

Chapter 8

How to initialize simulations

In most cases, initialization basically means the assignment of values at time $t = 0$ for membrane potential, gating states, and ionic concentrations at every spatial position in the model. A model is properly initialized when the Init & Run button produces exactly the same results, regardless of previous history of the simulation. Of course we assume that model parameters have not changed between runs, and that any random variable has been re-initialized with the same seed so that it produces the same sequence of "random" numbers. For simulations of networks, initialization also includes specifying which events are in transit to their destinations at time 0 (i.e. generated, at least conceptually, at $t \leq 0$ for delivery at $t \geq 0$). If linear circuits are included in the model, then all voltages across capacitors and the internal states of operational amplifiers must be specified as part of the initialization. Complex models often have complex recording and analysis methods, perhaps involving counters and vectors, which may need to be initialized. Models described by kinetic schemes require that each of the chemical kinetic states be initialized to some concentration.

State variables and STATES

In rough mathematical terms, if a system consists of n first order differential equations, then initialization consists in specifying the starting values of n variables. For the Hodgkin–Huxley membrane patch (only one compartment), these equations have the form

$$\frac{dv}{dt} = f_1(m, h, n, v) \quad \text{Eq. 8.1a–d}$$

$$\frac{dm}{dt} = f_2(m, v)$$

$$\frac{dh}{dt} = f_3(h, v)$$

$$\frac{dn}{dt} = f_4(n, v)$$

so that, knowing the value of each variable at time t , we can specify the slope of each variable at time t . We have already seen (**Chapter 7**) that integration of these equations is an iterative process in which the purpose of an individual integration step (`fadvance()`)

is to carry the system from time t to time $t + \Delta t$ using more or less sophisticated equations of the form

$$v(t + \Delta t) = v(t) + \Delta t \frac{dv(t^*)}{dt} \quad \text{Eq. 8.2}$$

$$m(t + \Delta t) = m(t) + \Delta t \frac{dm(t^*)}{dt}$$

...

where the sophistication is in the choice of a value of t^* somewhere between t and $t + \Delta t$. However, regardless of the integration method, the iterative process cannot begin without choosing starting values for v , m , h , and n . This choice is arbitrary over the domain of the variables ($-\infty < v < \infty$, $0 \leq m \leq 1$, ...), but once the initial v , m , h , and n are chosen, all auxiliary variables (e.g. conductances, currents, d/dt terms) at that instant of time are determined, and the equations determine the trajectories of each variable forever after. The actual evaluation of these auxiliary variables is normally done with assignment statements, such as

```
gna = gnabar*m*m*m*h
ina = gna*(v - ena)
```

This is why the model description language NMODL designates `gna` and `ina` as ASSIGNED variables, as opposed to the gating states, `m`, `h`, and `n`, which are termed STATE variables, i.e. the dependent variables of differential equations.

Unfortunately, over time an abuse of notation has evolved in which STATE also refers to any variable that is an unknown quantity in a set of simultaneous algebraic equations, and ASSIGNED refers to any variable that is not a STATE or a PARAMETER (PARAMETERS can be meaningfully set by the user as constants throughout the simulation, e.g. `gnabar`). Nowadays, within a single model description, STATE just specifies which variables are the dependent variables of KINETIC schemes, algebraic equations in LINEAR and NONLINEAR blocks, or differential equations in DERIVATIVE blocks. Generally the number of STATES in a model description is equal to the number of equations. Thus, locally in a model description, the membrane potential `v` is never a dependent variable (there is no equation that solves for its value in the model description) and it cannot be meaningfully thought of as a user-specified value, so it is declared in the model descriptions as an ASSIGNED variable, despite its obvious status as a state variable at the level of the entire simulation. This abuse of the STATE concept also occurs in the linear circuit, where the potential at every node of the circuit is an unknown to be solved and therefore a STATE. However you should be aware that a resistive network does not add any differential equation to the system (although it adds algebraic equations) and so those additional dependent variables do not strictly need to be initialized.

While state variables may be assigned any values whatever during initialization, in practice only a few general types of initialization are carried out. These are based on analogous experimental methods for preparing a system for stimulation, e.g. letting the system rest without experimental perturbation, or using a voltage clamp or constant

injected current to hold the system at a defined membrane potential. The idea is that the system should reach an unchanging steady state independent of previous history. It is from this steady state that the simulation begins at time $t = 0$. When there is no steady state, as for oscillating or chaotic systems, whatever initialization is ultimately chosen will need to be saved in order to be able to reproduce the simulation. More complicated initializations involve finding parameters that meet certain conditions, such as what value of some parameter or set of parameters yields a steady state with a desired potential. Some initial conditions may not be physically realizable by any possible manipulations of membrane potential. For example, with hh channels the h gating state has a steady state of 1 at large hyperpolarized potentials and the n gating state has a steady state of 1 at large depolarized potentials. It would therefore be impossible to reach a condition of $h = 1$ and $n = 1$ by controlling only voltage.

Basic initialization in NEURON: `finitialize()`

Basic initialization in NEURON is accomplished with the

```
finitialize(v_init)
```

function, which is defined in `nrn-5.4/src/nrnoc/fadvance.c` (UNIX/Linux). This carries out several actions.

1. t is set to 0 and the event queue is cleared (undelivered events from the previous run are thrown away).
2. Variables that receive a random stream (the list defined by `Random.play` statements) are set to values picked from the appropriate random distributions.
3. All internal structures that depend on topology and geometry are updated and chosen solvers are made ready.
4. The controller for `Vector.play` variables is initialized. The controller makes use of the discrete event system for `Vector.play` specifications that define transfer times for a step function in terms of dt or a time `Vector`.

Discrete events at time $t = 0$ (i.e. the appropriate `Vector.play` events) are delivered.

5. If `finitialize()` was called with an argument `v_init`, the membrane potential `v` in every compartment is set to the value `v_init` with a statement equivalent to

```
forall for (x) v(x) = v_init
```

6. The `INITIAL` block of every inserted mechanism in every segment of every section is called. This includes the `INITIAL` blocks of `POINT_PROCESS` mechanisms (discussed later in this chapter under **INITIAL blocks in NMODL**). The order in which mechanisms are initialized depends on whether any mechanism has a `USEION` statement or `WRITES` an ion concentration.

Ion initialization is performed first, including calculation of reversal potentials. Then mechanisms that `WRITE` an ion concentration are initialized. This necessitates

recalculation of the reversal potentials for those ions. Finally, all other mechanism INITIAL blocks are called.

Apart from these constraints, the call order of user-defined mechanisms is currently defined by the alphabetic list of mod file names or the order of the mod file arguments to nrnivmodl (or mknrndll). However one should avoid sequence-dependent INITIAL blocks. Thus if the INITIAL block of one mechanism needs the values of variables another mechanism, the values of the latter should be assigned before finitialize() is executed.

If extracellular mechanisms exist, their vext states are initialized to 0 before any other mechanism is initialized. Therefore, for every mechanism that computes an ELECTRODE_CURRENT, v_init refers to both the internal potential and the membrane potential.

INITIAL blocks are discussed in further detail below.

7. LinearMechanism states, if any, are initialized.
8. Network connections are initialized. This means that the INITIAL block inside any NET_RECEIVE block that is a target of a NetCon object is called to initialize the states of the NetCon object.
9. The INITIAL blocks may have initiated net_send events whose delay is 0. These events are delivered to the corresponding NET_RECEIVE blocks.
10. If fixed step integration is being used, all mechanism BREAKPOINT blocks are called (essentially equivalent to a call to fcurrent()) in order to initialize all assigned variables (conductances and currents) based on the initial STATE and membrane voltage.

If any variable time step method is active, then those integrators are initialized. In this case, if you desire to change any state variable (here "state variable" means variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, and ion concentrations in accumulation models) after finitialize() is called, you will need to call ccode.re_init() to notify the variable step methods that their copy of the initial states needs to be updated. Note that initialization of the differential-algebraic solver IDA consists of two very short ($\Delta t = 10^{-6}$ ms) fully implicit time steps in order to ensure the validity of

$$f(\mathbf{y}', \mathbf{y}) = \mathbf{0}.$$

11. Vector recording of variables using the list defined by the ccode.record(&state, vector) statements is initialized. This is the only meaningful way at this time to retrieve results from a simulation that uses the local variable time step method, since t values on return from a sequence of fadvance() calls are not monotonic and only a small fraction of states (the states in only one cell) is integrated on a single fadvance() call.
12. Vectors that record a variable (the list defined by Vector.record statements) record the value in Vector.x[0], if t = 0 is a requested time for recording.

Default initialization in the standard run library: `stdinit()` and `init()`

Default initialization for the standard run library takes effect when you enter a new value for `v_init` into the field editor next to the `Init` button, or when you press either the `Init` or the `Init & Run`, button in the `RunControl` panel. These buttons do not call the `init()` procedure directly but instead execute a procedure called `stdinit()` which has the implementation

```
proc stdinit() {
    realtime=0 // "run time" in seconds
    startsw() // initialize run time stopwatch
    setdt()
    init()
    initPlot()
}
```

`setdt()` ensures (by reducing `dt`, if necessary) that the points plotted fall on time step boundaries, i.e. that $1/(\text{steps_per_ms} * dt)$ is an integer. The `initPlot()` procedure begins each plotted line at $t = 0$ with the proper `y` value.

The default `init()` procedure itself is

```
proc init() {
    finitialize(v_init)
    // User-specified customizations go here.
    // If this invalidates the initialization of
    // variable time step integration and vector recording,
    // uncomment the following code.
    /*
    if (cnode.active()) {
        cnode.re_init()
    } else {
        fcurrent()
    }
    frecord_init()
    */
}
```

Custom initialization is generally accomplished by inserting additional statements after the call to `finitialize()`. These statements often have the effect of changing one or more state variables, i.e. variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, and ion concentrations in accumulation models. This invalidates the initialization of the variable time step integrator, making it necessary to call `cnode.re_init()` to notify the variable step integrator that its copy of the initial states needs to be updated. If instead fixed step integration is being used, the `fcurrent()` function should be called; this makes the values of conductances and currents consistent with the new states. Changing state variables after calling `finitialize()` can also cause incorrect values to be stored as the first element of recorded vectors. Adding `freload_init()` to the end of `init()` prevents this.

INITIAL blocks in NMODL

INITIAL blocks for channel models generally set the channel gating states to their steady state values with respect to the present value of v . HH-like models do this easily and explicitly by calculating the voltage sensitive alpha and beta rates for each gating state and using the two state formula for the steady state, e.g.

```
PROCEDURE rates(v(mv)) {
  minf = alpha(v)/(alpha(v) + beta(v))
  . . .
}
```

and then

```
INITIAL {
  rates(v)
  m = minf
  . . .
}
```

When channel models are described by kinetic schemes, it is common to calculate the steady states with the idiom

```
INITIAL {
  SOLVE scheme METHOD steadystate
}
```

where `scheme` is the name of a KINETIC block. To place this in an almost complete setting, consider a three state potassium channel with two closed states and an open state

```
NEURON {
  USEION k READ ek WRITE ik
}

STATE { c1 c2 o }

INITIAL {
  SOLVE scheme METHOD steadystate
}

BREAKPOINT {
  SOLVE scheme METHOD sparse
  ik = gbar*o*(v - ek)
}

KINETIC scheme {
  rates(v) : calculate the 4 k rates.
  ~ c1 <-> c2 (k12, k21)
  ~ c2 <-> o ( k2o, ko2)
}
```

where we have only left out some of the minor variable declarations and the voltage-sensitive implementation of the rates procedure. As mentioned above in **Default initialization in the standard run library: `stdinit()` and `init()`**, when initialization has been customized so that states are changed after `finitialize()` has been called, it is generally useful to call the `fcurrent()` function so that the values of all conductances and currents consistently with the newly initialized states. In particular this will call the BREAKPOINT block (twice, in order to compute the Jacobian (di/dv))

elements for the voltage matrix equation) for all mechanisms in all segments, and on return the ionic currents such as `ina`, `ik`, and `ica` will contain the corresponding net ionic currents through each segment.

Default initialization of STATES

In model descriptions, a default initialization of the STATES of the model occurs just prior to the execution of the INITIAL block. However this default initialization is rarely useful and one should always explicitly implement an INITIAL block. If the name of a STATE variable is `state`, then there is also an implicitly declared parameter called `state0`. The default value of `state0` is specified either in the PARAMETER block

```
PARAMETER {
  state0 = 1
}
```

or implicitly in the STATE declaration with the syntax

```
STATE {
  state START 1
}
```

If a default value for `state0` is not declared by the user, the default value will be 0. `state0` is not accessible from the interpreter unless it is explicitly mentioned in the GLOBAL or RANGE list, as in

```
NEURON {
  GLOBAL m0
  RANGE h0
}
```

in which every `m` will be set to the single global `m0` value during initialization, while `h` will be set to the possibly spatially-varying `h0` values. Clarity will be served if, in the use of the `state0` idiom, you explicitly use an INITIAL block of the form

```
INITIAL {
  m = m0
  h = h0
  n = n0
}
```

Ion concentrations and equilibrium potentials

Prior to version 4.1, model descriptions could not initialize concentrations, or at least it was very cumbersome to do so. Each ion type was managed by its own separate ion mechanism, which in turn initialized the ionic concentration adjacent to the membrane according to global variables. That is, if `cai` and `cao` were integrated by a model declaring

```
USEION ca READ ica WRITE cai, cao
```

there would also be an automatically created mechanism called `ca_ion`, so that the initial values of `cai` and `cao` were set globally to the values of `cai0_ca_ion` and `cao0_ca_ion`, respectively (see *Initializing concentrations in hoc* below).

The reason that model mechanisms were not allowed to specify ion variables (or other potentially shared variables such as `celsius`) was that confusion could result if more than one mechanism at the same location set the same variables to different values. This of course causes confusion of a different kind in that, when an ion variable such as `ena` is clearly assigned a value as a `PARAMETER` in a model, that assignment has no effect. In this case a warning occurs as in the mechanism

Since calcium currents, concentrations, and equilibrium potentials are managed by the `ca_ion` mechanism, one might reasonably ask why we can refer to the short names `ica`, `cai`, `cao` and `eca`, rather than the longer forms that include the suffix `_ion`, i.e. `ica_ca_ion` etc.. The answer is that there is unlikely to be any confusion about the meaning of `ica`, `cai`, ... so we might as well take advantage of the convenience of using these short names.

```
NEURON {
  SUFFIX test
  USEION na READ ena
}

PARAMETER {
  ena = 25 (mV)
}
```

When this model is translated

```
$ nrnivmodl test.mod
Translating test.mod into test.c
Warning: Default 25 of PARAMETER ena will be ignored and set by NEURON.
```

and use of the model in NEURON shows that the value of `ena` is that defined by the `na_ion` mechanism itself instead of the test model.

```
$ nrngui
Additional mechanisms from files
test.mod
.
.
.
oc>create soma
oc>access soma
oc>insert test
oc>ena
      50
```

If we add the initialization

```
INITIAL {
  printf("ena was %g\n", ena)
  ena = 30
  printf("we think we changed it to %g\n", ena)
}
```

to `test.mod`, then we can see that it still does not have an effect.

```
oc>finitialize(-65)
ena was 50
we think we changed it to 30
      1
oc>ena
      50
```

It is perhaps not a good idea to invite diners into the kitchen, but the reason for this can be seen from the careful hiding of the ion variables by making local copies of them in the C code generated by the `nocmodl` translator. Translation of the `INITIAL` block into a model-specific `initmodel` function is, except for some trivial boiler plate, an almost verbatim copy. However, `finitialize()` calls this indirectly via the model generic `nrn_init` function, which can be seen in all its gory detail in the C file output from `nocmodl test.mod`:

```

/*****/
static nrn_init(_count, _nodes, _data, _pdata, _type_ignore)
    int _count, _type_ignore; Node** _nodes; double** _data; Datum** _pdata;
{ int _ix; double _v;
  _p = _data; _ppvar = _pdata;

  #if _CRAY
  #pragma _CRI ivdep
  #endif
  for (_ix = 0; _ix < _count; ++_ix) {
    _v = _nodes[_ix]->_v;
    v = _v;
    ena = _ion_ena;
    initmodel(_ix);
  }
}
/*****/

```

It suffices merely to call attention to the statement

```
ena = _ion_ena;
```

which shows the difference between the local copy of `ena` and the pointer to the ion variable itself. The model description can touch only the local copy and is unable to change the value referenced by `_ion_ena`. Some old model descriptions circumvented this hiding by using the actual reference to the ion mechanism variables in the `INITIAL` block (from a knowledge of the translation implementation), but that was always considered an absolutely last resort.

This hands-off policy for ion variables has recently been relaxed for the case of models that `WRITE` ion concentrations, but only if the concentration is declared to be a `STATE` *and* the concentration is initialized explicitly in an `INITIAL` block. It is meaningless for more than one model at the same location to specify the same concentrations, and an error is generated if multiple models `WRITE` the same concentration variable at the same location.

If we try this mechanism

```

NEURON {
  SUFFIX test2
  USEION na WRITE nai
  RANGE nai0
}

PARAMETER {
  nai0 = 20 (milli/liter)
}

STATE {
  nai (milli/liter)
}

```

```
INITIAL {
  nai = nai0
}
```

we get this result

```
oc>create soma
oc>access soma
oc>insert test2
oc>nai
    10
oc>finitiaLize(-65)
    1
oc>nai
    20
oc>nai0_test2 = 30
oc>finitiaLize(-65)
    1
oc>nai
    30
```

If the INITIAL block is not present, the nai0_test2 starting value will have no effect.

Initializing concentrations in hoc

The best way to initialize concentrations depends on the design and intended use of the model. One must ask whether the concentration is supposed to start at the same value in all sections where the mechanism has been inserted, or should it be nonuniform from the outset?

Take the case of a mechanism that WRITES an ion concentration. Such a mechanism has an associated global variable that can be used to initialize the concentration to the same value in each section where the mechanism exists. These global variables have default values for na, k and ca that are "reasonable" but probably incorrect for any specific preparation. The default concentrations for ion names created by the user are 1 mM; these should be assigned correct values in hoc. A subsequent call to finitiaLize() will use this to initialize the ionic concentration.

The name of the global variable is formed from the name of the ion that the mechanism uses and the concentration that it WRITES. For example, suppose we have a mechanism kext that implements extracellular potassium accumulation as described by Frankenhaeuser and Hodgkin [Frankenhaeuser, 1956 #307]). The kext mechanism WRITES ko, so the corresponding global variable is ko0_k_ion. The sequence of instructions

```
ko0_k_ion = 10      // seawater, 4 x default value (2.5)
ki0_k_ion = 4*54.4 // 4 x default value, preserves ek
finitiaLize(v_init) // v_init is the starting Vm
```

will set ko to 10 mM and ki to 217.6 mM in every segment that has the kext mechanism.

What if one or more sections of the model are supposed to have different initial concentrations? For these particular sections the ion_style() function would be used to assert that the global variable is not to be used to initialize the concentration for this

particular ion. A complete discussion of `ion_style()`, its arguments, and its actions is contained in NEURON's help system, but we will consider one specific example here. Let's say we have inserted `kext` into section `dend`. Then the numeric arguments in the statement

```
dend ion_style("k_ion", 3, 2, 1, 1, 0)
```

would have the following effects on the `kext` mechanism in the `dend` section (in sequence): treat `ko` as a STATE variable; treat `ek` as an ASSIGNED variable; on call to `finitialize()` use the Nernst equation to compute `ek` from the concentrations; compute `ek` from the concentrations on every call to `fadvance()`; do *not* use `ko0_k_ion` or `ki0_k_ion` to set the initial values of `ko` and `ki`. The proper initialization would now be to set `ko` and `ki` explicitly for this section, e.g.

```
ko0 k_ion = 10 // all sections start with ko = 10 mM
dend {ko = 5 ki = 2*54.4} // . . . except dend
finitialize(v_init)
```

Examples of custom initializations

Initializing to a particular "resting potential"

Perhaps the most trivial custom initialization is to force the initialized voltage to be the resting potential. Returning our consideration to initialization of the HH membrane compartment,

```
finitialize(-65)
```

will indeed set the voltage to -65 mV, and `m`, `h`, and `n` will be in steady state relative to that voltage. However, this must be considered analogous to a voltage clamp initialization since the sum of all the currents may not be 0 at this potential, i.e. -65 mV may not be the resting potential. For this reason it is common to adjust the leakage equilibrium potential so that the resting potential is precisely -65 mV.

This can be done with a user-defined `init()` procedure based on the idea that total membrane current at steady state must be 0. For our single compartment HH model, this means that

$$0 = i_{na} + i_k + g_{l_hh} * (v - e_{l_hh})$$

so our custom `init()` is

```
proc init() {
  finitialize(-65)
  el_hh = (ina + ik + gl_hh*v)/gl_hh
  if (cnode.active()) {
    cnode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Remember to load user-defined versions of functions or procedures that are part of the standard library, such as `init()`, *after* the standard library. Otherwise, the user-defined version will be overwritten.

The `cvode.re_init()` call is not essential here since states have not been changed, but it is still good practice since it will update the calculation of all the $dstate/dt$ (note that now dv/dt should be 0 as a consequence of the change in `e1_hh`) as well as internally make a call to `fcurrent()` (which is necessary because changing `e1_hh` requires a recalculation of `il_hh`).

Calculating the value of leakage equilibrium potential in order to realize a specific resting potential is not fail-safe in the sense that the resultant value of `e1_hh` may be very large and out of its physiological range—after all, `g1_hh` may be a very small quantity. It may sometimes be better to introduce a constant current mechanism and set its value so that

$$0 = i_{na} + i_k + i_{ca} + i_{\text{constant}}$$

holds at the desired resting potential. An example of such a mechanism is

```

: constant current for custom initialization

NEURON {
  SUFFIX constant
  NONSPECIFIC_CURRENT i
  RANGE i, ic
}

UNITS {
  (mA) = (milliamp)
}

PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}

```

and the corresponding custom `init()` would be

```

proc init() {
  finitialize(-65)
  ic_constant = -(ina + ik + il_hh)
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}

```

Before moving on to the next example, we should mention that testing is required to verify that the system is stable at the desired `v_init`, i.e. that the system returns to `v_init` after small perturbations.

Initializing to steady state

In **Chapter 4** we mentioned that NEURON's default integrator uses the implicit Euler method, which can find the steady state of a linear system in a single step if the integration step size is large compared to the longest system time constant. Implicit Euler can also find the steady state of many nonlinear systems, but it may be necessary to perform several iterations with large Δt . An `init()` that takes advantage of this fact is

```

proc init() { local dtsav, temp
  finitialize(v_init)
  t = -1e10
  dtsav = dt
  dt = 1e9
  // if cvode is on, turn it off to do large fixed step
  temp = cvode.active()
  if (temp!=0) { cvode.active(0) }
  while (t<-1e9) {
    fadvance()
  }
  // restore cvode if necessary
  if (temp!=0) { cvode.active(1) }
  dt = dtsav
  t = 0
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}

```

This first performs a preliminary "voltage clamp" initialization to `v_init`. Then it sets time to a very large negative value (to prevent triggering point processes and other events) and integrates over several steps with a large fixed Δt so that the system can reach steady state. The procedure wraps up by returning Δt to its original value, setting `t` back to 0, and, if necessary, reactivating the variable step integrator. The last few statements are the familiar re-initialization of `cvode` or invocation of `fcurrent()`, followed by initialization of vector recording.

This initialization strategy generally works well, but there are circumstances in which it may fail. Active transport mechanisms can be troublesome with fixed time step integration if Δt is large, because even a small pump rate may produce a negative concentration. To see a more mundane example of instability with large Δt , construct a single compartment model that has the `hh` mechanism. With the default `hh` parameters, and in the absence of any injected current, this is quite stable even for huge values of Δt (e.g. 10^5 ms). Now reduce `gnabar_hh` to 0, increase Δt to 100 ms, and watch what happens over the course of 5000 ms. The result is an oscillation whose peak-to-peak amplitude gradually increases to ~ 10 mV. It would be all too easy to miss such oscillations when using steady-state initialization with large steps. This underscores the need for careful testing of any initialization strategy, since in a sense all of them work "behind the scenes."

Initializing to a desired state

Suppose the end of some run is to serve as the initial condition for subsequent runs; this is a particularly useful strategy for dealing with models that oscillate or otherwise lack a "resting" state. We can save all the states with

```
objref svstate, f
svstate = new SaveState()
svstate.save()
```

The binary state information can be saved for use in later neuron sessions with

```
f = new File("states.dat")
svstate.fwrite(f)
```

and future sessions can read the file into the `SaveState` object with

```
objref svstate, f
svstate = new SaveState()
f = new File("states.dat")
svstate.fread(f)
```

Whether or not the state information comes from a `svstate.save()` in this session or was read from a file, we only have to make a minor change to `init()` in order to use that information to initialize the system:

```
proc init() {
  finitialize(v_init)
  svstate.restore()
  t = 0 // t is one of the "states"
  if (cnode.active()) {
    cnode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Initializing by changing the model

Occasionally the aim is to bring a model to an initial condition that it would never reach on its own. This can be a particular challenge if the model involves several interacting nonlinear processes, making it difficult or impossible to know in advance what values the states should have. Such problems can sometimes be circumvented by changing the parameters of the model so that initialization reaches the desired state, and then restoring the original parameters of the model.

As a specific example, consider a conceptual model of the regulation of the calcium concentration in a thin intracellular compartment ("shell") adjacent to the cell membrane (Fig. 8.1). Calcium (Ca^{+2}) can enter or leave the shell in one of three ways: by diffusion between the shell and the core of the cell, by active transport via a membrane-bound pump, or as a result of non-pump calcium current I_{Ca} (i.e. transmembrane calcium flux not produced by the pump). For the sake of simplicity, we will assume that Ca_{core} and Ca_o ($[\text{Ca}^{+2}]$ in the core and extracellular solution) are constant. However, the problems

$$\text{calcium current} \quad \text{Ca}_{\text{shell}} \xleftarrow{1/2 F \text{ vol}} -I_{\text{Ca}} \quad \text{Eq. 8.3d}$$

where τ is the time constant for equilibration of Ca^{+2} between the shell and the core, F is Faraday's constant, and vol is the volume of the shell.

The NMODL code that implements this mechanism is

```

NEURON {
  SUFFIX capmp
  USEION ca READ cao, ica, cai WRITE cai, ica
  RANGE tau, width, cacore, ica, pump0
}

UNITS {
  (um)      = (micron)
  (molar)   = (1/liter)
  (mM)      = (millimolar)
  (uM)      = (micromolar)
  (mA)      = (milliamp)
  (mol)     = (1)
  FARADAY   = (faraday) (coulomb)
}

PARAMETER {
  width = 0.1      (um)
  tau = 1          (ms) : corresponds to D = 2e-7 cm2/s
  : D for Ca in water is 6e-6 cm2/s, i.e. 30x faster
  k1 = 5e8        (/mM-s)
  k2 = 0.25e6     (/s)
  k3 = 0.5e3      (/s)
  k4 = 5e0        (/mM-s)
  cacore = 0.1    (uM)
  pump0 = 3e-14   (mol/cm2)
}

ASSIGNED {
  cao      (mM) : on the order of 10 mM
  cai      (mM) : on the order of 0.001 mM
  ica      (mA/cm2)
  ica_pmp  (mA/cm2)
  ica_pmp_last (mA/cm2)
}

STATE {
  cashell  (uM)      <1e-6>
  pump     (mol/cm2) <1e-16>
  capump   (mol/cm2) <1e-16>
}

INITIAL {
  ica = 0
  ica_pmp = 0
  ica_pmp_last = 0
  SOLVE pmp STEADYSTATE sparse
}

```

```

BREAKPOINT {
  SOLVE pmp METHOD sparse
  ica_pmp_last = ica_pmp
  ica = ica_pmp
}

KINETIC pmp {
  : volume/unit surface area has dimensions of um
  : area/unit surface area is dimensionless
  COMPARTMENT width {cashell}
  COMPARTMENT (1e13) {capump}
  COMPARTMENT 1(um) {cacore}
  COMPARTMENT (1e3)*1(um) {cao}
  CONSERVE pump + capump = (1e13)*pump0
  ~ cacore <-> cashell (width/tau, width/tau)
  ~ cashell + pump <-> capump ((1e7)*k1, (1e10)*k2)
  ~ capump <-> cao + pump ((1e10)*k3, (1e10)*k4)
  ica_pmp = (1e-7)*2*FARADAY*(f_flux - b_flux)

  : ica_pmp is the "new" value, but cashell must be
  : computed using the "old" value, i.e. ica_pmp_last
  ~ cashell << (-(ica - ica_pmp_last)/(2*FARADAY)*(1e7))

  cai = (0.001)*cashell
}

```

Initializing the mechanism

For the sake of convenience we will assume that our model cell has only one section called `soma`, and that `soma` is the default section. Also suppose that we have already assigned the desired value of Ca_{shell} to a parameter we will call `ca_init`, e.g. with a statement of the form `ca_init = somevalue`. Our problem is how to ensure that initialization makes `cashell_capmp` take on the value of `ca_init`.

As a naive first stab at this problem, we might try changing the `init()` procedure like this

```

proc init() {
  cashell_capmp = ca_init
  f_initialize(v_init)
}

```

i.e. by inserting a line that sets the desired value of Ca_{shell} before calling `f_initialize()`. To see whether this has the desired effect, we need only to run a simulation and examine the time course of Ca_{shell} and the pump current $I_{Ca_{pump}}$. This quickly shows that, no matter what value we first assign to `cashell_capmp`, `f_initialize()` drives Ca_{shell} and $I_{Ca_{pump}}$ to the same steady state levels (Fig. 8.2). We might have anticipated this result, because it is what steady state initialization is supposed to do. If Ca_{shell} is too high, the excess calcium will diffuse into the core or be pumped out of the cell until Ca_{shell} returns to the steady state value. On the other hand, if Ca_{shell} is too low, calcium will diffuse into the shell from the core, and the pump will slow or may even reverse, until Ca_{shell} comes

back to the steady state value. Thus, regardless of how we perturb Ca_{shell} , steady state initialization always brings the model back to the same condition.

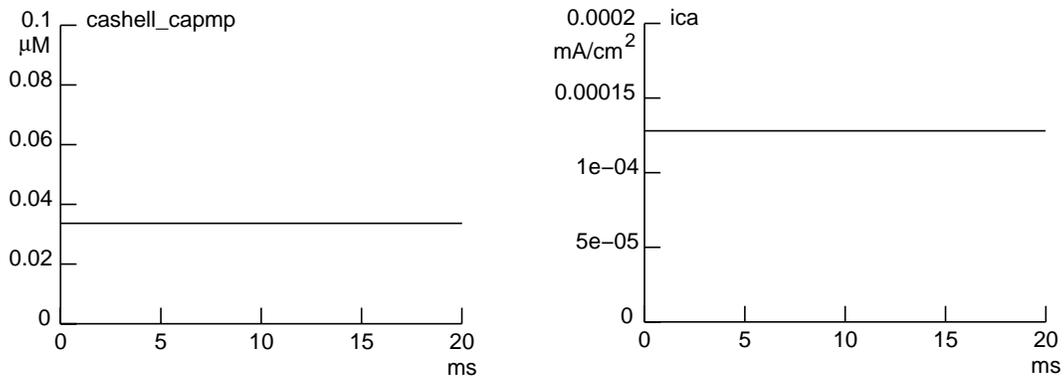


Fig. 8.2. Default initialization after setting `cashell_capmp` to $0.1 \mu\text{M}$ leaves Ca_{shell} (left) and $I_{\text{Ca}_{\text{pump}}}$ (right) at their steady-state levels of $\sim 0.034 \mu\text{M}$ and $\sim 1.3 \times 10^{-4} \text{ mA/cm}^2$, respectively.

For our second attempt we try calling `finitialize()` first, and then setting the desired value of Ca_{shell} .

```
proc init() {
  finitialize(v_init)
  cashell_capmp = ca_init
  // we've changed a state, so the following are needed
  if (cnode.active()) {
    cnode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This is partly successful, in that it does affect Ca_{shell} and $I_{\text{Ca}_{\text{pump}}}$, but plots of these variables seem to start from the wrong initial conditions. For example, if we try `ca_init = 0.1 μM`, the plot of `cashell_capmp` appears to start with a value of $\sim 0.044 \mu\text{M}$ instead. Using the Graph's Color/Brush menu to change the color or thickness of the plots of `cashell_capmp` and `ica`, we discover the presence of early, fast transients that overlie the y axis (Fig. 8.3 top). Thus `cashell_capmp` really does start at the right initial value, but in less than 5 microseconds it drops by $\sim 56\%$. So we have solved one mystery only to uncover another: what causes these fast transients?

Some reflection brings the realization that, although we changed the concentration in the shell, we did not properly initialize the pump. Consequently, as soon as we launch a simulation, Ca^{+2} starts binding to the pump, and this is responsible for the precipitous drop of Ca_{shell} . At the same time, the rate of active transport begins to rise, which is

reflected in the increase of $I_{Ca_{pump}}$. These changes produce the "pump transients" in Ca_{shell} and $I_{Ca_{pump}}$, which can be quite large as Fig. 8.3 shows.

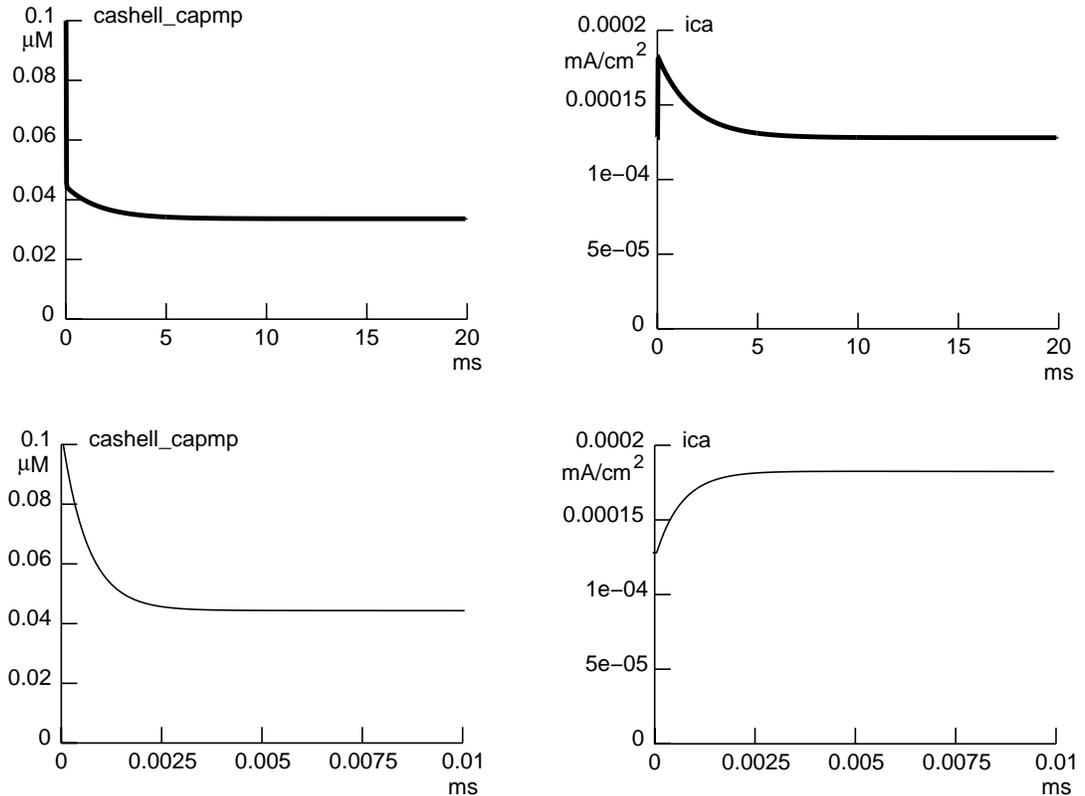


Fig. 8.3. Time course of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) following an initialization that increased Ca_{shell} abruptly after calling `init()`. The traces in the top figures were thickened to make the early fast transients easier to see. The time scale of the bottom figures has been expanded to reveal the details of these fast transients. The final steady-state levels of Ca_{shell} and $I_{Ca_{pump}}$ are the same as in Fig. 8.2.

A strategy that does what we want is to change the value of `cacore_capmp` to `ca_init` and make τ very fast before calling `finititalize()`, then wrap up by restoring the values of `cacore_capmp` and τ . This amounts to changing the model in order to achieve the desired initialization. One example of such a custom `init()` is

```
proc init() { local savcore, savtau
  // make cacore equal to ca_init
  savcore = cacore_capmp
  cacore_capmp = ca_init
  // initialize cashell to cacore
  savtau = tau_capmp
  tau_capmp = 1e-6 // so cashell tracks cacore closely
  finitalize(v_init)
```

```

// restore cacore and tau
cacore_capmp = savcore
tau_capmp = savtau
if (cnode.active()) {
  cnode.re_init()
} else {
  fcurrent()
}
frecord_init()
}

```

This code ensures that the difference between Ca_{shell} and Ca_{core} becomes vanishingly small, and at the same time allows the pump to initialize properly (Fig. 8.4).

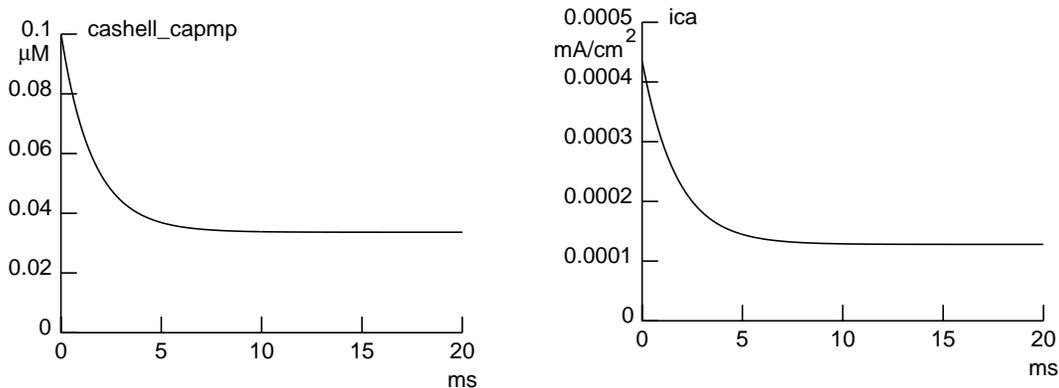


Fig. 8.4. Following proper initialization, plots of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) begin at the correct values and do not display the early fast transient that appeared in Fig. 8.3.

Now we can plot the starting value of $I_{Ca_{pump}}$ as a function of the initial Ca_{shell} .

Figure 8.5 shows a Grapher configured to do this. To make this a semilog plot, we used an independent variable x to sweep ca_{init} from 10^{-4} to 10^2 μM in 30 logarithmically equally spaced intervals. For each value of x the Grapher calculated the corresponding value of ca_{init} as 10^x , called our custom `init()`, and plotted the resulting `ica_capmp` vs. `log10(cashell_capmp)`, i.e. $\log_{10}(Ca_{shell})$. Note that $\log_{10}(cashell_capmp)$ ranges from -4 to 2 , which means that Ca_{shell} ranges from 10^{-4} to 10^2 μM , i.e. exactly the same range of concentrations as ca_{init} . This confirms the ability of our custom `init()` to set `cashell_capmp` to the desired values.

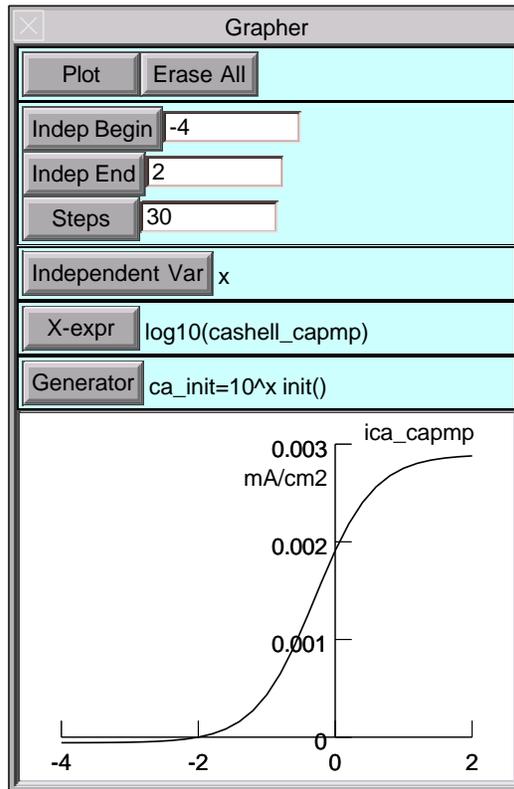


Fig. 8.5. A Grapher used to plot of $I_{\text{Ca}_{\text{pump}}}$ vs. initial Ca_{shell} . The Graph menu's Change Text was used to add the mA/cm2 label.