for `graphList[1]` and `[2]`). If `stdrun_quiet` is 0, `Plot()` also calls `begin()` and `flush()` methods for items in the `flush_list` so that any `Vector` plots are updated. Lastly it calls the `fast_flush()` method for each item in the `fast_flush_list` so that any color changes are seen on the screen.

During `continuerun()`, the `fast_flushPlot()` procedure is called once at every second of simulation time and the `flushPlot()` procedure is called at the end. `fast_flushPlot()` calls the `fast_flush()` method for each item in the four `graphList` lists. This special call is very efficient for time plots because it erases and redraws only the portion of the lines that accumulated since the last `fast_flush`. Otherwise, damaging a small part of a line entails damaging the entire bounding box of the line, which implies damaging all the lines that intersect the bounding box, which ends up damaging the entire canvas and consequently requires erasing and redrawing everything on the canvas. `flushPlot()` calls the `flush()` method for each item in all six lists, which ends up redrawing everything in every canvas. While this is expensive, the screen accurately reflects exactly the internal data structures of the lines and shapes.

A `Graph` object constructed by the user with

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

can be added to the standard run system with

```
graphList[0].append(g)
```

or perhaps even better with

```
addplot(g, 0)
```

since the latter will also set the abscissa to range from 0 to `tstop` (and the vertical axis from −1 to 1). Also, since the methods called on a `graphList` are `begin()`, `plot(t)`, `view_count()`, `fast_flush()`, `flush()`, and `size(x0, x1, y0, y1)`, any object that implements these functions, even as stubs, can be appended to `graphList[0]` in order to carry out calculations during a run. The SpikePlot of the

NetGui tools is implemented in just this way. This is an example of how the `hoc` interpreter provides a poor man's version of polymorphism; more information about object−oriented programming in `hoc` is presented in **Chapter 13**.