NEURON Hands-on Course

EXERCISES

Table of Contents

1. Introduction to the GUI 3
Lipid bilayer model

2. Interactive modeling 7
Squid axon model

3. The CellBuilder 9
Constructing a ball and stick model, saving session files

4. Using morphometric data 23
The Import3D tool

5. Python + NEURON 31

6. Using NMODL files 35
Single compartment model with HHk mechanism

7. Using ModelDB and Model View 37
Neuroinformatics tools for finding and understanding models

8. Reaction diffusion 43

9. Specifying inhomogeneous channel distributions 45
with the CellBuilder

10. Custom initialization 49

11. Introduction to the Linear Circuit Builder 51
A two-electrode voltage clamp

12. State and parameter discontinuities 57

13. Bulletin board parallelization 59
Speed up embarrassingly parallel tasks

14. HOC exercises 67
Introduction to the hoc programming language

15. Multithread parallelization 77

16 A. Networks : discrete event simulations with artificial cells **81**
Introduction to the Network Builder

16 B. Networks : continuous simulations of nets with biophysical model cells **85**
Network ready cells from the CellBuilder

17. Synchronization network in Python **89**

18. MPI parallelization **97**

19. Using the Neuroscience Gateway Portal **99**

Informal extras

Some useful hoc idioms **101**

Vectors and Matrices: reading data **103**

Vectors and Matrices: processing data **105**
Subtracting linear response

Simulation control: a family of simulations **107**
Automating the execution of a family of related simulations

Simulation control: forcing functions **113**

Optimizing a model **117**

Rectifying gap junction **133**
implemented with the Linear Circuit Builder

Analyzing electrotonus **139**
with the Impedance class

Hopfield Brody synchronization (sync) model **147**
Networks via hoc

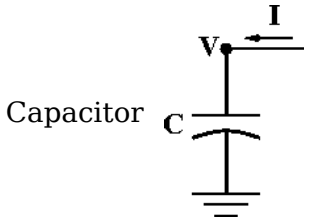
Numerical Methods Exercises **155**

Introduction to the GUI

Physical System

An electrolyte-containing vesicle with a lipid bilayer membrane that has no ion channels.

Conceptual Model



Simulation

Computational implementation of the conceptual model

We're going to use the CellBuilder to make a cell that has surface area = 100 um^2 and no ion channels.

Alternatively, we could do this in HOC

```
create soma
. . . plus more code . . .
```

or in Python

```
from neuron import h
soma = h.Section(name='soma')
. . . plus more code . . .
```

1. Start NEURON in this exercise's directory.
   Open a terminal or console (users of Windows versions prior to Windows 10 Anniversary Edition may prefer to use the bash shell in the NEURON program group),
   then assuming that you extracted the course archive to your desktop
   `cd path_to_your_Desktop/course/intro_to_gui`
2. Launch Python
   `python`
3. Make Python load NEURON's h and gui modules by entering this command at the >>> prompt:
   `from neuron import h, gui`
4. Get a CellBuilder

NEURON Main Menu / Build / CellBuilder

5. Specify surface area
   CellBuilder / Geometry
   Make sure there is a check mark in the "Specify Strategy" box.
   Select "area".
   Click on "Specify Strategy" to clear the check mark.
   Make sure that area is 100 um^2.

Click on the "Continuous Create" box so that a check mark appears in it.

Congratulations! You now have a model cell with a soma that has surface area = 100 um^2.

Save this to a "session file" so you can recreate it whenever you like.
NEURON Main Menu / File / save session

Using the computational model

You'll need these items:

- controls for running simulations
- a graph that shows somatic membrane potential vs. time
- a current clamp to inject a current pulse into the model cell
- a graph that shows the stimulus current vs. time

Be sure to save your work to a session file as you go along!

Make a RunControl panel for launching simulations.
NEURON Main Menu (NMM) / Tools / RunControl
Make a graph that shows somatic membrane potential vs. time
NMM / Graph / Voltage axis
Now run a simulation by clicking on RunControl's "Init & Run". What happened?
Make a current clamp.
NMM / Tools / Point Processes / Managers / Point Manager
Make this an IClamp.
PointProcessManager / SelectPointProcess / IClamp
Show the IClamp's Parameters panel
PPM / Show / Parameters
Make it deliver a 1 nA x 1 ms current pulse that starts at 1 ms.
del (ms) = 1
dur (ms) = 1
amp (nA) = 1
Make a graph that shows the stimulus current vs. time.
NMM / Graph / Current axis
Make it show the IClamp's i
Click on the graph's menu button (left upper corner of its canvas) and select "Plot what?"
Plot what? / Show / Objects
Select IClamp (left panel)
Select 0 (middle panel)

Select i (middle panel)
Plot what?'s edit field should now contain IClamp[0].i
Click on Accept

Run a simulaton.
Too big? Divide IClamp.amp by 10 and try again.

Exercises

1. Run a simulation.

Double the duration and halve the amplitude of the injected current (injected charge is constant).

Notes:

- Select Show / Parameters in the PointProcessManager window to view the field editors.
- Multiplying a field editor value by 2 can be done by typing *2 <return> at the end of the value. Divide by 2 by typing /2 <return>.
- The mouse cursor must be in the panel that contains the field editor.

Return the duration and amplitude to their default values (click in the checkmark box with left mouse button).
Halve the duration and double the amplitude.

2. Insert passive conductance channels, then see how this affects the model.

In the CellBuilder, click on the Biophysics radio button.
Make sure there is a check mark in the "Specify Strategy" box.
Click on the "pas" button so that it shows a check mark.

Now repeat exercise 1.

Notice that v tends toward -70 mV, which is the default reversal potential for the passive channel. This is because the initial condition v = -65 is not the steady state for this model.

In the "RunControl" set the Init field editor to -70 and repeat the run.

To rescale a Graph automatically, click on its menu box (square in upper left corner of its canvas) and select "View=Plot"

3. Change the stimulus amplitude to 1A (1e9 nA) and run. Rescale the graphs to see the result.
This is an example of a model that is a poor representation of the physical system.

4. Change the stimulus duration to 0.01 ms.
This is an example of a simulation that is a poor representation of the model.

Change the number of Points plotted/ms to 100 and dt to 0.01 ms and run again.

5. Increase the amplitude to 1e4 nA, cut the duration to 1e-5 ms, increase Points plotted/ms to 1e5, and start a simulation . . .
After a few seconds of eternity, stop the simulation by clicking on RunControl / Stop

Bring up a Variable Time Step tool.
NMM / Tools / VariableStepControl
Select "Use variable dt" and try again.

When you're done, use "NMM / File / Quit" to exit NEURON or type `exit()` at the terminal prompt.

A final word

intro_to_gui/solution/bilayer.hoc contains a properly configured CellBuilder, plus a custom interface for running simulations. The one item it doesn't have is a VariableStepControl.

bilayer.hoc is actually a session file that was given the "hoc" extension so that MSWin users could launch it by double clicking on the file name. For didactic reasons we prefer that you load it from Python instead.
1. cd to the directory that contains bilayer.hoc
2. Launch Python from the command line.
3. At the >>> prompt enter the commands

```
from neuron import h, gui
h.load_file('bilayer.hoc')
```

Interactive Modeling

Physical System

Giant axon from Loligo pealei

Image from http://www.mbl.edu

Conceptual Model

Hodgkin-Huxley cable equations

$$\frac{D}{4R_a}\frac{\partial^2 V}{\partial x^2} = C_m\frac{\partial V}{\partial t} + \bar{g}_{na}m^3h(V-E_{na}) + \bar{g}_k n^4(V-E_k) + g_\ell(V-E_\ell)$$

$$\frac{dm}{dt} = -\alpha_m\, m + \beta_m(1-m) \quad \alpha_m = \frac{0.1(V+40)}{1-\exp(-0.1(V+40))} \quad \beta_m = 4\exp(-(V+65)/18)$$

$$\frac{dh}{dt} = -\alpha_h\, h + \beta_h(1-h) \quad \alpha_h = 0.07\exp(-0.05(V+65)) \quad \beta_h = \frac{1}{1+\exp(-0.1(V+35))}$$

$$\frac{dn}{dt} = -\alpha_n\, n + \beta_n(1-n) \quad \alpha_n = \frac{0.01(V+55)}{1-\exp(-0.1(V+55))} \quad \beta_n = 0.125\exp(-(V+65)/80)$$

Simulation

Computational implementation of the conceptual model

We *could* implement this model in Python:

```
from neuron import h, gui

axon = h.Section(name='axon')
axon.L = 2e4
axon.diam = 100
axon.nseg = 43
axon.insert('hh')
```

But for this exercise, let's instead use the CellBuild tool to create the model. Save the model in `hhaxon.ses` using NEURONMainMenu/File/savesession.

Using the computational model

If starting from a fresh launch of python, you can load the saved ses file by loading NEURON and its GUI: `from neuron import h, gui` and then selecting NEURONMainMenu/File/loadsession.

Alternatively you can use NEURON to execute `interactive_modeling/hhaxon.ses`

1. cd to course/interactive_modeling
2. Start python, and at the >>> prompt enter the commands

```
from neuron import h, gui
h.load_file('hhaxon.ses')
```

Exercises

1) Stimulate with current pulse and see a propagated action potential.

The basic tools you'll need from the NEURON Main Menu **:**
Tools / Point Processes / Manager / Point Manager to specify stimulation
Graph / Voltage axis and Graph / Shape plot to create graphs of v vs t and v vs x.
Tools / RunControl to run the simulation
Tools / Movie Run to see a smooth evolution of the space plot in time.

2) Change excitability by adjusting sodium channel density.

Tool needed:
Tools / Distributed Mechanisms / Viewers / Shape Name

3) Use two current electrodes to stimulate both ends at the same time.

4) Up to this point, the model has used a very fine spatial grid calculated from the Cell Builder's d_lambda rule.
Change nseg to 15 and see what happens.

NEURON Python documentation

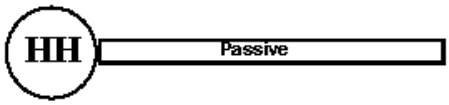
Using the CellBuilder

Physical System

A mammalian CNS neuron

Conceptual Model

Ball-and-stick approximation to cell



Simulation

Computational implementation of the conceptual model

This will be constructed with the CellBuilder, a graphical tool for building and managing models of individual cells. With this and other GUI tools, you will

1. set up a "virtual experimental preparation" (the model cell itself).
2. set up a "virtual lab rig".
    - Simulation control: RunControl
    - Instrumentation:
        Stimulator--PointProcessManager configured as IClamp
        Graphs--v vs. t, v vs. distance ("space plot")

You will also learn a simple but effective strategy for modular organization of your programs.

- Separate the specification of the representation of the biological system (anatomy, biophysics, connections between cells in a network . . .) from other items, such as the specification of instrumentation (voltage or current clamps, graphs etc.) and controls (e.g. RunControl panel).
- Use a short program that pulls all of the pieces together.

Modular organization makes it easier to

- develop and debug models
- reuse the same model cell in many different kinds of simulation experiments
- perform the same kind of experiment on many different model cells

Getting started

1. cd to course/cellbuilder
2. Start Python, and at the >>> prompt enter the command

```
from neuron import h,gui
```

Making the representation of the biological properties

Use the CellBuilder to make a simple ball and stick model that has these properties:

Section	Anatomy	Compartmentalization	Biophysics
soma	length 20 microns diameter 20 microns	nseg = 1	Ra = 160 ohm cm, Cm = 1 uf/cm^2 Hodgkin-Huxley channels
dend	length 1000 microns diameter 5 microns	nseg = 1	Ra = 160 ohm cm, Cm = 1 uf/cm^2 passive with Rm = 10,000 ohm cm^2

Hints

1. To start a CellBuilder, click on
   NEURONMainMenu / Build / CellBuilder.
2. **CellBuilder overview and hints.**
3. Helpful items in the on-line Programmer's Reference :
   diam L nseg hh pas

Using the representation of the biological properties

At this point you should have:

1. entered the specification of the ball and stick model in the CellBuilder
2. saved the CellBuilder, with its Continuous Create button checked, to a session file called ballstk.ses and verified what you saved
3. exited NEURON

In the course/cellbuilder directory, make an `init.py` file with the contents

```
from neuron import h, gui

# load your model specification
h.load_file('ballstk.ses')
# your user interface
h.load_file("rig.ses")
```

Also make a rig.ses file that contains the single command

```
print "ready!"
```

Actually you could put any innocuous statements you like into the rig.ses file, because you'll eventually overwrite this file with a custom user interface that you construct.

Start Python with the init.py argument.

```
python init.py
```


Oops! Runs and almost immediately exits!

Try again, but this time execute the command

```
python -i init.py
```

The -i switch makes Python enter interactive mode instead of exiting immediately after execution of init.py is complete.

Exercises

1. Establish that the representation in the computer corresponds to the conceptual model.
Connectivity? (`h.topology()`)
Are the properties what you expect? Try

```
from pprint import pprint  # prettier printing (optional; could use print)
pprint(h.soma.psection())
pprint(h.dend.psection())
```

Comments:

- The CellBuilder works by executing HOC code, so the section names that you specify in a CellBuilder become the HOC names of those sections. But just like with HOC functions, simply stick the h. prefix onto the front of the name of a section created by HOC, and you have its Python name.

2. Use the NEURONMainMenu toolbar to construct an interface that allows you to inject a stimulus current at the soma and observe a plot of somatic Vm vs. time.

3. When a current stimulus is injected into the soma, does it flow into the dendrite properly? Hint: examine a space plot of membrane potential.

Saving and Retrieving the Experimental Rig

You now have a complete setup for doing simulation experiments. The CellBuilder, which specifies your "experimental preparation," is safe because you saved it to the session file ballstk.ses. However, the GUI that constitutes your nicely-configured "lab rig" (the RunControl, PointProcessManager, graph of v vs. t, and space plot windows) will be lost if you exit NEURON prematurely or if the computer crashes.

To make it easy to reconstitute the virtual lab rig, use the Print & File Window Manager (PFWM) to save these windows to a session file. Here's how to bring up the PFWM and use it to select the windows for everything but the CellBuilder, then save these windows to a session file called rig.ses. This will allow you to immediately begin with the current GUI.

Test rig.ses by using NEURONMainMenu / File / load session to retrieve it. Copies of the "lab rig" windows should overlay the originals. If so, exit NEURON and then restart it with the init.hoc argument. It should start up with the windows that you saved.

More exercises

4. How does the number of segments in the dendrite affect your simulation?
Turn on Keep Lines in the graph of Vm vs. t so you will be able to compare runs with different nseg.
Then at the >>> prompt execute the command

```
h.dend.nseg *= 3
```

and run a new simulation. Repeat until you no longer see a significant difference between consecutive runs.
Finally, use the command

```
print(h.dend.nseg)
```

to see how many dendritic segments were required.

5. Is the time step (h.dt in Python) short enough?

6. Here's something you should try on your own, perhaps after class tonight: using the CellBuilder to manage models "on the fly."

Footnotes and Asides

1. The CellBuilder can be used to make your own "digital stem cells." If you have a model cell that you would like to return to later, save the CellBuilder to a session file. To bring the model back, just retrieve the session file. This is a good way to create an "evolutionary sequence" of models that differ only in certain key points.

2. The CellBuilder can also be used to manage models based on detailed morphometric reconstructions. This is covered in a later exercise.

Overview of the CellBuilder

The CellBuilder is a graphical tool for creating, editing, and managing models of nerve cells. It is probably most useful in two different settings.

1. Building a model from scratch that will have only a few sections. If you need more than 5 - 10 sections, it may be more convenient to write an algorithm that creates the model under program control.
2. Managing the biophysical properties of a model that is based on complex morphometric data, without having to write any code.

The CellBuilder breaks the process of creating and managing a model of a cell into tasks that are analogous to what you would do if you were writing a program in hoc or Python. They are

1. setting up the model's topology (branching pattern)
2. grouping sections into subsets. For example, it might make sense to group dendritic branches into subsets according to shared anatomical criteria (e.g. basilar, apical, oblique, spiny, aspiny), or biophysical criteria (passive, active).
3. assigning anatomical or biophysical properties to individual sections or subsets of sections

The CellBuilder can import a model that already exists during a NEURON session. It can also be used in conjunction with NEURON's Import3D tool to create new model cells based on detailed morphometric reconstructions.

Starting the CellBuilder

Go to the working directory of your choice, start python, and enter the command
```
from neuron import h,gui
```

Then get a CellBuilder by selecting NEURONMainMenu / Build / Cell Builder

Using the CellBuilder

Across the top of the CellBuilder there is a row of radio buttons, plus a checkbox labeled "Continuous Create". For now you should leave the checkbox empty.

Use the radio buttons to select the following pages.

About
: Scan this information, but don't worry if everything isn't immediately obvious. You can reread it any time you want.

Topology
: This is where you change the branching architecture of the cell.
Select "Make Section" from the list of actions, and then L click in the graph panel on a blank space to the right of the soma. Use the other actions as necessary to make your model look like the figure in this exercise.

Subsets

Subsets can simplify management of models that have many sections. The CellBuilder automatically creates the "all" subset, which contains every section in the model. There is no need to define subsets for the ball and stick model, so just skip this page.

Geometry

This is for setting the dimensions and spatial grid (nseg) of the soma and dendrite.

1. Make sure the Specify Strategy button is checked.
   Choose soma from the list of subsets and section names, and then select L, diam, and nseg.
   Repeat for dend.
   Note: choosing nseg lets you set the number of segments manually. The CellBuilder also offers you the option of automatically adjusting nseg according to one of its built-in compartmentalization strategies; we will return to this later.
2. Clear the Specify Strategy button.
   Use the list of section names to select the soma and dendrite individually, and enter the desired dimensions in the numeric fields for L and diam.
   For now leave nseg = 1.

Biophysics

Use this to endow the sections with biophysical properties (ionic currents, pumps, buffers etc.).

1. Specify Strategy. This is for inserting biophysical mechanisms into the sections of your model (Ra and cm for "all," hh for the soma, and pas for the dend section).
2. To examine and adjust the parameters of the mechanisms that you inserted, clear the Specify Strategy button.

Management

This panel is not used in the ball and stick exercise.

When you are done, the CellBuilder will contain a complete specification of your model cell. However, no sections will actually exist until you click on the CellBuilder's Continuous Create button.

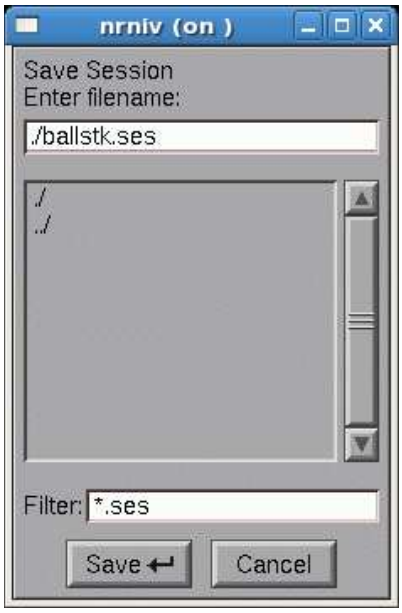
At this point you should turn Continuous Create ON because many of NEURON's GUI tools require sections to exist before they can be used (e.g. the PointProcessManager).

Saving your work

This took a lot of effort and you don't want to have to do it again. So save the completed CellBuilder to a session file called ballstk.ses in the working directory exercises/cellbuilder. To do this, click on

NEURONMainMenu / File / save session

This brings up a file browser/selector panel. Click in the top field of this tool and type
ballstk.ses
as shown here



Then click on the Save button.

Checking what you saved

Retrieve ballstk.ses by clicking on
NEURONMainMenu / File / load session
and then clicking on ballstk.ses.

A new CellBuilder window called CellBuild[1] will appear. Since Continuous Create is ON, this new CellBuilder forces the creation of new sections that will replace any pre-existing sections that have the same names.

The terminal will display a message warning you that this happened:

```
Previously existing soma[0] points to a section which is being deleted
Previously existing dend[0] points to a section which is being deleted
```

Check Topology, Geometry, and Biophysics. When you are sure they are correct, exit NEURON.

Questions and answers about sessions and ses files

What's a session? What's a ses file good for? What's in a ses file? For answers to these and other questions about sessions and ses files, read this.

Saving Windows

Saving the windows of an interface for later re-use is an essential operation. The position, size, and contents of all the windows you have constructed often represent a great deal of labor that you don't want to do over again. Losing a CellBuilder or Multiple Run Fitter window can be painful.

This article explains how to save windows, how to save them in groups in separate files in order to keep separate the ideas of specification, control, graphing, optimization, etc, and how to recover as much work as possible if a saved window generates an error when it is read from a file.

What is a session?

We call the set of windows, including the ones that are hidden, a "session". The simplest way to save the windows is to save them all at once with NEURONMainMenu / File / save session. This creates a session file, which NEURON can use to recreate the windows that were saved. Session files are discussed below.

When (and how) to save all windows to a ses file

Near the beginning of a project it's an excellent practice to save the entire session in a temporary file whenever a crash (losing all the work since the previous save) would cause distress. Do this with NEURONMainMenu / File / save session. Be sure to verify that the session file can be retrieved (NEURONMainMenu / File / load session) before you overwrite an earlier working session file!

It is most useful to retrieve such a session file right after launching NEURON, when no other windows are present on the screen. It is especially useful if one of the windows is a CellBuild or NetGUI ("Network Builder"), because most windows depend on the existence of information declared by them. Conflicts can arise if there are multiple CellBuild or NetGui windows that could interfere with one another, especially if they create sections with the same names.

When (and how) to save selected windows

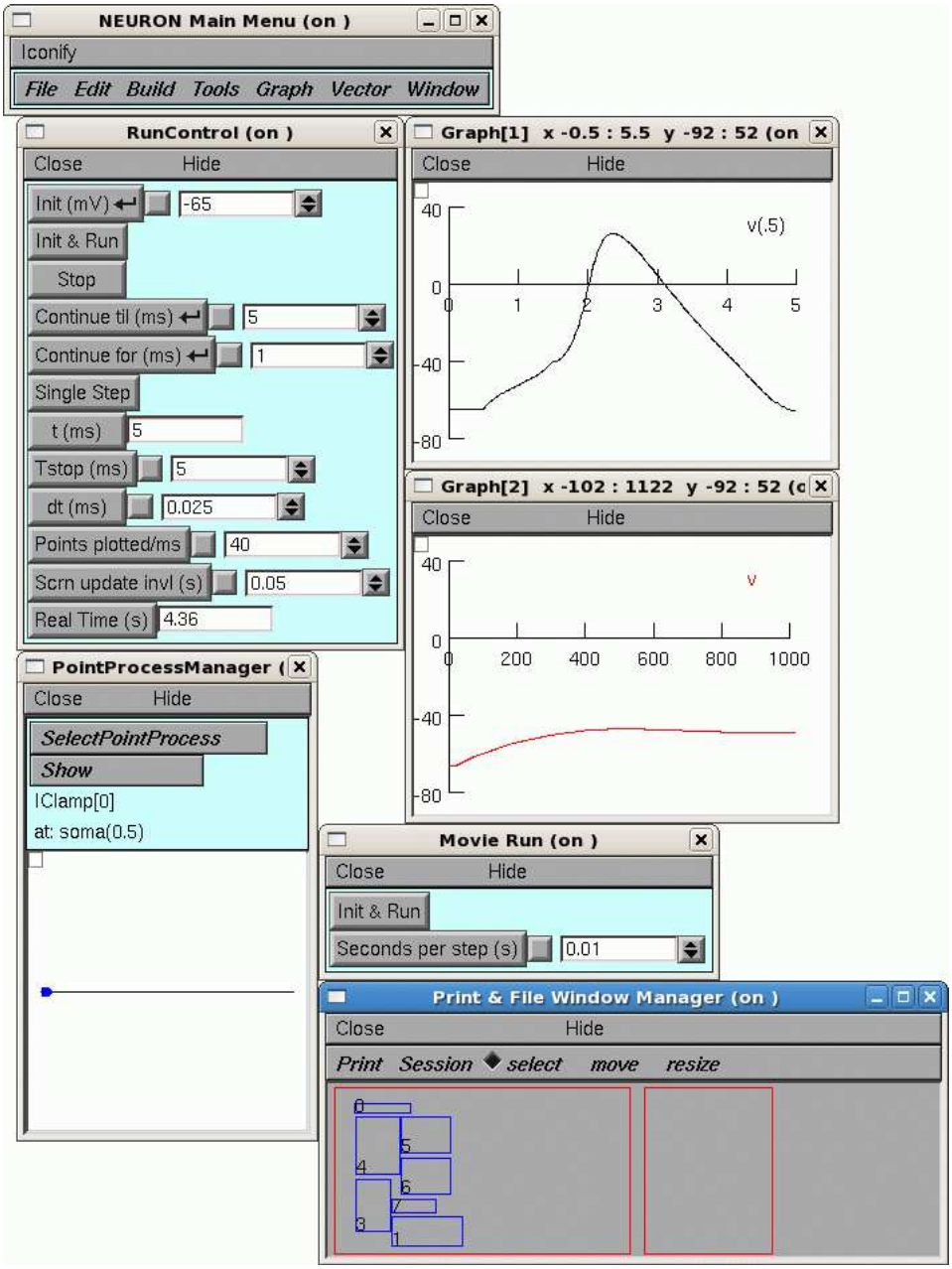
For small modeling tasks, it is most convenient to save *all* windows to a single session file. The main drawback to saving all windows in a single session file is that it mixes specification, control, parameter, and graphing windows.

For more complex modeling tasks, it may be necessary to have more control over what groups of windows are created. This allows you to easily start a simulation by retrieving the desired variant of a CellBuilder window, separately retrieving one of several possible stimulus protocols and parameter sets, and lastly retrieving groups of graph windows.

The Print & File Window Manager (PFWM)

The PFWM has many useful features, especially saving session files, unhiding hidden windows, printing hard copy, and generating PostScript and ASCII (plain text) output files. This discussion focusses on how to use it to save selected windows to a session file.

To bring up the PFWM, click on
NEURONMainMenu / Window / Print & File Window Manager



The figure above contains several NEURON windows, with a PFWM in the lower right corner. Notice the two red boxes in the bottom panel of the PFWM. The box on the left is a virtual display of the computer monitor: each of the blue rectangles corresponds to one of NEURON's windows. The relative positions and sizes of these rectangles represent the arrangement of the windows on the monitor.

The toolbar just above the red boxes contains two menu items (Print, Session), and three radio buttons (select, move, resize) that help you use the PFWM. The radio buttons set the "mode" of the PFWM, i.e. they determine what happens when you click on the blue

rectangles. When the PFWM first comes up, it is in select mode and the radio button next to the word "select" is highlighted.

How to select and deselect windows

First make sure the PFWM is in select mode. If it isn't, click on "select" in the toolbar.

Decide which of the rectangles in the virtual display correspond to the windows you want to save to a ses file. If you aren't sure which blue rectangle goes with which window, drag a window on your screen to a new location, and see which rectangle moves.

When you have decided, click inside the desired rectangle in the virtual display, and a new blue rectangle, labeled with the same number, will appear in the PFWM to the right of the virtual screen. You can select as many windows as you like. To deselect a window, just click inside the corresponding blue rectangle on the right hand side of the PFWM.

Saving the selected windows

To save the selected windows, click on
Session / Save selected
in the PFWM, and use the file browser/selector panel to specify the name of the ses file that is to be created.

What's in a ses file

A session file is actually just a sequence of hoc instructions for reconstructing the windows that have been saved to it. Session files are generally given the suffix ".ses" to distinguish them from user-written hoc files.

In a session file, the instructions for each window are identified by comments. It is often easy to use a text editor to modify those instructions, e.g. change the value of a parameter, or to remove all the instructions for a window if it is preventing the ses file from being loaded.

What can go wrong, and how to fix it

The most common reason for an error during retrieval of a session file is when variables used by the window have not yet been defined. Thus, retrieving a point process manager window before the prerequisite cable section has been created will result in a hoc error. Retrieving a Graph of SEClamp[0].i will not succeed if SEClamp[0] does not exist. In most cases, loading the prerequisite sessions first will fix the error. The init.hoc file is an excellent place to put a sequence of load_file statements that start a default session. Errors due to mismatched object IDs are easy to correct by editing the session file. Mismatched object IDs can occur from particular sequences of creation and destruction of windows by the user. For example, suppose you

1. Start a PointProcessManager and create the first instance of an IClamp. This will be IClamp[0]
2. Start another PointProcessManager and create a second instance of an IClamp. This will be IClamp[1]
3. Close the first PointProcessManager. That destroys IClamp[0].
4. Start a graph and plot IClamp[1].i
5. Save the session.

If you now exit and re-launch NEURON and retrieve the session, the old IClamp[1] will be re-created as IClamp[0], and the creation of the Graph window will fail due to the invalid variable name it is attempting to define. The fix requires editing the session file and changing the IClamp[1].i string to IClamp[0].i

---

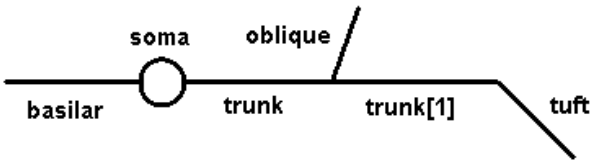
Managing Models on the Fly

If the CellBuilder's Continuous Create button is checked, changes are passed from the CellBuilder to the interpreter as they occur. This means that you can immediately explore the effects of anything you do with the CellBuilder.

To see how this works, try the following.

1. Start NEURON with its standard GUI in the exercises/cellbuilder directory (remember how?).

2. Bring up the CellBuilder and construct a cell that looks like this:



Use any anatomical and biophysical properties you like; these might be interesting to start with:

Section	L (um)	diam (um)
soma	30	30
trunk	400	3
trunk[1]	400	2
oblique	300	1.5
tuft	300	1
basilar	300	3

- nseg = 1 for all sections
- Ra = 160, Cm = 1, both uniform throughout the cell
- soma has hh
- trunk and all its tributaries have hh, but with gnabar & gkbar reduced by a factor of 10 and gleak = 0 (Subsets makes this easy)
- all dendrites have pas with gpas = 3e-5

3. Toggle Continuous Create ON.

4. In the interpreter verify the structure and parameters of your model with `h.topology()` and `for sec in h.allsec(): print(sec.psection())`

5. At the proximal end of tuft, place an alpha function synapse that has onset = 0 ms, tau = 1 ms, gmax = 0.01 umho, and e = 0 mV (hint: NEURON Main Menu / Tools / Point Processes / Managers / Point Manager).

6. Open a graph window to plot soma Vm vs. time. Also set up a space plot that shows Vm along the length of the cell from the distal end of the basilar to the distal end of the tuft.

7. Run a simulation. If necessary, increase Tstop until you can see the full time course of the cell's response to synaptic input.

8. Increase nseg until the spatial profile of Vm is smooth enough (a couple of applications of

```
# python
for sec in h.allsec():
  sec.nseg *= 3
```

or

```
forall nseg *= 3 // hoc
```

in the interpreter window should do the trick). You may need to adjust the peak synaptic conductance in order to trigger a spike. Then use the command

```
# python
for sec in h.allsec():
  print('%s: %d' % (sec, sec.nseg))
```

or

```
forall print secname(), " ", nseg // hoc
```

to see how many segments are in each section.

9. You can also set nseg for any or all sections using the CellBuilder according to options that you select by Geometry/Specify Strategy. You can set the number of segments manually, or let the CellBuilder adjust them automatically according to one of these criteria:

- **d_lambda** (the maximum length of any segment, expressed as a fraction of the AC length constant at 100 Hz for a cylindrical cable with identical diameter, Ra, and cm)
  or
- **d_X** (the maximum anatomical length of any segment)

Try each of these criteria, setting different values for d_lambda or d_X, and see what happens to nseg in each section and how this affects the spatial profile of membrane potential.

Comments:

- Don't forget that that execution of your strategy is sequential. In other words, if your strategy specifies d_lambda for the all subset, but then sets nseg = 1 for the tuft section, the tuft will end up with nseg = 1 even though it needs a much finer grid according to the d_lambda criterion.
- Whether you choose d_lambda or d_X, the final value of nseg will be an odd number (this preserves the node at x = 0.5).
- Of these two options, d_lambda has a more rational basis is generally preferable. `d_lambda<=0.1` is usually adequate.

10. What happens if the sodium channels are blocked throughout the apical dendrites? Use the CellBuilder to reduce apical gnabar to 0 and then run a simulation.

Working with morphometric data

If you have detailed morphometric data, why not use it? This may be easier said than done, since quantitative morphometry typically produces hundreds or thousands of measurements for a single cell -- you wouldn't want to translate this into a model by hand. Several programs have been written to generate NEURON code from morphometric data files, but NEURON's own Import3D tool is probably the most powerful and up-to-date. Currently Import3D can read Eutectic, Neurolucida (v1 and v3 text files), swc, and MorphML files. It can also detect and localize errors in these files, and repair many of the more common errors automatically or with user guidance.

Exercises

A surprising result

Some morphometric data files contain surprises, but the Import3D tool handled this one nicely:

Reading a morphometric data file and converting it to a NEURON model.

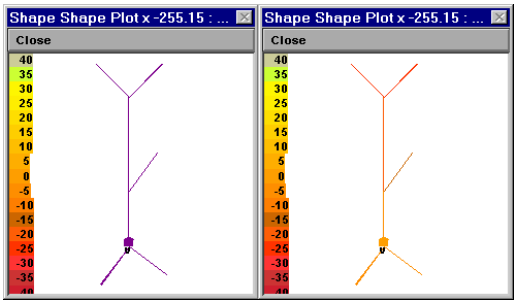
Exploring morphometric data with the Import3D tool.

A "litmus test" for models with complex architecture

Some morphometric reconstructions contain orphan branches, or measurement points with diameters that are (incorrectly) excessively small or even zero. Here's a test that can quickly detect such problems:

1. Use the data to create a model cell.
2. Insert the `pas` mechanism into all sections.
   If you're dealing with a very extensive cell (especially if the axon is included), you might want to cut Ra to 10 ohm cm and reduce g_pas to 1e-5 mho/cm^2.
3. Turn on Continuous Export (if you haven't already).
4. Bring up a Shape Plot.
5. Turn this into a Shape Plot of Vm (R click in the Shape Plot and scroll down the menu to "Shape Plot". Release the mouse button and a color scale calibrated in mV should appear).
6. Examine the response of the cell to a 3 nA current step lasting 5 ms applied at the soma.
   For very extensive cells, especially if you have reduced g_pas, you may want to increase both Tstop and the duration of the injected current to 1000 ms and use variable dt.

Here's an example that used a toy cell:



Left: Vm at t = 0 ms. Right: Vm at 5 ms.

Quantitative tests of anatomy

This Python code checks for pt3d diameters smaller than 0.1 um, and reports where they are found:

```
for sec in h.allsec():
    for i in range(sec.n3d()):
        if sec.diam3d(i) < 0.1:
            print('%s %d %g' % (sec, i, sec.diam3d(i)))
```

If you're reusing someone's HOC files, you could use the equivalent HOC statement

```
forall for i=0, n3d()-1 if (diam3d(i) < 0.1) print secname(), i, diam3d(i)
```

There are many other potential strategies for checking anatomical data, such as

- creating a space plot of diam. Bring up a Shape Plot and use its Plot what? menu item to select diam. Then select its Space plot menu item, click and drag over the path of interest, and voila!
- making a histogram of diameter measurements, which can reveal outliers and systematic errors such as "favorite values" and quantization artifacts (what is the smallest diameter that was measured? how fine is the smallest increment of diameter?). This requires some coding, which is left as an exercise to the reader.

Detailed morphometric data: sources, caveats, and importing into NEURON

Currently the largest collection of detailed morphometric data we know of is NeuroMorpho.org. There are many potential pitfalls in the collection and use of such data. Before using any data you find at NeuroMorpho.org or anywhere else, be sure to carefully read any papers that were written about those data by the anatomists who obtained them.

Some of the artifacts that can afflict morphometric data are discussed in these two papers, which are well worth reading:

Kaspirzhny AV, Gogan P, Horcholle-Bossavit G, Tyc-Dumont S. 2002. Neuronal morphology data bases: morphological noise and assesment of data quality. Network: Computation in Neural Systems 13:357-380. doi:10.1088/0954-898X_13_3_307

Scorcioni, R., Lazarewicz, M.T., and Ascoli, G.A. Quantitative morphometry of hippocampal pyramidal cells: differences between anatomical classes and reconstructing laboratories. Journal of Comparative Neurology 473:177-193, 2004. doi:10.1002/cne.20067

Data input and model output

To import morphometric data into NEURON, bring up an Import3d tool, and specify the file that is to be read. Look at what you got, then export the data as a NEURON model.

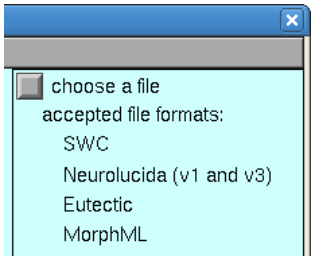
A. Get an Import3D tool.

Start NEURON and go to exercises/using_morphometric_data, if you're not alredy there.

Open an Import3D tool by clicking on Tools / Miscellaneous / Import3D in the NEURON Main Menu.

B. Choose a file to read.

In the Import3D window, click on the "choose a file" checkbox.

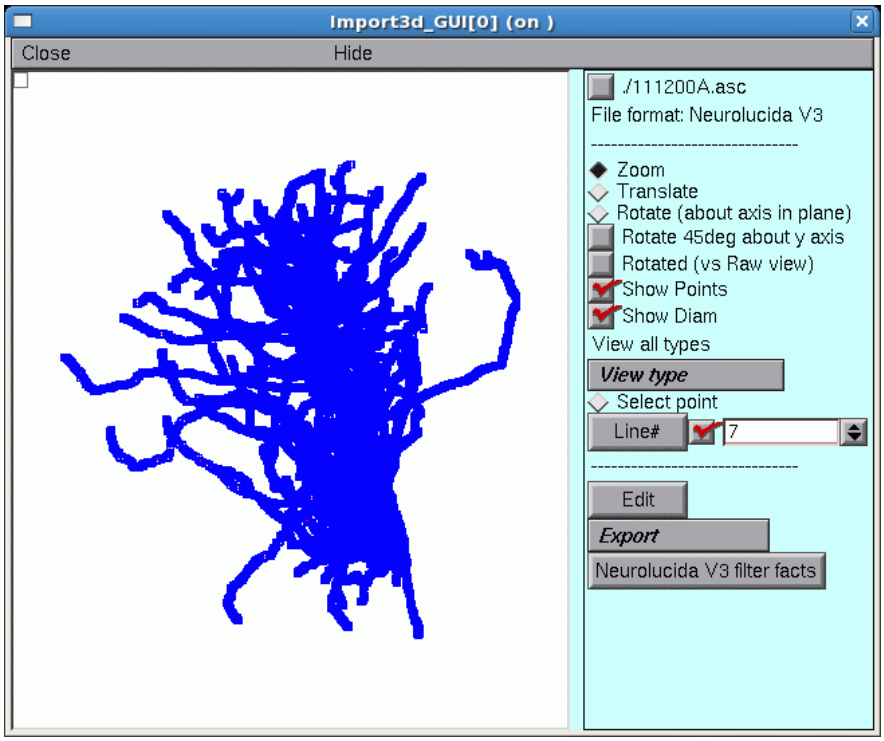


This brings up a file browser. Since you're already in the directory that contains the data, just click on the name of the data file (111200A.asc), and then click the file browser's Read button.

NEURON's xterm (interpreter window) prints a running tally of how many lines have been read from the data file.

```
oc>
21549 lines read
```

After a delay that depends on the size of the file and the speed of the computer, a figure will appear on the Import3D tool's canvas. Each point at which a measurement was made is marked by a blue square.



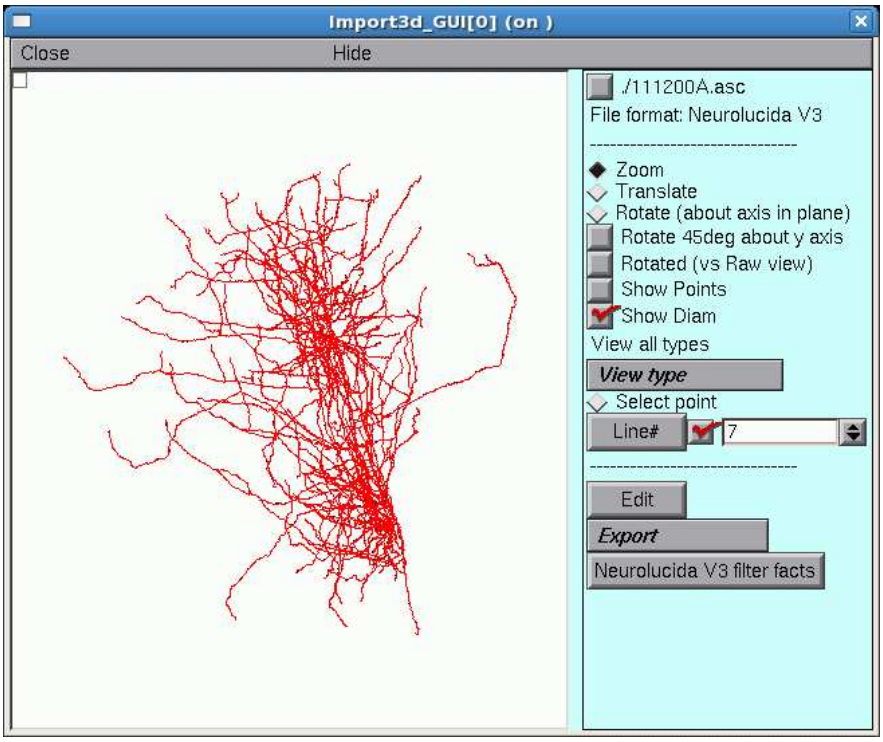
If the Import3D tool finds errors in the data file, a message may be printed in the xterm, and/or a message box may appear on the screen. For this particular example there were no errors--that's always a good sign!

The top of the right panel of the Import3D tool will show the name and data format of the file that was read. The other widgets in this panel, which are described elsewhere, can be used to examine and edit the morphometric data, and export them to the CellBuilder or the hoc interpreter.

C. Let's see what it looks like.

It's always a good idea to look at the results of any anatomical data conversion--but those blue squares are in the way!

To get rid of the blue squares that are hiding the branched architecture, click on the Show Points button in the right panel of the Import3D tool. The check mark disappears, and so do the blue squares.

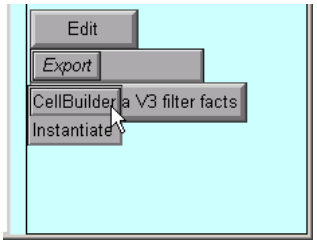


That's a very dense and complex branching pattern.

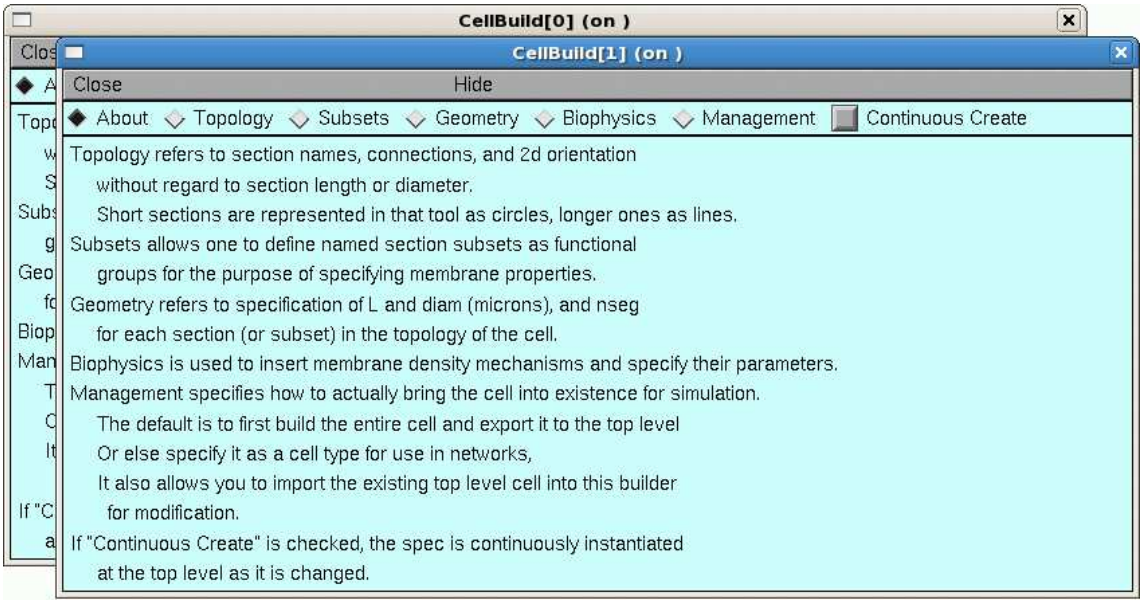
D. Exporting the model.

The Import3D tool allows us to export the topology (branched architecture) and geometry (anatomical dimensions) of these data to a CellBuilder, or straight to the hoc interpreter. It's generally best to send the data to the CellBuilder, which we can then save to a session file for future re-use. The CellBuilder, which has its own tutorial, is a very convenient tool for managing the biophysical properties and spatial discretization of anatomically complex cell models.

So click on the Export button and select the CellBuilder option.



But this example contains a surprise: instead of one CellBuilder, there are two! Under MSWin, they are offset diagonally as shown here, but under UNIX/Linux they may lie right on top of each other so you'll have to drag the top one aside.

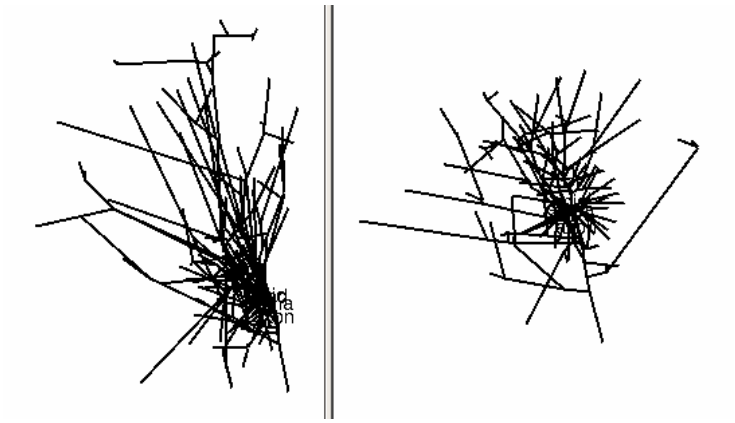


Does getting two CellBuilders mean that the morphometric data file contained measurements from two cells? Maybe that's why the branching pattern was so dense and complex.

But there is an unpleasant alternative: maybe all this data really is from one cell. If there was a mistake in data entry, so that the proximal end of one branch wasn't connected to its parent. one CellBuilder would contain the orphan branch and its children, and the other CellBuilder would contain the rest of the cell.

How can you decide which of these two possibilities is correct?

Examining the Topology pages of these CellBuilders shows that CellBuild[0] got most of the branches in the bottom half of the Import3D's canvas, and CellBuild[1] got most of the branches in the top half. The morphologies are ugly enough to be two individual cells; at least, neither of them is obviously an orphan dendritic or axonal tree.



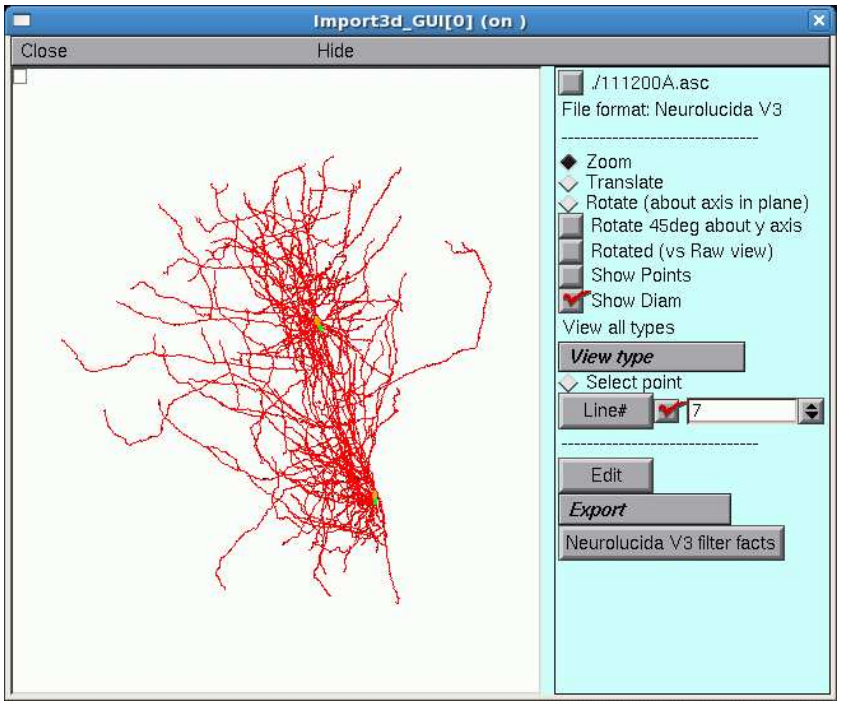
Until you know for sure, it is safest to use the Print & File Window Manager (PFWM) to save each CellBuilder to its own session file. I optimistically called them bottomcell.ses and topcell.ses, respectively.

At this point, you should really use the Import3D tool to closely examine these data, and try to decide how many cells are present.

Go back to the main page to learn more about the Import3D tool.

Examining morphometric data with Import3D

Take a new look at the shape in the Import3D tool.



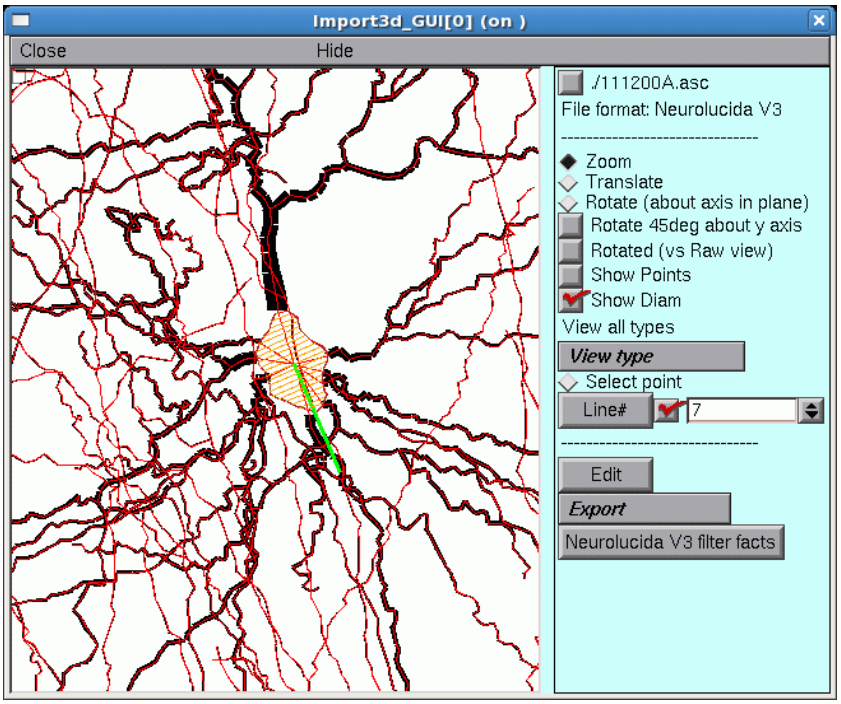
Those two little green lines in the dense clusters are new. They appeared *after* exporting to the CellBuilder. And is there a little orange blob at one end of each green line?

To find out what this is all about, it is necessary to discover what lies at the center of these dense clusters.

A. Zooming in

To zoom in for a closer look, first make sure that the Import3D tool's Zoom button is on (if it isn't, just click on it).
Then click on the canvas, just to the right of the area of interest, and hold the mouse button down while dragging the cursor to the right. If it becomes necessary to re-center the image, click on the Translate button, then click on the canvas and drag the image into postion. To start zooming again, click on the Zoom button.

Repeat as needed until you get what you want.



The irregular shape at the center, with the transverse orange lines, is the soma of a neuron. The green line is its principal axis, as identified by the Import3D tool. At least 9 neurites converge on it, and a fine red line connects the proximal end of each branch to the center of the soma.

If you zoom in on the other green line and orange blob, you'll find another soma there.

So by zooming in, it is possible to discover that this particular morphometric data file contained measurements from at least two different cells.

To zoom out, make sure the Zoom button is on,
then click near the right edge of the canvas and drag toward the left. To fit the image to the window, just use the graph's "View = plot" menu item.

Go back to the main page

NEURON Scripting Exercises

Exercise 1:

Given a `list` of sections:

```
from neuron import h, gui

secs = [
    h.Section(name='soma'),
    h.Section(name='dend'),
    h.Section(name='axon')
]
```

and given some data

```
data = {
    "axon": {"L": 100, "diam": 1},
    "soma": {"L": 10, "diam": 10},
    "dend": {"L": 50, "diam": 3}
}
```

Write a script that uses the data to set the `L` and `diam` parameters for each section. Set the number of segments such that there is (1) an odd number of segments and (2) each segment is as close to 5 microns long as possible without going over. Connect all sections not named `soma` to the center of the soma.

Use a `h.PlotShape()` to check your work. Store the object as, e.g. `ps`. Then use `ps.show(0)` to display the diameters.

Bonus challenge: Move the data into a JSON file and use Python's `json` module to read it from the file.

Exercise 2:

Simulate and visualize a propagating action potential on a long cable with Hodgkin-Huxley dynamics triggered by a current clamp.

Things to consider:

What is an appropriate diameter, length of cable, and current injection amplitude?

How can you best illustrate a propagating action potential in a single figure?

What discretization should you use?

Exercise 3:

Continuing with the previous exercise, record and plot the membrane potential, sodium current, and potassium current at the center of the cable as a function of time.

Hint: Use the `Vector` class' `record` method.

Things to consider:

How does the relationship between the sodium and potassium currents differ in a Hodgkin-Huxley simulation from the relationship in mammalian cells?

Exercise 4:

Construct two Y-shaped cells, where each branch is 2 microns in diameter and 100 microns long. Add Hodgkin-Huxley channels with the default parameters. Inject current into one cell and ensure that you are able to generate a propagating action potential in that cell while the other one remains at rest.

Connect the two cells with a gap junction (see `halfgap.mod`) at a location of your choosing. A 3 MΩ resistance as in `halfgap.mod` should allow an action potential in one cell to initiate an action potential in the other. Verify this. What is the strongest resistance that still allows action potential initiation in the second cell?

Bonus challenge: Make each cell an instance of a Python `class`.

Bonus challenge: Modify the `class` to allow repositioning (moving and/or rotating) the cells.

Exercise 5:

Download the model from http://modeldb.yale.edu/126814 Create a Python script that loads the model, injects current into the center of the soma from t=2 to t=4 ms sufficient to generate an

action potential, and records and plots membrane potential, sodium current, and potassium current as functions of time from t=0 to t=10 ms.

Hint: `h.load_file('mosinit.hoc')`

Hint: This model uses the variable time step solver. Be sure to record t or switch to a fixed step solver with `h.cvode_active(0)`.

Bonus challenge: Export the recorded data to a CSV file and then open and plot it in Excel, MATLAB, or a similar tool of your choice.

Bonus challenge: How does the plotted data change as `h.celsius` is varied?

Exercise 6:

NEURON's `ExpSyn` mechanism generates synaptic currents of the form

$$i = g(v - E_{syn})$$

where

$$g' = -g/\tau$$

What is the role of E_{syn} ? How does it change for an excitatory vs an inhibitory synapse? (Note, this is the parameter `e` in the code.)

Construct two single compartment neurons with Hodgkin-Huxley dynamics, one of which receives a current pulse at 2 ms, another which receives a current pulse at 10 ms. Ensure that both cells fire action potentials after the input.

Now, using `NetCon` and `ExpSyn`, construct an inhibitory synapse between the two with the cell that fires later as the post-synaptic cell. Choose a delay and strength such that the post-synaptic cell is inhibited from firing. Plot the membrane potentials vs time.

halfgap.mod

```
NEURON {
  POINT_PROCESS HalfGap
  POINTER vgap
  RANGE r, i
  ELECTRODE_CURRENT i
}

PARAMETER {r = 3 (megohm)}

ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}

BREAKPOINT { i = (vgap - v) / r }
```


Using NMODL files

Step 0:

Copy hhkchan.mod into an empty directory.

Step 1:

All versions of NEURON on MacOS and Linux, and NEURON 7.7+ on Windows

In a terminal, navigate to the folder containing the hhkchan.mod file and run the shell script:

```
nrnivmodl
```

On a PC using a version of NEURON before 7.7

Launch mknrndll from the icon in the NEURON program group.

> Navigate to the directory containing the desired mod files.
> Select "Make nrnmech.dll".

and the "mknrndll" script will create a nrnmech.dll file which contains the HHk model.

Step 2:

Using the location of hhkchan.mod as the working directory, start NEURON by typing `python` and then

```
from neuron import h, gui
```

This will automatically load the mechanisms compiled in Step 1. If NEURON doesn't find any compiled mechanism, only the "built-in" mechanisms (hh, pas, IClamp, etc) will be available.

Step 3:

Bring up a single compartment model with surface area of 100 um^2 (NEURON Main Menu / Build / single compartment) and toggle the HHk button in the Distributed Mechanism Inserter ON. Verify that the new HHk model (along with the Na portion of the built-in HH channel) produces the same action potential as the built-in HH channel (using both its Na and K portions).

Using ModelDB

http://senselab.med.yale.edu/modeldb/ is the the home page of ModelDB, but for most of this exercise we'll just use a local copy of some files from it.

We'll be analyzing a couple of models in order to answer these questions:

1. What physical system is being represented, and for what purpose?
2. What is the representation of the physical system, and how was it implemented?
3. What is the user interface, how was it implemented, and how do you use it?

Example: Moore et al. 1983

> Moore JW, Stockbridge N, Westerfield M (1983)
> On the site of impulse initiation in a neurone.
> J. Physiol. 336:301-11

This one doesn't have any mod files, but there's plenty to keep us busy.

The model archive moore83.zip has already been downloaded and unzipped. You'll find its contents in exercises/modeldb_and_modelview/moore83

1. What physical system is being represented, and for what purpose?

ModelDB provides a link to PubMed for the abstract (here's a local copy of the abstract). At some point we should also read the paper.

Go to exercises/modeldb_and_modelview/moore83 and read the README file. Does it provide any more clues?

2. What is the representation of the physical system, and how was it implemented?

Load mosinit.hoc in exercises/modeldb_and_modelview/moore83

```
from neuron import h, gui
h.load_file('mosinit.hoc')
```

The PointProcessManager shows a shape plot of the cell.

Using Python to Examine What's in the Model

This model was written in HOC, but we can still use Python to explore it. To see what sections exist and how they are connected, type

```
h.topology()
```

and

```
from pprint import pprint  # optional; could use print
for sec in h.allsec():
    pprint(sec.psection())
```

at the >>> prompt.

At this point, you might think you'd have to start reading source code in order to get any more information about what's in the model. But the Model View tool can save you a lot of time and effort, and it has the advantage of telling you exactly what the model specification is. This is a big advantage, since some programs use complex specification and initialization routines that change model structure and parameters "on the fly."

Using Model View to Discover What's in a Model

Start a Model View tool
NEURON Main Menu / Tools / Model View

This shows a browsable outline of the model's properties.
Asterisks mark the items that are "expandable."
Expand the outline by clicking on "1 real cells". Then peer inside the model by clicking on each of its expandable subitems.

Notice what happens when you get down to
1 real cells / root soma / 6 inserted mechanisms / hh / 2 gnabar_hh
In the ModelView Range Graph, click on the "SpacePlot" button, then click and drag in that tool's shape plot to bring up a space plot of gnabar_hh. Do the same for gkbar_hh and gl_hh.

A shortcut for discovering the distributions of spatially inhomogeneous parameters:
Density Mechanisms / Heterogeneous parameters
reveals all that are spatially nonuniform. Click on any one to make the ModelView Range Graph reveal its values over the model.

Analyzing the Underlying Code

Was this model specified by user-written hoc code, or was a CellBuilder used?
Exit the simulation and search the hoc files for create statements.
In the terminal execute

```
grep create *hoc
```

(MSWin users: first open a bash shell, then cd to the exercises/modeldb_and_modelview /moore83 directory)

Alternatively you could try Windows Explorer's semi-useful Search function, or open each hoc file with a text editor and search for create.

If no hoc file contains the create keyword, maybe the CellBuilder was used.
Run mosinit.hoc again and look for a CellBuilder.
If you don't see one, maybe a Window Group Manager is hiding it.
Click on NEURON Main Menu/Window and look for one or more window names that are missing a red check mark. If you see one, scroll down to it and release the mouse button.
If a CellBuilder pops up, examine its Topology, Subsets, Geometry, and Biophysics pages.
Do they agree with the output of forall psection() and/or what you discovered with the Model View tool?

"Extra Credit" Question

Now you know what's in the model cell, and how it was implemented. Suppose you wanted to get a copy of it that you could use in a program of your own. Would you do this by saving a CellBuilder to a new session file, or by using a text editor to copy create, connect, insert etc. statements from one of the hoc files?

3. What is the user interface, how was it implemented, and how do you use it?

What is that panel with all the buttons?
What happens if you click on one of them?
Click on a different one and see what happens to the string at the top of the panel.
Click on some more and see what happens to the blue dot in the PointProcessManager's shape plot.
Is this one of the standard GUI tools you can bring up with the NEURON Main Menu?
How does it work?

> Hints: look for an xpanel statement in one of the hoc files.
> Read about xpanel, xbutton, and xvarlabel in the help files.

Find the procedures that implement the actions that are caused by clicking on a button.
The last statement in each of these procedures launches a simulation.
What does the very first statement do?
What does the second statement do?

> The remaining statements do one or more of the following:
> change model parameters (e.g. spatial distribution of HH in the dendrite)
> change stimulus parameters (e.g. stimulus location and duration)
> change simulation parameters

Why does the space plot automatically save traces every 0.1 ms?

> Hint: analyze the procedure that actually executes a simulation
> Which hoc file contains this procedure?

What procedure actually changes the stimulus location, duration, and amplitude? Read about PointProcessManager in the help files.

Another example: Mainen and Sejnowski 1996

> Mainen ZF, Sejnowski TJ (1996)
> Influence of dendritic structure on firing pattern in model neocortical neurons.
> Nature 382:363-6

This one has interesting anatomy and several mod files.

The model archive patdemo.zip has already been downloaded and unzipped. Its contents are in exercises/modeldb_and_modelview/patdemo

1. What physical system is being represented, and for what purpose?

This is ModelDB's link to PubMed for the abstract, and here's a local copy of the abstract. Another paper to read.

Read the README.txt file in exercises/modeldb_and_modelview/mainen96 . Any more clues here?

2. What is the representation of the physical system, and how was it implemented?

Compile the mod files, then load mosinit.hoc as in the previous exercise.

Four different cell morphologies are available. Select one of them, then click on the Init button to make sure that all model specification and initialization code has been executed. Use Model View to browse the model, and examine the heterogeneous parameters.

Now it's time to discover how this model was created. Where are the files that contain the pt3d statements of these cells?

This program grafts a stylized myelinated axon onto 3d specifications of detailed morphometry.
Where is the hoc code that accomplishes this grafting?
If you load mosinit.hoc and then try to import one of the cell morphologies into the CellBuilder, do you also get the axon?

Length and diameter are scaled in order to compensate for the effect of spines on dendritic surface area. Find the procedure that does this.
What is an alternative way to represent the effect of spines?

nseg is adjusted in each section so that no segment is longer than 50 um. What procedure does this?

Five active currents and one pump mechanism are included. Examine these mod files.
Do they appear to be compatible with CVODE?

Check them with modlunit.
Did you find any inconsistencies?
Do any of these seem likely to affect simulation results?
Are there any other warning messages?
Is there anything that would cause numerical errors?
How might you fix the problems that you found?

3. What is the user interface, how was it implemented, and how do you use it?

mosinit.hoc brings up a minimal GUI for selecting cells and running simulations.
How did they do that?

4. Reuse one of their cells in a model of your own design

Import its morphology into a CellBuilder, then save the CellBuilder to a session file and exit the simulation.

Restart nrngui and load the CellBuilder's session file.
Assign a plausible set and spatial distribution of biophysical properties and save to a session file.

Instrument your new model and run a simulation.
Save the model, with instrumentation, to a session file.

J Physiol. 1983 Mar;336:301-11.
On the site of impulse initiation in a neurone.
Moore JW, Stockbridge N, Westerfield M.

In the preceding paper (Moore & Westerfield, 1983) the effects of changes in membrane properties and non-uniform geometry on impulse propagation and threshold parameters were investigated. In this paper the contributions of these and other parameters to the site of initiation of an impulse were determined by computer simulations using the Hodgkin-Huxley membrane description, the cable equations, and geometry appropriate for a simplified motoneurone with a non-myelinated axon. Antidromic invasion of action potentials into the soma was found to depend upon (a) the ionic channel rate constants (determined by the temperature), (b) the abruptness of the transition from the small-diameter axon to the larger diameter (and increased load) of the soma-dendrite, (c) extensions of active properties into the dendrite, and (d) density of ion channels. The location of the apparent site of initiation of impulses was not necessarily at the site of synaptic input nor the nearest active membrane. Its position depended upon (a) the fraction of the dendritic tree with excitable membrane, and secondarily on (b) the stimulus strength. Even with uniform excitability in the active membrane, the apparent site of initiation could be moved a considerable distance from the soma and the site of stimulation by appropriate choice of the various parameters noted above.

Nature. 1996 Jul 25;382(6589):363-6.
Influence of dendritic structure on firing pattern in model neocortical neurons.
Mainen ZF, Sejnowski TJ.
Howard Hughes Medical Institute, Computational Neurobiology Laboratory, Salk Institute for Biological Studies, La Jolla, California 92037, USA.

Neocortical neurons display a wide range of dendritic morphologies, ranging from compact arborizations to highly elaborate branching patterns. In vitro electrical recordings from these neurons have revealed a correspondingly diverse range of intrinsic firing patterns, including non-adapting, adapting and bursting types. This heterogeneity of electrical responsivity has generally been attributed to variability in the types and densities of ionic channels. We show here, using compartmental models of reconstructed cortical neurons, that an entire spectrum of firing patterns can be reproduced in a set of neurons that share a common distribution of ion channels and differ only in their dendritic geometry. The essential behaviour of the model depends on partial electrical coupling of fast active conductances localized to the soma and axon and slow active currents located throughout the dendrites, and can be reproduced in a two-compartment model. The results suggest a causal relationship for the observed correlations between dendritic structure and firing properties and emphasize the importance of active dendritic conductances in neuronal function.

Reaction-Diffusion Exercises

Exercise 1:

Consider the signaling molecule IP3, which in Wagner et al 2004 was modeled as having a diffusion coefficient in the cytosol of 0.283 $\mu m^2/ms$. (Our units here are not those usually used by cell biologists, but they are the ones used by NEURON's reaction-diffusion module as they are reasonable for neuronal electrophysiology.)

Consider a 101 micron long section of dendrite discretized into 101 compartments. Set IP3 at an initial concentration of 1 μM for 0.4 ≤ seg.x ≤ 0.6 and 0 elsewhere. (Be careful about units.) How long does it take the concentration at seg.x = 0.7 to rise to a concentration of 100 nM? What is the peak concentration obtained at this point over all time? What is the limiting value as t ➡ ∞ ?

In 25 °C water, calcium has a diffusion coefficient of 1.97×10^{-5} cm^2/s according to [1]. What is this in NEURON's units of $\mu m^2/ms$? (Note: calcium will have a different effective diffusion constant in a cell than in water due to buffering molecules, etc.) Answer the question about time for a concentration increase at seg.x = 0.7 for calcium in water. Do the same for glucose which has a diffusion coefficient of 6×10^{-6} according to the same source. How does varying the diffusion coefficient affect the time it takes to raise concentration at a given distance by a given amount?

Exercise 2:

Do the same pulse-diffusion calcium simulation, but on a cylindrical domain with 5 cylindrical spines 3 microns long and 1 micron in diameter. How does the addition of spines affect the diffusion? Why?

How many molecules of calcium would be present in such a spine if it had an average concentration of 100 nM, a biological plausible concentration? (Hint: one cubic micron at 1mM concentration has about 602,200 molecules.) If one molecule stochastically entered or left the spine, what would be the percentage change in the spine's concentration?

Exercise 3:

Import a CA1 pyramidal cell into NEURON from NeuroMorpho.Org. Declare chemical species X to diffuse across the entire cell at D=1 $\mu m^2/ms$ with initial concentration of 1 mM in the soma and 0 elsewhere. View the distribution of X at 5 and 25ms on both a ShapePlot and along the apical

[1] http://www.physiologyweb.com/calculators/diffusion_time_calculator.html

as a function of distance from the soma. Plot a time series of the concentration of X at the center of the soma.

Exercise 4:

Neurons are well-known for their propagating action potentials, but electrical signalling is not their only form of regenerative signaling. Berridge 1998 described neuronal calcium dynamics as functioning as a “neuron-within-a-neuron.”

Why is pure diffusion insufficient? What is the expected time for a molecule of IP3 to travel a distance of 50 microns? 100 microns? 1 meter (the approximate length of the longest axon in a human)?

Hint: $E[t] = \frac{x^2}{2D}$

The actual dynamics underlying these calcium wave dynamics is complicated, so let’s start with the simplest reaction-diffusion model of a propagating wave:

On a 500 micron long piece of dendrite add a species Y with diffusion constant 1 changing by an rxd.Rate of scale * (0 - y) * (alpha - y) * (1 - y). For a first test take scale = 1 and alpha = 0.3. Set initial conditions such that Y is 1 where seg.x < 0.2 and 0 elsewhere. Plot the concentration of Y vs position at several time points to show the development of a traveling wave front.

Run the same simulation with alpha = 0.7. What changes? More generally, how does alpha affect the wave?

Calculate the speed of propagation of the wave front at alpha = 0.3. How can you change scale and the diffusion coefficient to get a wave of the same shape that travels at 50 microns per second? What happens to the shape of the wave front if the diffusion coefficient is increased but the other parameters stay the same?

Things to consider:

How do you know if you are using an appropriate value of nseg or dt?

References

Berridge, M. J. (1998). Neuronal calcium signaling. Neuron, 21(1), 13-26.

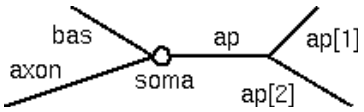
Wagner, J., el al. (2004). A wave of IP 3 production accompanies the fertilization Ca 2+ wave in the egg of the frog, Xenopus laevis: theoretical and experimental support. Cell Calcium, 35(5), 433-447.

Inhomogeneous channel distribution

Physical System

Conceptual Model

The conceptual model is a much simplified stylized representation of a pyramidal cell with active soma and axon, passive basilar dendrites, and weakly excitable apical dendrites.



Computational Model

Here is the complete specification of the computational model:

	Geometry		
Section	L (um)	diam (um)	Biophysics
soma	20	20	hh
ap[0]	400	2	hh*
ap[1]	300	1	hh*
ap[2]	500	1	hh*
bas	200	3	pas
axon	800	1	hh

*--gnabar_hh, gkbar_hh, and gl_hh in the apical dendrites decrease linearly with path distance from the soma. Density is 100% at the origin of the tree, and falls to 0% at the most distant termination.
To ensure that resting potential is -65 mV throughout the cell, e_pas in the basilar dendrite is -65 mV.
Other parameters: cm = 1 µf/cm^2, Ra = 160 ohm cm, nseg governed by d_lambda = 0.1.

The exercise

1. Use the GUI to implement and test a model cell with the anatomical and biophysical properties described above.

- Start by using the CellBuilder to make a "version 0 model" that has uniform membrane properties in the apical dendrites (hh mechanism with default conductance densities).

    - Verify that the anatomical and biophysical properties of the model are correct--especially the channel distributions.
    - Test the model with a virtual lab rig that includes a RunControl, IClamp at the soma, and plots of v vs. t and v vs. distance. Employ a modular strategy so that you can reuse this experimental rig with a different model cell.
- Next, copy the "version 0 model" CellBuilder and modify this copy to implement version 1: a model in which gnabar_hh, gkbar_hh, and gl_hh in the apical tree decrease linearly with distance from the origin of the apical tree, as described above.

  Verify the channel distributions, and test this new model with the same rig you used for version 0.

2. Pick any anatomically detailed morphology you like, import it into NEURON, and implement a model with biophysical channel densities similar to those described above.

Hints

1. Before doing anything, think about the problem. In particular, determine the formulas that will govern channel densities in the apical tree.

In each apical section, gnabar_hh at any point x in that section will be
gnabar_hh = gnabar_max * (1 - distance/max_distance)
where
distance = distance from origin of the apical tree to x
and
max_distance = distance from {origin of the apical tree} to {the most remote dendritic termination in the apical tree}.

The formulas for gk_hh and gl_hh are similar.

distance/max_distance is "normalized distance into the apical tree from its origin." So the distance metric p should be 0 at the origin of the apical tree, and 1 at the end of the most remote dendritic termination in the apical tree.

2. Hints for using the CellBuilder to specify an inhomogeneous channel distribution.

Overview of the task

1. Specify the
    - subset
    - distance metric `p`
    - parameter `param` that depends on distance
    - function `f` that governs the relationship between the parameter and the distance metric
      `param = f(p)`
2. Verify the implementation.

Details

This involves a lot of steps--remember to save intermediate results to session files!

Act 1: Specify the subset

On the CellBuilder's Subsets page, create the subset.

Act 2: Specify the distance metric

Do this with a SubsetDomainIterator.

- Create a SubsetDomainIterator.
    - Click on the subset, then click on "Parameterized Domain Page"
    - Click on "Create a SubsetDomainIterator".
      The name of the SubsetDomainIterator will appear in the middle panel of the CellBuilder. It will be called *name*_x, where *name* is the name of the subset.
- Use the SubsetDomainIterator to specify the distance metric.
    - Click on *name*_x.
      The right panel of the CellBuider will show the controls for specifying the distance metric. Default is path length in um from the root of the cell (0 end of soma).
      Drag the slider back and forth to see the corresponding location(s) in the shape plot (boundary between red and black). Distance from the root to the red-black boundary is shown on the canvas as "p=*nnn.nnn*". You can also click on the canvas near the shape and drag the cursor back and forth; the canvas will now show the "range" of the boundary in the nearest section, and the name of that section.
    - **metric** offers three choices: path length from root, radial (euclidian) distance from root, and "projection onto line" (distance from a plane that passes through the root and is orthogonal to the principal axis of the root section).
    - **proximal** allows specifying an "offset" for the proximal end(s) of the subset. This lets you assign a distance of 0 to the origin of the apical tree.
    - **distal** allows specifying whether or not to normalize the distance metric, and if normalized, whether the metric is to become 1 only at the most distal end, or at all distal ends.

Act 3: Specify the parameters that depend on distance

Do this on the CellBuilder's Biophysics page.

- Select Strategy (make a check mark appear in the box).
  *name*_x (the name of the SubsetDomainIterator) will appear in the middle panel.
- Click on *name*_x.
- In the right panel of the CellBuider, select the parameters that the SubsetDomainIterator will control.

Intermezzo

Time to step back and see where we are.
At this point, *name*_x "knows" these things:

- the sections that are in its subset
- the distance metric for each spatial location in these sections
- the parameters in these sections that will be governed by this distance metric

But it doesn't know the functional relationships between the parameters and the distance metric--and each parameter can have its own function. Defining these functions is the next item to take care of.

Act 4: Specify the functional relationship between each parameter and the distance metric

- Clear the Strategy checkbox.
  *name*_x will appear in the middle panel of the CellBuilder, and beneath it will be the names of each of the paramters that were selected in the strategy.
- Click on one of the parameter names.
  The right hand panel presents controls for specifying f.
  The default is a Boltzmann function. ***f(p)*** offers this and three other choices:
  - **Ramp** (linear)
  - **Exponential**
  - **New** lets you enter your own function

  Each of these has its own list of user-settable parameters.

  show allows you to bring up a graph or shape plot for visual confirmation of how the biophysical parameter varies in space. This is convenient, but you'll probably want to check the finished model with Model View.

Finale trionfale

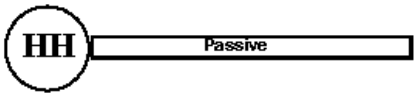
Turn on Continuous Create, and use Model View to verify the channel distributions.

Custom initialization

Physical System

Conceptual Model

Ball-and-stick approximation to cell



Simulation

The aim of this exercise is to learn how to perform one of the most common types of custom initializaton: **initializing to steady state**.

We start by making a ball and stick model in which the natural resting potential of the somatic sphere and dendritic cylinder are different. No matter what v_init you choose, default initializaton to a uniform membrane potential results in simulations that show an initial drift of v as the cell settles toward a stable state.

After demonstrating this initial drift, we will implement and test a method for initializing to steady state that leaves membrane potential nonuniform in space but constant in time.

Getting started

In this exercise you will be creating several files, so you need to be in a working directory for which you have file "write" permission. Start NEURON with exercises/custom_initialization as the working directory.

Computational implementation of the conceptual model

Use the CellBuilder to make a simple ball and stick model that has these properties:

Section	Anatomy	Compartmentalization	Biophysics
soma	length 20 microns diameter 20 microns	d_lambda = 0.1	Ra = 160 ohm cm, Cm = 1 uf/cm^2 Hodgkin-Huxley channels
dend	length 1000 microns diameter 5 microns	d_lambda = 0.1	Ra = 160 ohm cm, Cm = 1 uf/cm^2 passive with Rm = 10,000 ohm cm^2

Turn Continuous Create ON and save your CellBuilder to a session file.

Using the computational model

Bring up a RunControl and a voltage axis graph.
Set Tstop to 40 ms and run a simulation.
Use View = plot to see the time course of somatic membrane potential more clearly.

Add dend.v(1) to the graph (use Plot what?), then run another simulation.
Use Move Text and View = plot as needed to get a better picture.

Add a space plot and use its Set View to change the y axis range to -70 -65.
Run another simulation and watch how drastically v changes from the initial condition.

Save everything to a session file called `all.ses` (use File / save session) and exit NEURON.

Exercise: initializing to steady state

In the exercises/custom_initialization directory, make an `init_ss.py` file with the contents

```
from neuron import h, gui

def ss_init(t0=-1e3, dur=1e2, dt=0.025):
  """Initialize to steady state.
  Executes as part of h.finitialize()
  Appropriate parameters depend on your model
  t0 -- how far to jump back (should be < 0)
  dur -- time allowed to reach steady state
  dt -- initialization time step
  """
  h.t = t0
  # save CVode state to restore; initialization with fixed dt
  old_cvode_state = h.cvode.active()
  h.cvode.active(False)
  h.dt = dt
  while (h.t < t0 + dur): h.fadvance()
  #
  # restore cvode active/inactive state if necessary
  h.cvode.active(old_cvode_state)
  h.t = 0
  if h.cvode.active():
    h.cvode.re_init()
  else:
    h.fcurrent()
  h.frecord_init()

fih = h.FInitializeHandler(ss_init)

# model specification
h.load_file('all.ses') # ball and stick model with exptl rig
```

Now execute `init_ss.py`.
Click on Init & Run and see what happens.

"Special credit" exercise

Another common initialization is for the initialized model to satisfy a particular criterion.
Create an initialization that ensures the resting potential throughout the cell equals v_init.

Introduction to the Linear Circuit Builder

The linear circuit builder is a graphical interface for electrical circuits consisting of idealized resistors, capacitors, and first order operational amplifiers. Circuits may have intra- and extra-cellular connections to cells at multiple locations. Batteries and current sources use a 3 step protocol similar to the IClamp.

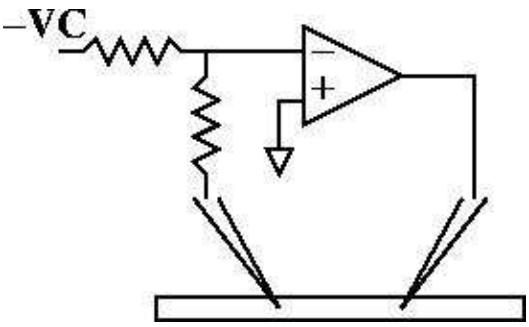
During a simulation, the voltages at each circuit node and the currents through each battery and op amp are computed along with the voltage and state variables of each cable section. At present, simulation runs can only use the default implicit fixed time step method (don't use cvode).

Linear circuits are simulated as a specific instance of the more general LinearMechanism class. This latter class allows NEURON to transcend its historical limitation to tree structure and allows simulation of arbitrary extracellular fields and low resistance gap junctions. Unfortunately this generality comes at a significant performance cost. In the worst case (gap junctions connecting every compartment to every compartment) the simulation time for gaussian elimination increases from order N to order N^3. A single gap junction between two cells does not increase the gaussian elimination time. But a gap junction connecting one end of a cable to the other end doubles the gaussian elimination time.

Physical System

Two electrode Voltage clamp of an HH axon.

Model



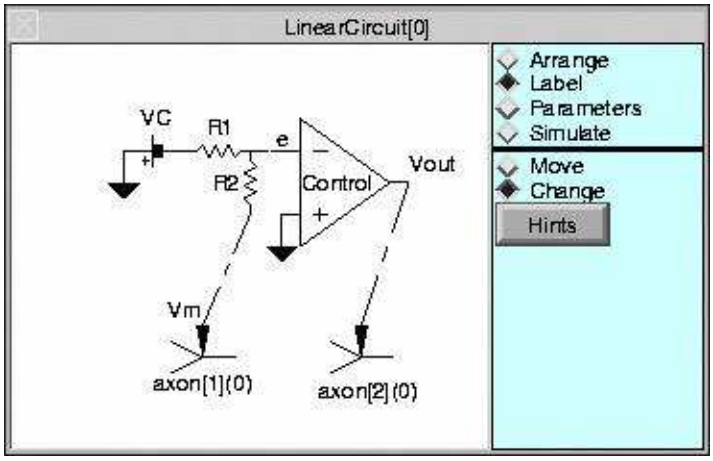
Ideal voltage clamp of axon containing standard hh channels. The axon is 300 um long and 10 um in diameter. Ra=35 ohm-cm. The ideal voltage recording electrode is 100 um from the left and the ideal current injection electrode is 100 um from the right.

At least a plausible case can be made for how the voltage clamp circuit works by realizing that the input to the high gain amplifier can only be 0 volts if (assuming the resistors are equal) the membrane potential of the recording electrode is equal to VC. If it is less than VC then the input will be negative and the op amp will produce a high voltage output thus injecting current into the cell and causing the membrane potential to increase toward VC. Whether this circuit will be stable remains to be seen.

Simulation

1) Use the cellbuilder to create the HH axon. This axon should be in three pieces each 100 um long so that regardless of the values of nseg, the electrodes can be placed precisely at 100 and 200 microns.
Example

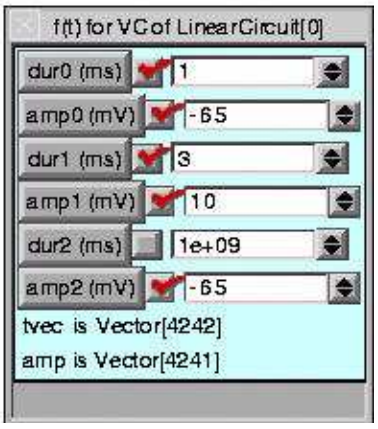
2) Arrange components to define the structure of an ideal voltage clamp circuit using the Linear Circuit Builder. When arranged and labeled, the Linear Circuit Builder should look like



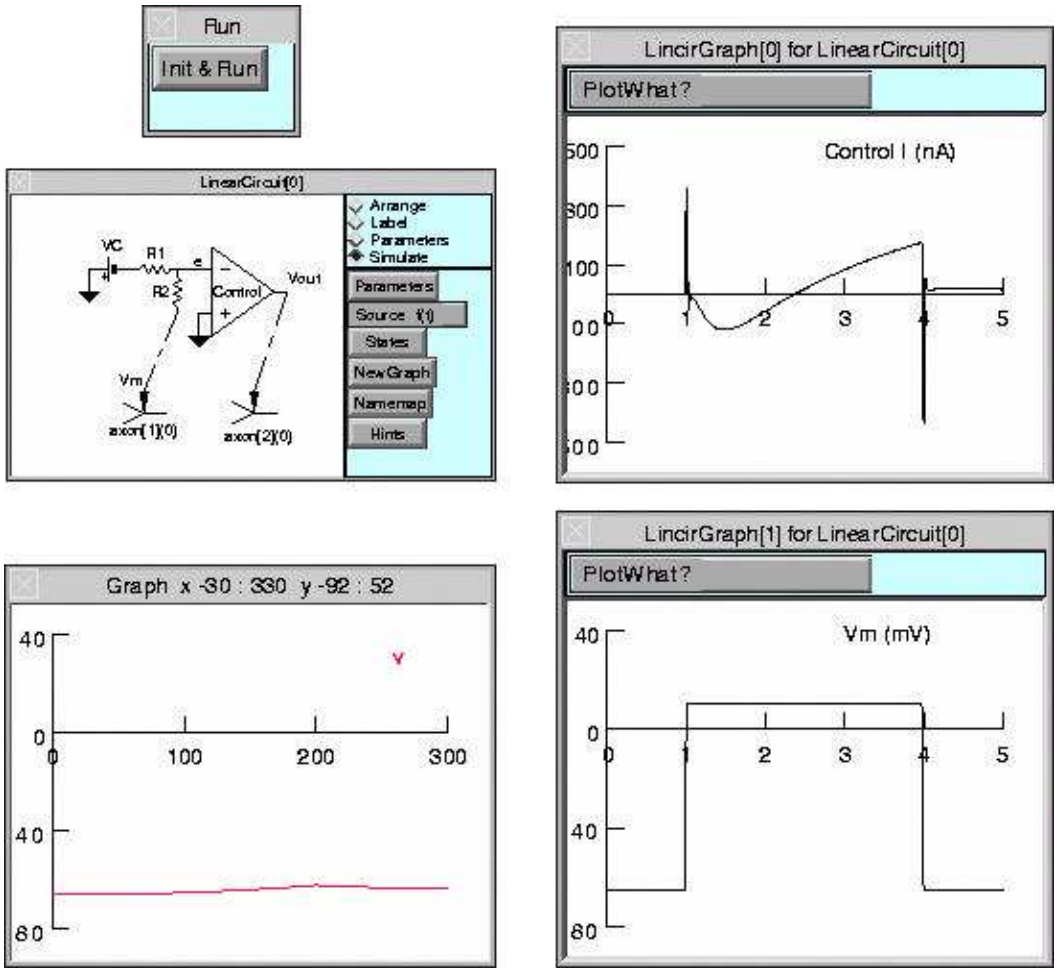
Construction Hints

Notice that the battery has its negative terminal connected to R1 so that Vm will have the same sign as VC.

3) Set the parameters of the circuit. R1 and R2 should have the same value and be large so that not much current goes through the recording electrode to change the membrane potential. The Control amplifier gain should be large so that e can be a good virtual ground. Set the battery pulse currents to start at rest (-65 mV), jump to 10mV at 2 ms for 3 ms and then return to rest



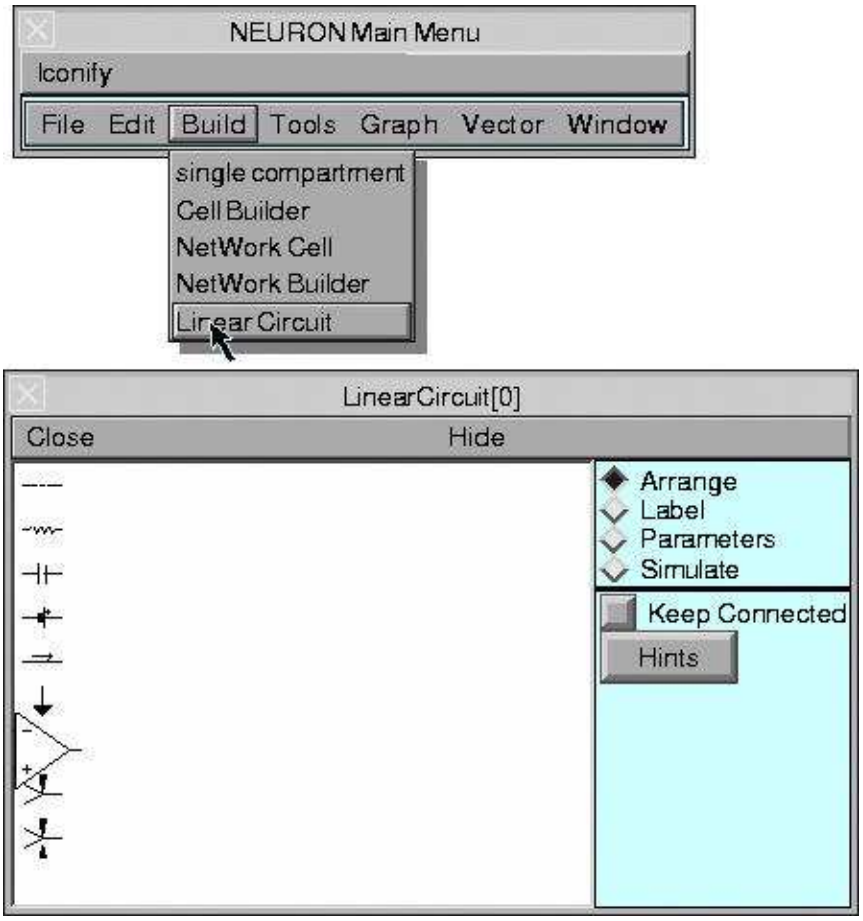
4) Simulate the circuit and plot the Control current and Vm. Also show a space plot of the entire cable.



Completed example

Building and labeling a two electrode voltage clamp with the Linear Circuit Builder

Pop up a Linear Circuit Builder with



The usage style is to

1) With the "Arrange" tool, place the components of the circuit so they are properly connected.

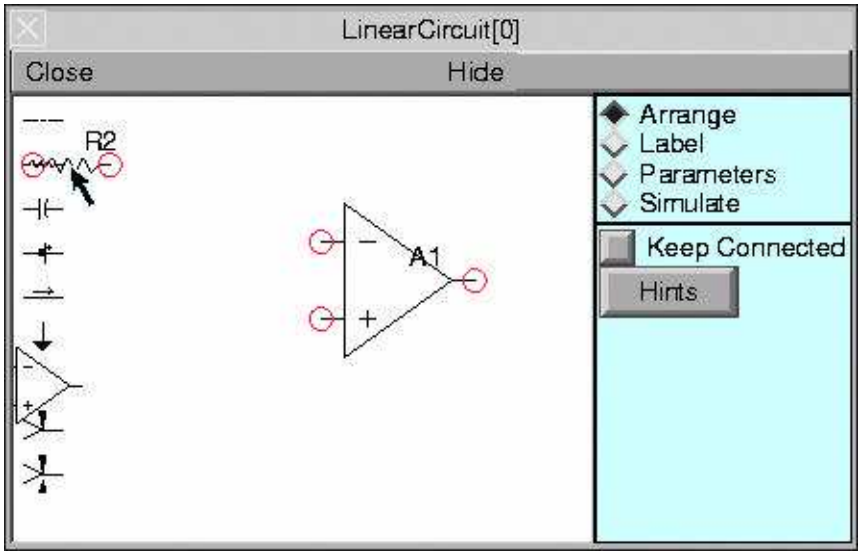
2) With the "Label" tool, rename components and specify the names of important circuit nodes. Move the labels so they are clearly associated with the proper component/node. Also, cell locations, if any, are specified with the "Label" tool.

3) With the "Parameters" tool, specify values for the components and pulse protocols for any batteries and constant current devices.

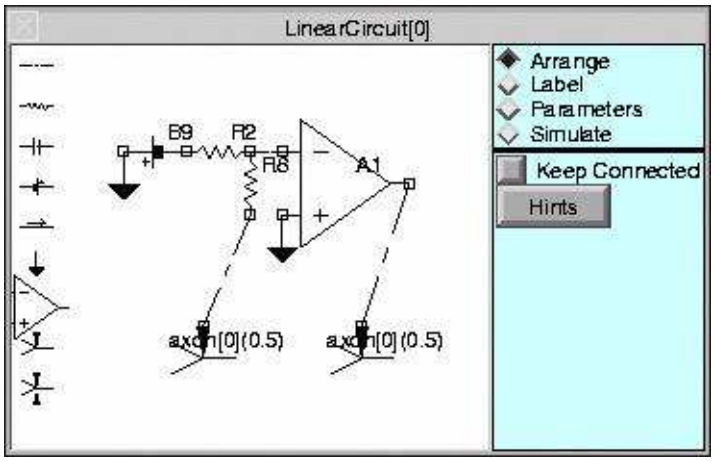
4) With the "Simulate" tool, one specifies states (node voltages and/or internal component states) for plotting. The circuit is not added to the neuron equations unless the "Simulate" tool is active.

Each of these tools has a brief "Hints" button which gives basic usage information.

Arranging the voltage clamp parts on the graph panel is done by selecting parts on the left of the scene which are then dragged to their desired position.

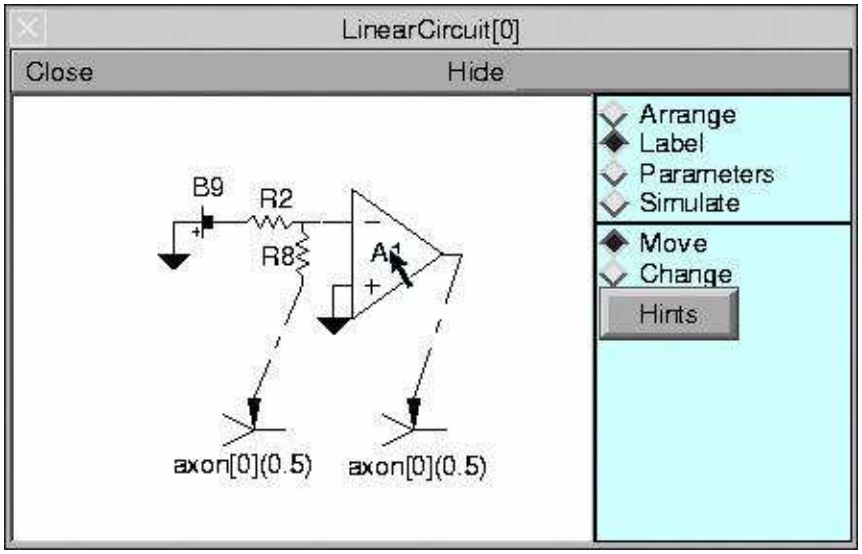


Unconnected ends of components are indicated by red circles. When the ends of two components overlap, a connection is implied and that fact is indicated by a small black square. Selecting the center of a component allows positioning. Selecting and dragging an end of a component allows scaling and rotation. An attempt is made to keep the ends of components on implicit grid points. The completed arrangement of parts

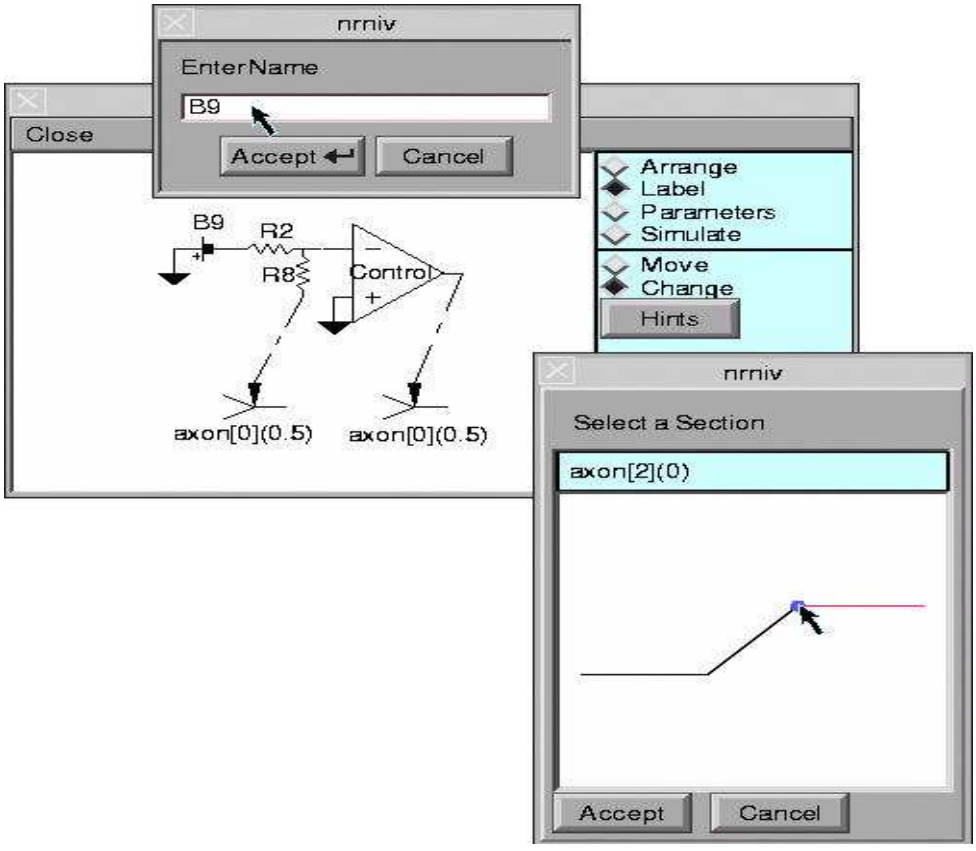


gives a fairly understandable circuit topology. However the components are labeled according to component type with indices in the order in which they were created. Those labels are not pleasingly formatted in relation to the components. The default cell connections are not our desired locations. And important voltage nodes of the circuit are unlabeled. All these problems are overcome with the "Label" tool.

It is probably best to start by moving the existing labels to better locations.



When "Change" is selected, clicking on a component label pops up a string input dialog. Clicking on a cell name pops up a location browser.



State and Parameter Discontinuities in HOC

Physical System

Transient voltage clamp to assess action potential stability.

Model

Force a discontinuous change in potential during an Action Potential

Simulation

To work properly with variable time step methods, models that change states and/or parameters discontinuously during a simulation must notify NEURON when such events take place. This exercise illustrates the kinds of problems that occur when a model is changed without reinitializing the variable step integrator.

1) Start with a current pulse stimulated HH patch. E.g. HH Patch

2) Discontinuously change the voltage by +20 mV using

```
objref fih
fih = new FInitializeHandler("cvode.event(2, \"change()\")")
proc change() {
    print "change at ", t
    v += 20
}
```

Notice the difference between fixed and variable step methods.

3) Replace the "change" procedure with the following and try again.

```
proc change() {
    print "change at ", t
    v += 20
    cvode.re_init()
}
```

4) What happens if you discontinuously change a parameter such as gnabar_hh during the interval 2-3 ms without notifying the variable time step method.

```
objref fih
fih = new FInitializeHandler("cvode.event(2, \"change(1)\")")
proc change() {
    print "change at ", t
    if ($1 == 1) {
        gnabar_hh *= 2
        cvode.event(3, "change(2)")
    } else {
        gnabar_hh /= 2
    }
```



```
    // cvode.re_init // should be here for varstep method
}
```

It will be helpful to use the Crank-Nicholson fixed step method and compare the variable step method with and without the cvode.re_init() . Zoom in around the discontinuity at 2ms.

Extra:

Parameter Discontinuities in NMODL

This older exercise makes use of the deprecated at_time way of notifying cvode of the time for a discontinuity. Nowadays, a NET_RECEIVE block is recommended for dealing with discontinuities but the old exercise still is a good cautionary example of what happens when there is a discontinuity without notifying the variable step method.

Batch runs with bulletin board parallelization

Sooner or later most modelers find it necessary to grind out a large number of simulations, varying one or more parameters from one run to the next. The purpose of this exercise is to show you how bulletin board parallelization can speed this up.

This exercise could be done at several levels of complexity. Most challenging and rewarding, but also most time consuming, would be for you to develop all code from scratch. But learning by example also has its virtues, so we provide complete serial and parallel implementations that you can examine and work with.

> The following instructions assume that you are using a Mac or PC, with at least NEURON 7.1 under UNIX/Linux, or NEURON 7.2 under macOS or MSWin. For UNIX, Linux, or macOS, be sure MPICH 2 or OpenMPI is installed. For Windows, be sure Microsoft MPI is installed. If you are using a workstation cluster or parallel supercomputer, some details will differ, so ask the system administrator how to get your NEURON source code (.py, .ses, .mod files) to where the hosts can use them, how to compile .mod files, and what commands are used to manage simulations.

Physical system

You have a cell with an excitable dendritic tree. You can inject a steady depolarizing current into the soma and observe membrane potential in the dendrite. Your goal is to find the relationship between the amplitude of the current applied at the soma, and the spike frequency at the distal end of the dendritic tree.

Computational implementation

The model cell

The model cell is a ball and stick with these properties:

soma
- L 10 um, diam 3.1831 um (area 100 um^2)
- cm 1 $\mu F/cm^2$, Ra 100 ohm cm
- nseg 1
- full hh

dend
- L 1000 um, diam 2 um
- cm 1 $\mu F/cm^2$, Ra 100 ohm cm
- nseg 25 (appropriate for d_lambda = 0.1 at 100 Hz)
- reduced hh (all conductances /=2)

The implementation of this model is in cell.py

Code development strategy

Before trying to set up a program that runs multiple simulations, it is useful to have a program that executes a single simulation. This is helpful for exploring the properties of the model and collecting information needed to guide the development of a batch simulation program.

The next step is to create a program that performs serial execution of multiple simulations, i.e. executes them one after another. In addition to generating simulation results, it is useful for this program to report a measure of computational performance. For this example the measure will be the total time required to run all simulations and save results. The simulation results will be needed to verify that the parallel implementation is working properly. The performance measure will help us gauge the success of our efforts, and indicate whether we should look for additional ways to shorten run times.

The final step is to create a parallel implementation of the batch program. This should be tested by comparing its simulation results and performance against those of the serial batch program in order to detect errors or performance deficiencies that should be corrected.

In accordance with this development strategy, we provide the following three programs. For each program there is a brief description, plus one or more examples of usage. There are also links to each program's source code and code walkthroughs, which may be helpful in completing one of this exercise's assignments.

Finally, there is a fourth program for plotting results that have been saved to a file, but more about that later.

initonerun.py

Description
- Executes one simulation with a specified stimulus.
- Displays response and reports spike frequency.

Usage
- `python -i initonerun.py`
- A new simulation can be launched by entering the command
- `onerun(x)`
- at the >>> prompt, where x is a number that specifies the stimulus current amplitude in nA.
- Example:
- `onerun(0.3)`

Source
- initonerun.py
- code walkthrough

initbatser.py

Description
- Executes a batch of simulations, one at a time, in which stimulus amplitude increases from run to run.
- Then saves results, reports performance, and optionally plots an f-i graph.

Usage
`python initbatser.py`, or
`python -i initbatser.py` to see the graph.
Source
initbatser.py
code walkthrough

initbatpar.py

Description
Performs the same task as initbatser.py, i.e. executes a batch of simulations, but does it serially or in parallel, depending on how the program is launched. Parallel execution uses NEURON's bulletin board.
Usage
Serial execution: `python initbatpar.py`
runs simulations one after another on a single processor, i.e. serially. Parallel execution: `mpiexec -n N python initbatpar.py`
launches N processes that carry out the simulations. On a multicore PC or Mac, parallel execution with N equal to the number of cores can reduce total run time to about 1/N of the run time required by initbatser.py, serial execution of initbatpar.py, or parallel execution of initbatpar.py with N = 1.
Source
initbatpar.py
code walkthrough

Things to do

1. Run a few simulations with initonerun.py just to see how the cell responds to injected current. You might try to find the smallest and largest stimuli that elicit repetitive dendritic spiking.

2. Compare results produced by serial and parallel simulations, to verify that parallelization hasn't broken anything. For example:

```
python initbatser.py
mv fi.dat fiser.dat
python initbatpar.py
mv fi.dat finompi.dat
mpiexec -n 4 python initbatpar.py
mv fi.dat fimpi4.dat
cmp fiser.dat finompi.dat
cmp fiser.dat fimpi4.dat
```

Instead of cmp, MSWin users will have to use fc in a "Command Prompt" (`cmd`) window.

3. Evaluate and compare performance of the serial and parallel programs.
Here are results of some tests I ran:

```
NEURON 7.5 (266b5a0) 2017-05-22 under Windows Subsystem for Linux
on a quad core desktop.
initbatser     6.16 s
initbatpar
  without MPI  6.03
  with MPI             speedup
  n = 1        6.03    1 (performance baseline)
```



```
2       3.41   1.77
3       2.62   2.30
4       2.05   2.94
```

4. Make a copy of initbatpar.py and edit it, inserting `print` calls that reveal the sequence of execution, i.e. which processor is doing what. These statements should report whatever you think would help you understand program flow. Here are some suggestions for things you might want to report:

- the identity of the host process (i.e. the pc.id)
- the name of the proc or func that is being entered or exited
- the value that is being passed or returned
- the index of the simulation that is being run
- anything else that you think might illuminate this dark little corner of computational neuroscience

Focus on the parts of the program that are involved in the master's principal function (posting jobs and gathering results), and the workers' principal function (entering, executing, and exiting `fi`).

After inserting the print calls, change NRUNS to 3 or 4, then run a serial simulation and see what happens.

Next run parallel simulations with -n 1, 2, 3 or 4 and see what happens. Do the monitor reports make sense?

5. Examine an f-i curve from data saved to one of the dat files.

```
python -i initplotfi.py
```

then use its file browser to select one of the dat files.

Examine initplotfi.py to see how it takes advantage of procs that are built into NEURON's standard run library (UNIX/Linux users see nrn/share/nrn/lib/hoc/stdlib.hoc, MSWin users see c:\nrn\lib\hoc\stdlib.hoc).

Bulletin board code walkthroughs

initonerun.py

Description
: Executes one simulation with a specified stimulus.
Displays response and reports spike frequency.

Usage
: `python -i initonerun.py`
A new simulation can be launched by entering the command
`onerun(x)`
at the >>> prompt, where x is a number that specifies the stimulus current amplitude in nA.
Example:
`onerun(0.3)`

Source
: initonerun.py

Code walkthrough

initonerun.py is organized in a modular fashion. Only highlights are mentioned.

Simulation parameters
: Firing frequency should be determined after the model has settled into a stable firing pattern. Tests show that the first few interspike intervals vary slightly, so the first NSETTLE=5 ISIs are ignored and frequency is computed from the last 10 ISIs in a simulation. The slowest sustained repetitive firing is > 40 Hz (longest ISI < 25 ms), so TSTOP = 375 ms would allow at least 15 ISIs. TSTOP has been set to 500 ms so that repetitive firing produces > 15 ISIs, and runs with < 15 are ignored.

Model specification
: loads the cell's source code

Instrumentation
: stimulus--attaches an IClamp to soma(0.5)
data recording and analysis--uses a NetCon to record the times at which spikes reach dend(1)
get_frequency(spvec) verifies that enough spikes have occurred, then calculates freq from the last NINVL=10 recorded ISIs.

Simulation control and reporting of results
: onerun() expects a single numerical argument that specifies the stimulus amplitude. It creates a graph that will show dend(1).v vs. time, runs a simulation, analyzes the results, and prints out the stimulus amplitude and firing frequency.

initbatser.py

Description
: Executes a batch of simulations, one at a time, in which stimulus amplitude increases from run to run. Then saves results, reports performance, and optionally

plots an f-i graph.

Usage

```
python -i initbatser.py
```

Source

initbatser.py

Code walkthrough

initbatser.py is based on initonerun.py. Only significant differences are mentioned.

Simulation parameters

If PLOTRESULTS is True, an f-i curve will be generated at the end of program execution; if not, the program simply exits when done.
AMP0, D_AMP, and NRUNS specify the stimulus current in the first run, the increment from one run to the next, and the number of simulations that are executed, respectively.

Instrumentation

setparams(run_id) assigns values to the parameters that differ from run to run. In this example, it sets stimulus amplitude to a value that depends on its argument. Its argument is the "run index", a whole number that ranges from 0 to NRUNS-1 (see batchrun(n) in the following discussion of "Simulation control").

Simulation control

This has been separated from reporting of results.
`trun = time.time()` records system time at the beginning of the code whose run time will be evaluated.
batchrun() contains a `for` loop that iterates the run counter run_id from 0 to NRUNS - 1. Each pass through this loop results in a new simulation with a new stimulus amplitude, finds the spike frequency, and saves the stimulus amplitude and frequency to a pair of lists. It also prints a message to the terminal to indicate progress.

Reporting of results

saveresults() writes the stimulus and frequency vectors to a text file in the format used by "NEURON Main Menu / Vector / Save to File" and "Retrieve from File".
After this is done, the program reports run time.
Then it plots an f-i curve or quits, depending on PLOTRESULTS.

initbatpar.py

Description

Performs the same task as initbatser.py, i.e. executes a batch of simulations, but does it serially or in parallel, depending on how the program is launched. Parallel execution uses NEURON's bulletin board.

Usage

Serial execution: `python initbatpar.py`
runs simulations one after another on a single processor, i.e. serially. Parallel execution: `mpiexec -n N python initbatpar.py`
launches N processes that carry out the simulations. On a multicore PC or Mac, parallel execution with N equal to the number of cores can reduce total run time to about 1/N of the run time required by initbatser.py, serial execution of initbatpar.py,

or parallel execution of initbatpar.py with N = 1.

Source

initbatpar.py

Code walkthrough

initbatpar.py is based on initbatser.py. Only key differences are mentioned below.

ParallelContext

An instance of the h.ParallelContext class is created near the start of the program. Print statements inserted after this point to monitor program execution can report not only what code is being executed in the course of which simulation, but also the identity (pc.id) of the host that is executing the code.

Simulation control

This is where most of the changes have been made.

The speedup of bulletin board style parallelization depends on keeping the workers as busy as possible, while minimizing communication (data exchange via the bulletin board) as much as possible. To this end, the master should post as little data as necessary to the bulletin board. The workers should do as much work as possible, and then return as little data as necessary to the bulletin board.

The serial program initbatser.py has a batchrun(n) that uses this `for` loop to execute a series of simulations, one at a time, on a single processor:

```
for run_id in range(n):
    set_params(run_id)
    h.run()
    stims.append(stim.amp)
    freqs.append(get_frequency())
    print('Finished %d of %d.' % (run_id + 1, n))
```

In initbatpar.py, everything that can be offloaded to the workers has been taken out of batchrun() and inserted into a new function `fi(run_id)` that is defined prior to batchrun.

```
def fi(run_id):
    """set params, execute a simulation, analyze and return results"""
    set_params(run_id)
    h.run()
    return (run_id, stim.amp, get_frequency(spvec))
```

Notice that `fi` contains the procedures that involve the most computational overhead. Also notice that `fi` expects a single numerical argument -- the run_id -- and returns a tuple with the run_id, the value of the stimulus, and the frequency obtained from the simulation. An alternative implementation could have reduced communication by returning only the frequency, unpacking the job index (equal to the run_id), and recomputing the stimulus amplitude. It is important to balance convenience, the aim of keeping the workers busy, and minimizing communication overhead.

Here is initbatpar.py's batchrun procedure:


```
def batchrun(n):
    # preallocate lists of length n for storing parameters and results
    stims = [None] * n
    freqs = [None] * n
    for run_id in range(n):
        pc.submit(fi, run_id)
    count = 0
    while pc.working():
        run_id, amp, freq = pc.pyret()
        stims[run_id] = amp
        freqs[run_id] = freq
        count += 1
        print('Finished %d of %d.' % (count, n))
    return stims, freqs
```

There still is a `for` loop, but it uses pc.submit() to post jobs to the bulletin board. Communication is minimized by passing only the function handle (`fi`) and the simulation index run_id for each run that is to be carried out.

Next comes a `while` loop in which the master checks the bulletin board for returned results. If nothing is found, the master picks a task from the bulletin board and executes it. If a result is present, the master retrieves it from the bulletin board: pc.pyret() gets the value returned by `fi`, which is unpacked into its three components.

run_id is used to place the results in the appropriate locations in the stims and freqs lists, as there is no guarantee that simulation results will be returned in any specific sequence.

After the last job has been completed, the master exits the `while` loop, and batchrun is finished. Then pc.done() releases the workers.

But the master still has to save the results.

HOC exercises

// Executable lines below are shown with the hoc prompt // Typing these, although trivial, can be a valuable way to get familiar with the language

oc> // A comment

oc> /* ditto */

Data types: numbers strings and objects

- **anything not explicitly declared is assumed to be a number**

  oc> x=5300 // no previous declaration as to what 'x' is

  - **numbers are all doubles (high precision numbers)**

    there is no integer type in Hoc

  - **Scientific notation use e or E**

    oc> print 5.3e3,5.3E3 // e preferred (see next)

  - **there are some useful built-in values**

    oc> print PI, E, FARADAY, R

- **Do you have anything to declare?: objects and strings**

  - **Must declare an object reference (=object variable) before making an object**

  - **Objref: manipulate references to objects, not the objects themselves**

    often names are chosen that make it easy to remember what an object reference is to be used for (eg g for a Graph or vec for a Vector) but it's important to remember that these are just for convenience and that any object reference can be used to point to any kind of object

- **Objects include vectors, graphs, lists, ...**

  oc> objref XO,YO // capital ’oh’ not zero

  oc> print XO,YO // these are object references

  oc> XO = new List() // ’new’ creates a new instance of the List class

  oc> print XO,YO // XO now points to something, YO does not

  oc> objref XO // redeclaring an objref breaks the link; if this is the only reference to that object the object is destroyed

  oc> XO = new List() // a new *new* List

  oc> print XO // notice the List[#] -- this is a different List, the old one is gone

- **After creating object reference, can use it to point a new or old object**

  oc> objref vec,foo // two object refs

  oc> vec = new Vector() // use ’new’ to create something

  oc> foo = vec // foo is now just another reference to the same thing

  oc> print vec, foo // same thing

  oc> vec=XO

  oc> print vec, foo // vec no longer points to a vector

  oc> objectvar vec // objref and objectvar are the same; redeclaring an objref breaks the link between it and the object it had pointed to

  oc> print vec, foo // vec had no special status, foo still points equally well

- **Can create an array of objrefs**

  oc> objref objarr[10]

  oc> objarr[0]=XO

  oc> print objarr, objarr[0] // two ways of saying same thing

  oc> objarr[1]=foo

  oc> objarr[2]=objarr[0] // piling up more references to the same thing

oc> print objarr[0],objarr[1],objarr[2]

- **Exercises: Lists are useful for maintaining pointers to objects so that they are maintained when explicit object references are removed**

  1. Make vec point to a new vector. Print out and record its identity (*print vec*). Now print using the object name (ie *print Vector[#]* with the right #). This confirms that the object exists. Destroy the object by reinitializing the vec reference. Now try to print using the object name. What does it say.

  2. As in Exercise 1: make vec point to a new vector and use print to find the vector name. Make XO a reference to a new list. Append the vector to the list: {XO.append(vec). Now dereference vec as in Exercise 1. Print out the object by name and confirm that it still exists. Even though the original objref is gone, it is still point to by the list.

  3. Identify the vector on the list: (*print XO.object(0)*). Remove the vector from the list (*print XO.remove(0)*). Confirm that this vector no longer exists.

- **Strings**

- **Must declare a string before assigning it**

  oc> mystr = "hello" // **ERROR**: needed to be declared

  oc> strdef mystr // declaration

  oc> mystr = "hello" // can’t declare and set together

  oc> print mystr

  oc> printf("-%s-", mystr) // tab-string-newline; printf=print formatted; see documentation

- **There are no string arrays; get around this using arrays of String objects**

- **Can also declare number arrays, but vectors are often more useful**

  oc> x=5

  oc> double x[10]

  oc> print x // overwrote prior value

  oc> x[0]=7


```
oc> print x, x[0] // these are the same
```

Operators and numerical functions

```
oc> x=8 // assignment
oc> print x+7, x*7, x/7, x%7, x-7, x^7 // doesn't change x
oc> x==8 // comparison
oc> x==8 && 5==3 // logical AND, 0 is False; 1 is True
oc> x==8 \\ 5==3 // logical OR
oc> !(x==8) // logical NOT, need parens here
oc> print 18%5, 18/5, 5^3, 3*7, sin(3.1), cos(3.1), log(10), log10(10), exp(1)
oc> print x, x+=5, x*=2, x-=1, x/=5, x // each changes value of x; no x++
```

Blocks of code {}

```
oc> { x=7
print x
x = 12
print x
}
```

Conditionals

```
oc> x=8
oc> if (x==8) print "T" else print "F" // brackets optional for single statements
oc> if (x==8) {print "T"} else {print "F"} // usually better for clarity
oc> {x=1 while (x<=7) {print x x+=1}} // nested blocks, statements separate by space
oc> {x=1 while (x<=7) {print x, x+=1}} // notice difference: comma makes 2 args of print
oc> for x=1, 7 print x // simplest for loop
```



```
oc> for (x=1;x<=7;x+=2) print x // (init;until;change)
```

Procedures and functions

```
oc> proc hello () { print "hello" }
oc> hello()
oc> func hello () { print "hello" return 1.7 } // functions return a number
oc> hello()
```

- ## Numerical arguments to procedures and functions

  ```
  oc> proc add () { print $1 + $2 } // first and second argument, then $3, $4...
  oc> add(5, 3)
  oc> func add () { return $1 + $2 }
  oc> print 7*add(5, 3) // can use the returned value
  oc> print add(add(2, 4), add(5, 3)) // nest as much as you want
  ```

- ## String ($s1, $s2, ...) and object arguments ($o1, $o2, ...)

  ```
  oc> proc prstuff () { print $1, "::", $s2, "::", $o3 }
  oc> prstuff(5.3, "hello", vec)
  ```

- ## Exercises

  *** Use printf in a procedure to print out a formatted table of powers of 2

  *** Write a function that returns the average of 4 numbers

  *** Write a procedure that creates a section called soma and sets diam and L to 2 args

Built-in object types: graphs, vectors, lists, files

- # Graph

  oc> objref g[10]

  oc> g = new Graph()

  oc> g.size(5, 10, 2, 30) // set x and y axes

  oc> g.beginline("line", 2, 3) // start a red (2), thick (3) line

  oc> {g.line(6, 3) g.line(9, 25)} // draw a line (x, y) to (x, y)

  oc> g.flush() // show the line

- # Exercises

  - **write proc that draws a colored line ($1) from (0, 0) to given coordinate ($2, $3) assume g is a graph object**
  - **write a proc that puts up two new graphs**
  - **bring up a graph using GUI, on graph use right-button right pull-down to "Object Name"; set 'g' objectvar to point to this graph and use g.size() to resize it**

- # Vector

  oc> objref vec[10]

  oc> for ii=0, 9 vec[ii]=new Vector()

  oc> vec.append(3, 12, 8, 7) // put 4 values in the vector

  oc> vec.append(4) // put on one more

  oc> vec.printf // look at them

  oc> vec.size // how many are there?

  oc> print vec.sum/vec.size, vec.mean // check average two ways

  oc> {vec.add(7) vec.mul(3) vec.div(4) vec.sub(2) vec.printf}

  oc> vec.resize(vec.size-1) // get rid of last value

  oc> for ii=0, vec.size-1 print vec.x[ii] // print values


```
oc> vec[1].copy(vec[0]) // copy vec into vec[1]

oc> vec[1].add(3)

oc> vec.mul(vec[1]) // element by element; must be same size
```

- ## Exercises
  - ### write a proc to make $o1 vec elements the product of $o2*$o3 elements

    (use resize to get $o1 to right size; generate error if sizes wrong eg

    if ($o2.size!=$o3.size) { print "ERROR: wrong sizes" return }

  - ### graph vector values: vec.line(g, 1) or vec.mark(g, 1)

    play with colors and mark shapes (see doc for details)

  - ### graph one vec against another: vec.line(g, vec[1]); vec.mark(g, vec[1])

  - ### write a proc to multiply the elements of a vector by sequential values from 1 to size-1

    hint: use vec.resize, vec.indgen, vec.mul

File

```
oc> objref file

oc> mystr = "AA.dat" // use as file name

oc> file = new File()

oc> file.wopen(mystr) // ’w’ means write, arg is file name

oc> vec.vwrite(file) // binary format

oc> file.close()

oc> vec[1].fill(0) // set all elements to 0

oc> file.ropen(mystr) // ’r’ means read
```



```
oc> vec[1].vread(file)

oc> if (vec.eq(vec[1])) print "SAME" // should be the same
```

- **Exercises**
  - **proc to write a vector ($o1) to file with name $s1**
  - **proc to read a vector ($o1) from file with name $s1**
  - **proc to append a number to end of a file: tmpfile.aopen(), tmpfile.printf**

List

```
oc> objref list

oc> list = new List()

oc> list.append(vec) // put an object on the list

oc> list.append(g) // can put different kind of object on

oc> list.append(list) // pointless

oc> print list.count() // how many things on the list

oc> print list.object(2) // count from zero as with arrays

oc> list.remove(2) // remove this object

oc> for ii=0, list.count-1 print list.object(ii) // remember list.count, vec.size
```

- **Excercise**
  - **write proc that takes a list $o1 with a graph (.object(0)) followed by a vector (.object(1)) and shows the vector on the graph**
  - **modify this proc to read the vector out of file given in $s2**

Simulation

```
oc> create soma
```



```
oc> access soma
oc> insert hh
oc> ismembrane("hh") // make sure it's set
oc> print v, v(0.5), soma.v, soma.v(0.5) // only have 1 seg in section
oc> tstop=50
oc> run()
oc> print t, v
oc> print gnabar_hh
oc> gnabar_hh *= 10
oc> run()
oc> print t, v // what happened?
oc> gnabar_hh /= 10 // put it back
```

Recording the simulation

```
oc> cvode_active(0) // this turns off variable time step
oc> dt = 0.025
oc> vec.record(&soma.v(0.5)) // '&' gives a pointer to the voltage
oc> objref stim
oc> soma stim = new IClamp(0.5) // current clamp at location 0.5 in soma
oc> stim.amp = 20 // need high amp since cell is big
oc> stim.dur = 1e10 // forever
oc> run()
oc> print vec.size()*dt, tstop // make sure stored the right amount of data
```

Graphing and analyzing data

```
oc> g=new Graph()
```



```
oc> vec.line(g, dt, 2, 2)

oc> g.size(0, tstop, -80, 50)

oc> print vec.min, vec.max, vec.min_ind*dt, vec.max_ind*dt

oc> vec[1].deriv(vec, dt)

oc> print vec[1].max, vec[1].max_ind*dt // steepest AP
```

- **Exercises**
  - **change params (stim.amp, gnabar_hh, gkbar_hh), regraph and reanalyze**
  - **bring up the GUI and demonstrate that the GUI and command line control same parameters**
  - **write proc to count spikes and determine spike frequency (use vec.where)**

Roll your own GUI

```
oc> proc sety () { y=x print x }

oc> xpanel("test panel")

oc> xvalue("Set x", "x")

oc> xvalue("Set y", "y")

oc> xbutton("Set y to x", "sety()")

oc> xpanel()
```

- **Exercises**
  - **put up panel to run sim and display (in an xvalue) the average frequency**

Last updated: Jun 16, 2003 (11:09)

Multithread parallelization for better performance on multicore workstations

If a model has more than a few thousand states, it may run faster with multiple threads.

Physical System

Purkinje Cell

Model

Miyasho et al. 2001

Simulation

Launch the model, let it run for about 30 seconds of wall time and stop it.
What ms of simulation time did it get to?

Pop up the NEURONMainMenu / Tools / ParallelComputing panel. How large is the model? (The "Total model complexity" value is approximately the number of states in the model). This is a large model that should be able to run faster on a multicore workstation if we can parallelize the simulation using threads.

Press the "Refresh" button to see how many useful processors your machine has. May have to press it several times to get a stable number. The value is determined by how much time it takes N threads to count to 1e8. If N is greater than the number of cores on your machine, then the total time will go up. Enter the number of processors into the "#Threads" field. Thread performance has a chance of being good only if the "Load imbalance" is less than 20%. That can only happen if there are more cells than threads and the cells can be distributed so that the total complexity on each thread is about the same. Here, there is only one cell so we have to split the cell into pieces and put the pieces into different threads. This is done by pressing the "Multisplit" button. On my 4-core computer, it splits the cell into 26 pieces and distributes the pieces on 4 threads for a load imbalance of just 4%. Unfortunately, an error message is printed to the terminal window saying:

```
cad is not thread safe
```

A look at the NMODL translator messages shows that not all of the mod files are threadsafe. We need to repair those mod files (cells that use a non-threadsafe mechanism are placed onto the main thread unless you force them onto a different thread, as above, in which case NEURON will generate an error message). A script to aid in the repair is called "mkthreadsafe" and is run in a bash terminal window. On mswin machines, start a bash terminal using the rxvt icon. When executed in the prknj directory it first complains about

```
        VERBATIM
        return 0;
        ENDVERBATIM
Translating K22.mod into K22.c
Notice: VERBATIM blocks are not thread safe
...
Force THREADSAFE? [y][n]: n
```


This VERBATIM block is an old and never needed idiom that some people use to return a value from a PROCEDURE or FUNCTION. You can edit the K22.mod file to remove it but it does not affect thread safety so you can type

```
y
```

so that the script adds the THREADSAFE keyword to the NEURON block. The script continues with the same messages for K23.mod, K2.mod, KC2.mod, KC3.mod, KC.mod, for which it is safe to type y. Something new comes up with

```
NEURON {
        SUFFIX Khh
        USEION k WRITE ik
        RANGE   gk,  gkbar, ik
        GLOBAL  ninf, nexp
}
Translating Khh.mod into Khh.c
Notice: This mechanism cannot be used with CVODE
Notice: Assignment to the GLOBAL variable, "nexp", is not thread safe
Notice: Assignment to the GLOBAL variable, "ninf", is not thread safe
Warning: Default 37 of PARAMETER celsius will be ignored and set by NEURON.
Force THREADSAFE? [y][n]: n
```

This is an even more common idiom that uses global variables to save space. I.e A block calls a rate procedure that computes rate values and temporarily stores them for use later in the block. The assumption was that between assignment and use, no other instance of the model assigns a value to those variables. That assumption is false when there are multiple threads. Type "y" for this case as well. The script will add the THREADSAFE keyword to the NEURON block of the mod file which will cause GLOBALs that are assigned values to become thread variables. That was the last problem mentioned by the script. Unfortunately, there is one other problem in CalciumP.mod which is not tested by the script and you will continue to get the "cad is not thread safe" error if you launch the model. The problem is

```
        SOLVE state METHOD euler
```

I never bothered to make euler thread safe since the best practical methods are "cnexp" for hh-like equations and "derivimplicit" for all the others. So change the "euler" to "cnexp" manually in CalciumP.mod .

Now one should build the dll as normally done on your machine and try the "Parallel Computing" tool again. My computer runs the model in 76s with one thread and 12s with 4 threads. The reason for the superlinear speedup is that multisplit forces "Cashe Efficient" on. It is often worthwhile turning that on even with a single thread (in my case, 49s).

Note: Multisplit, divides the cell into many independent cells which are connected together internally (check with "h.topology()"). When divided into pieces the cell as a whole is difficult to deal with (for example, h.distance() and Shape tools don't work well. Even h.topology() gives an incomplete idea of what is going on). So it is best to turn off "Multisplit" to re-assemble the cell to its original condition before doing any GUI manipulation.

Let's try another case using a network model.

Physical System

Cortex integrates sensory information. What is a moment in time?

Model

Transient synchrony. Hopfield and Brody 2001 implemented by Michele Migliore.

Simulation

This model has a home-brew interface that does not show elapsed walltime, but to run and time the "before training" simulation one can copy-paste the following into the Python prompt:

```
import time
from neuron import h, gui

h.load_file('mosinit.hoc')

start_time = time.time()
h.run_u()
print('wall time: {}s'.format(time.time() - start_time))
```

This model also has non-threadsafe mechanisms. So we need to repair with mkthreadsafe (Another case of using GLOBAL variables for temporary storage.) However, running a sim with two threads gives an error

```
...usable mindelay is 0 (or less than dt for fixed step method)
```

Sadly, threads cannot be used when any NetCon.delay is 0. Fortunately, this model is not critically sensitive to the delay, so try again by setting all delays to 0.5ms . (Copy-paste the following into the Python terminal)

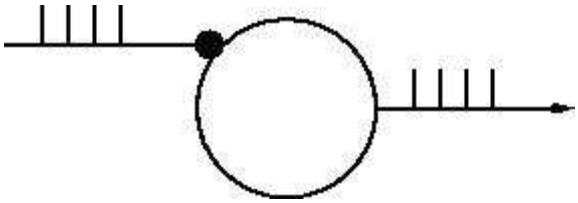
```
for netcon in h.List('NetCon'):
    netcon.delay = 0.5
```

With two threads the run will be faster, but far from twice as fast. Try again with "Cache Efficient" checked.

Introduction to the Network Builder

The Network Builder tools are principally of didactic value, in that the graphical creation of simple networks can generate "readable" hoc code which in turn embodies our notion of a fairly flexible programming style for construction of more complicated networks.

Model



Artificial Integrate and Fire cell stimulated by a burst of action potentials. This is a "hello world" level exercise that shows how to use the Network Builder and supporting tools to create artificial cell types, use those types to create cell objects, and connect the cell objects together. A subsequent exercise discusses how to create network ready cell types from more complicated cells specified by the cell builder.

Simulation

The strategy is to 1) define the types of cells (and stimulators), 2) create each cell in the network, 3) connect the cells together, 4) specify parameters such as delays and connection weights, 5) run a simulation and plot the input and output spike trains. If you have trouble with the following instructions, this executes a working exercise.

Define artificial cell types

What is an artificial cell in NEURON?

Start an ArtCellGUI tool by selecting the menu item:
NEURONMainMenu/Build/NetworkCell/ArtificialCell

Create a "C" type via a New/IntFire1 followed by a Rename. This type will be used to create our model cell.

Create an "S" type via a New/NetStim followed by a Rename. We will use this type later on to create a stimulator. The interval and number parameters should be set to 10 and 10 respectively.

The interface should now look something like this.

Cell creation

Start a NetGUI tool by selecting the menu item: NEURONMainMenu/Build/NetworkBuilder

The NetGUI instance should have its "Locate" radiobutton selected

Drag an "S" into the middle of the canvas. This creates an "S0". The text in the canvas explains what is happening during dragging.

Drag a "C" into the canvas. This creates a "C1".

The NetGUI window should now look something like this but won't have the explanatory text in the canvas unless you just selected the "Locate" radio button.

Note that cells are indexed according to the order of creation starting at 0. Dragging a created cell off the canvas destroys it (and reduces the indices of cells greater than that index by 1. That is, the largest index is total number of cells - 1. Dragging a new cell onto an existing cell replaces the existing cell and its index is unchanged. Currently, replacing a cell destroys all the connections to that cell -- possibly the wrong gui behavior when cells have only one possible input connection point.

At this point, what you see in the NetworkBuilder is just a specification and the cells don't yet exist. The pointprocesses aren't created (and therefore the network can't be simulated) until the "Create" button is pressed. After "Create" is pressed, no new cell type can be added to the NetGUI. Within the confines of the existing types, any number of cells can be created or destroyed along with their connections and the "real (simulatable)" network will be constantly updated to reflect the NetGUI specification.

It is a good idea to save the NetGUI tool often (or at least if substantial effort has gone into changing it since it was last saved).

Connections

In the NetGUI window, select the "Src->Tar" radiobutton and drag a line from S0 to C1.

Use the "Weights" button to pop up a NetEdgeGUI panel. With this panel, selecting an item on the left list places the number in the field editor. Selecting an item in the right list assigns the field editor value to its connection weight. This makes it convenient to assign the same weight to a subset of connections.

Enter "2" into the field editor.

Click on the right list item labeled "S0->C1 0". The label should change to "S0->C1 2". This weight is large enough so that every input event (from S0) should elicit an output event from C1

The NetGUI and NetEdgeGUI windows should now look something like this.

Simulation

In the NetGUI window, press the "Create" button.

Press the "SpikePlot" button to pop up a plot window.

Start a NEURONMainMenu/Tools/RunControl and VariableStepControl. Set TStop to 1000 and invoke "Use variable dt"

Do an Init&Run. You should see spikes in the SpikePlot graph.

The relevant windows should now look something like this.

Compare the discrete event simulation run time with the fixed step method ("Use variable dt" turned off).

Other Simulation Exercises

Reduce the fast interspike interval of the stimulus to 2 ms. Why are spikes missing from the C1 output?

For the remaining exercises, set the fast interspike interval back to 10.

For the C cell type, set the integration time constant to 100 ms and set the input connection weight to 0.2 . Observe the output spike train and its relation to the input train.

Plot the value of the state variable `m` in the IntFire1 pointprocess that implements the C1 cell. Why does `m` remain constant between events instead of decaying exponentially (even with the fixed time step)?

> Note: The "ShowCellMap" button in the NetGUI panel helps identify the actual C_IntFire1 object instance that contains the "pp" public object reference to the actual IntFire1 point process object instance (which is located in the dummy section, `acell_home_`) .

The automatically generated hoc code

It's one thing to manage a few cells and connections, quite another to manage thousands. Some help in this task is provided by functions that return various kinds of NetCon lists, e. g. all the `NetCon`s that connect to the same postcell, post synaptic point process, precell, etc. However, at this time there are no generic gui tools to view or manage large networks and it is necessary to craft viewing, control, and management routines based on the details of the particular network being investigated.

Specifying large networks practically requires interpreter programming and this in turn requires familiarity with a programming style suitable for conceptual control of such networks. The hoc code generated by the NetGUI tool for small networks can be used as a basic pattern and a large part of it re-used for the construction of larger networks involving procedural specification of network architecture with random connections, weights, etc. Certainly the cell types are re-usable as is and without change. Those of you with an interest in networks that are beyond the scope of the current NetGUI tool should save the above NetGUI specification as a hoc file and look at it with an editor. Hoc files are constructed using the "Hoc File" button in the NetGUI window. The ideas involving cell templates, the cell creation procedure (cell_append), and the connection procedure (nc_append) have wide applicability.

Cell templates are probably essential programming practice with regard to simulation of large numbers of non-artificial cells. Each template has a position procedure to set the 3-d location of the cell and public position coordinates x, y, z. The connect2target function creates a NetCon with "this" as the source, and the first objref argument as the target.

The hoc file for the above spec looks like this.

In NEURON, artificial cells are point processes that serve as both a target and source object for a network connection object. That they are targets mean that they have a NET_RECEIVE block which handles discrete event input streams through one or more NetCon objects. That they are also sources means that the NET_RECEIVE block also generates discrete output events which are delivered through one or more NetCon objects. Generally, such cells are computationally very efficient (hundreds of times faster than cells we have simulated up to now whose voltage response is a consequence of membrane conductance) because their computation time does not depend on the integration step, dt, but only on the number of events. That is, handling 100000 spikes in one hour for 100 cells takes the same time as handling 100000 spikes in 1 second for 1 cell. The total computation time is proportional to the total number of spikes delivered during a run and is independent of the number of cells or number of connections or interval between spikes.

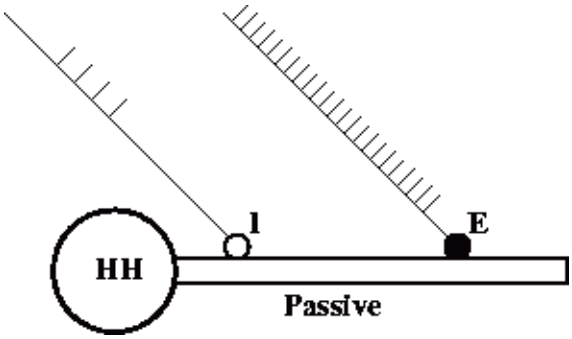
NEURON has four built-in point process classes which can be used to construct artificial cell types:

1. NetStim produces a user-specified train of one or more output events, and can also be triggered by input events
2. IntFire1, which acts like a leaky integrator driven by delta function inputs. That is, the state variable m decays exponentially toward 0. Arrival of an event with weight w causes an abrupt change in m. If m exceeds 1, an output event is generated and the cell enters a refractory period during which it ignores further inputs. At the end of the refractory period, m is reset to 0 and the cell becomes responsive to new inputs.
3. IntFire2, a leaky integrator with time constant taum driven by a total current that is the sum of
   { a user-settable constant "bias" current }
   plus
   { a net synaptic current }.
   Net synaptic current decays toward 0 with time constant taus, where taus > taum (synaptic current decays slowly compared to the rate at which "membrane potential" m equilibrates). When an input event with weight w arrives, the net synaptic current changes abruptly by the amount w.
4. IntFire4, with fast excitation current (rises abruptly, decays exponentially) and slower alpha function like inhibition current that is integrated by even slower membrane.

NEURON requires that all point processes be located in a section. To meet this (in this context, conceptually irrelevant) requirement, the Network Builder tool locates each point process of its instantiated artificial cells in the dummy section called `acell_home_`

Network ready cells from the CellBuilder

Model



Ball-Stick model cell with distal excitation and proximal inhibition. This is another "hello world" level exercise. It shows how to use the a specification from a cell builder to create a network ready cell (spike initiation site and a set of synapses) from a NetReadyCellGUI. It shows how to use these cell types in a NetGUI to make connections between spike initiation sites and synapses.

Simulation

The strategy is to
1) Use a CellBuilder window to create a cell type with specific morphology and membrane properties. The CellBuilder also specifies the type name and the spike initiation site.
2) Define synapse types with a SynTypeGUI. E.g. inhibitory and excitatory synapses from suitable PointProcesses.
3) Define a network ready cell type with a NetReadyCellGUI. I.e. specify where instances of the synapse types should be located on the cell. The NetReadyCellGUI references a CellBuilder to obtain the basic cell morphology and membrane properties. It references a SynTypeGUI which is used to obtain the synapse types.
4) Use a NetGUI to construct the network
5) Run a simulation and plot the input and output spike trains. If you have trouble with the following instructions, this executes a working exercise --- the NetReadyCellGUI (for the ball stick cell) and ArtCellGUI (for the stimulators) are in the "cell types" window group.

Ball-Stick cell model

Start with the ball-stick specification in course/net2/start.ses . This model has
soma area = 500 um2 with hh channels (standard density)
dendrite L = 200 um, diam = 1 um with pas channel (g_pas = .001 S/um2 and e_pas = -65 mV)
Ra = 100 ohm-cm
cm = 1 uF/cm2
and uses d_lambda=.1 compartmentalization policy.

From the CellBuild Management/CellType panel, the classname should be declared as "BallStick"

and the output variable which is watched for spike event should be soma.v(1). At this point the windows should look something like this.

Ball-Stick cell model with synapses.

The NEURONMainMenu/Build/NetworkCell/FromCellBuilder menu item starts a dialog box which requests references to a "Synapse type set" (left list) and a "CellBuild type" (right list). The synapse type set list is empty now but the CellBuild type list should have a reference to the cell builder previously loaded. Select "CellBuild[0]" and press the "Use Selection" button.

This starts a NetReadyCellGUI window (nrc) and a SynTypeGUI window. (It would have also started a new CellBuild window as well if no CellBuild type had been selected in the dialog). Note that the nrc contains a drawing of the cell topology. At this time you can close the CellBuild window --- It can always be re-created with the nrc's Info menu. In fact, when saving the nrc in a session, it is best to first close both the cell builder and the SynTypeGUI to avoid saving duplicate copies in the session file. After closing the CellBuild[0] window the interface should look something like this.

From the SynTypeGUI window, create a "E" synapse type via a New/ExpSyn followed by a Rename. Since the reversal potential for the standard ExpSyn is 0 mV, it is already excitatory. However, change the time constant from 0.1 to 2 ms.

In the same SynTypeGUI window create a "I" synapse type via a New/ExpSyn followed by a Rename. Set the reversal potential "e" to -80 mV so that it will be inhibitory and set tau to 5ms.

In the NetReadyCellGUI, press the Refresh button so that the new SynTypes appear.
Change the cell name to "B" so the label won't take up so much space later on when we use it in a NetGUI tool.

In the NetReadyCellGUI, press the "Locate" radiobutton and drag an E to location .8 on the dendrite. Then drag an I to location .1 on the dendrite. The label in the canvas will show whether the synapse type is close enough to be attached or not. Each synapse on the cell is given an index which is the order of creation. Several synapses can be attached to the same location. The synapse label can be dragged up to two font sizes above or below the location to avoid label overlap. If a label is dragged too far away from the cell it will become detached and the larger synapse indices will be reduced by 1. The interface at this point should look something like this. Enough work has been done up to this point so that you should save the NetReadyCellGUI in a session by itself (without the CellBuild or SynTypeGUI windows -- these may safely be closed as well).

This cell type is now ready for use in the NetGUI.

Stimulators

In analogy with the previous hands-on exercise create two stimulus types, "SE" and "SI" to provide event streams to stimulate the ball-stick model.
For SE, set interval=5 , number=50 , and start=0.
For SI, set interval=10, number=5, and start=20.
I.e. from NEURONMainMenu/Build/NetworkCell/ArtificialCell get an ArtCellGUI and use NetStim to define the stimulus types. After setting it up the window will look something like this.

At this point I created a Window Group called "cell types", placed the ArtCellGUI and

NetReadyCellGUI in it, and saved the group. In case things go wrong I can easily return to this point.

Cell creation

Start a NetGUI tool and create a "B0" ball-stick cell and "SE1" and "SI2" stimulators as shown in this picture

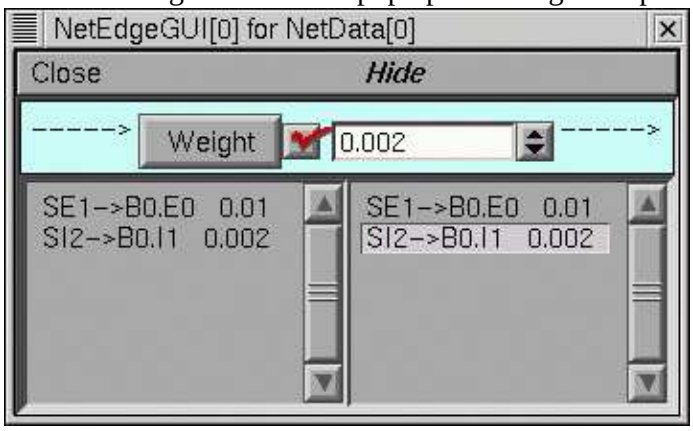
Connections

In the NetGUI window, select the "Src->Tar" radiobutton and drag a line from SE1 to B0. The string near the top of the canvas describes the operation to be performed when the mouse button is released. When the connection line gets near B0 a picture of the BallStick topology will be drawn and the mouse should be moved to the E0 synapse label. The following three figures illustrate the process.
Select the source cell
Select the target cell
Select the synapse on the target cell

Connect SI1 to I1 of B0

The NetGUI window should now look something like this.

Use the "Weights" button to pop up a NetEdgeGUI panel and enter the following weights.



Simulation

In the NetGUI window, press the "Create" button. The `topology()` statement should produce

```
oc>topology()

|-|       acell_home_(0-1)
|-|       B_BallStick[0].soma(0-1)
   `------|       B_BallStick[0].dend(0-1)

        1
oc>
```


Press the "SpikePlot" button to pop up a plot window.

Start a NEURONMainMenu/Tools/RunControl and VariableStepControl. Set TStop to 500 and invoke "Use variable dt"

Get a voltage style graph window and plot the soma voltage for the ball-stick cell. The ShowCellMap button on the NetGUI is useful here. The relevant interface looks something like this

Do an Init&Run.

The relevant windows should now look something like this.

Other Simulation Exercises

Plot the value of the conductance and current of the inhibitory synapse.

The automatically generated hoc code

The hoc file for the above spec looks like this.

Synchronization network model

The exercises here are intended primarily to familiarize the student with techniques and tools useful for the implementation of networks in NEURON. We have chosen a relatively simple network, very loosely based on Hopfield-Brody, in order to minimize distractions due to the complexities of channel kinetics, dendritic trees, detailed network architecture, etc. The model is described in two files: net.py and intfire.mod.

The core of the network consists of artificial integrate-and-fire cells without channels or compartments. This is implemented using an `ARTIFICIAL_CELL` defined in intfire.mod and wrapped in the `Cell` class in net.py. Within the core network, there is only one kind of cell, so there are no issues of organizating interactions between cell populations. All synapses within the core network are inhibitory. (Hopfield-Brody, by contast, uses a mix of inhibitory and excitatory cells).

A single additional cell with Hodgkin-Huxley dynamics, receiving input from all the integrate-and-fire cells, is used as a way to measure network synchrony (it fires when it receives enough inputs within a narrow enough time window).

As you know, NEURON is optimized to handle the complex channel and compartment simulations that have been omitted from this exercise. The interested student might wish to convert this network into a network of spiking cells with realistic inhibitory interactions or a hybrid network with both realistic and artificial cells. Such an extended exercise would more clearly demonstrate NEURON's advantages for performing network simulations.

Although this is a minimal model, learning the ropes is still difficult. Therefore, we suggest that you go through the entire lesson relatively quickly before returning to delve more deeply into the exercises. Some of the exercises are really more homework projects.

Methods

Standard integrate-and-fire implementation (e.g. intfire1.mod)

The basic intfire implementation in neuron utilizes a decaying state variable (*m* as a stand-in for voltage) which is pushed up by the arrival of an excitatory input or down by the arrival of an inhibitory input ($m = m + w$). When *m* exceeds threshold the cell "fires," sending events to other connected cells.

```
    if (m>1) {  ...
        net_event(t)    : trigger synapses
```

`IntIbFire` in sync model

The integrate-and-fire neuron in the current model must fire spontaneously with no input, as well as firing when a threshold is reached. This is implemented by utilizing a *firetime()* routine to calculate when the state variable *m* will reach threshold assuming no other inputs during that time. This firing time is calculated based on the natural firing interval of the cell (*invl*) and the time constant for state variable decay (*tau*). When an input comes in, a new firetime is calculated after taking into account the synaptic input ($m = m + w$) which perturbs the state variable's trajectory towards threshold.

Cell class

`IntIbFire` is wrapped by the class `Cell`. An instantiation of this class provides access to the underlying mechanism through its `dynamics`property.

Network

The network has all-to-all inhibitory connectivity with all connections set to equal non-positive values (initially 0). The network is initially set up with fast firing cells at the bottom of the graph (Cell[0], highest natural interval) and slower cells at the top (Cell[ncell-1], lowest natural interval). Cells in between have sequential evenly-spaced periods.

How it works

The synchronization mechanism requires that all of the cells fire spontaneously at similar frequencies. It is obvious that if all cells are started at the same time, they will still be roughly synchronous after one cycle (since they have similar intrinsic cycle periods). After two cycles, they will have drifted further apart. After many cycles, differences in period will be magnified, leading to no temporal relationship of firing.

The key observation utilized here is that firing is fairly synchronized one cycle after onset. The trick is to reset the cells after each cycle so that they start together again. They then fire with temporal differences equal to the differences in their intrinsic periods. This resetting can be provided by an inhibitory input which pushes state variable m down far from threshold (hyperpolarized, as it were). This could be accomplished through an external pacemaker that reset all the cells, thereby imposing an external frequency onto the network. The interesting observation in this network is that pacemaking can also be imposed from within, though an intrinsic connectivity that synchronizes all members to the will of the masses.

Exercises

- **Run the model.**
  Compile the mechanism with `nrnivmodl` or otherwise, then run `python -i net.py` Notice how without any synaptic connections in the network, every cell fires at its own period and the output cell is quiet.

- **Read the code.**
  Try to understand how it is generating the figures you see, and what options can be easily changed.

- **Increase the inhibition and see synchrony emerge.**
  Try running:

```
network.weight = -0.005
h.finitialize(-65)
h.continuerun(1000)
plot_raster_and_output_mv(network, t, output_v)
```

  Repeat with `network.weight = -0.05` and `network.weight = -0.5`. What happens to the network? What happens to the output cell?

- **Narrow the difference between fast and slow cells so as to make more of them fire.**
  In the `network.weight = -0.5` case, you may have noticed that many neurons do not fire. These have periods that are too long -- before they can fire, the population has fired again and reset them. Notice that the period of network firing is longer than the natural periods of the individual cells. This is because the threshold is calculated to provide this period when m starts at 0. However, with the inhibition, m starts negative.

- **Modify delay instead to acheive the same effect.**
- **Increase cell time constant for a moderate network weight.**
  This will destroy synchrony. Increase inhibitory weight; synchrony recovers. This is a consequence of the exponential rise of the state variable. If the interval is short but the time constant long, then the cell will amplify small variations in the amount of inhibition received.

- **Save the spikes as JSON.**
  Each `Cell` instance in `network.cells` stores its spike times in its `_spike_times` property as a NEURON Vector. Build a list or dictionary of spike times by cell (convert the Vector objects to a Python list) and save it to a file with indentation. Open the file in a text editor and confirm that it matches your expectations.

- **Load the spike time data from JSON and plot.**
  Make a version of `plot_raster` that loads the saved JSON spike time data and plots it. Confirm that it matches the original graph.

- **Restructure the spike times data.**
  Convert it to two lists: one listing every spike time, in order, and one listing the corresponding cell id. Plot this and compare with the original to make sure they are the same. As a check, note that the length of each of the new lists is equal to the sum of the lengths of the per-cell spike time lists.

- **Quantify synchrony.**
  The readout neuron gives us one way of looking at synchrony. Implement a more direct approach based on the spike times. One possible metric: look at how evenly distributed the set of all spikes are; the closer the spikes from all the cells are to being distributed uniformly, the less synchronous the network.

- **Plot synchrony as a function of synaptic weight, synaptic delay, and jointly of the two.**
  Use your metric from the previous question.

- **Alter the network connectivity.**
  As implemented, the network uses all-to-all connectivity. Experiment with different connectivity schemes, e.g. connecting according to a given probability (use Random123), connecting to the nearest *n* cells, etc, and assess how this affects synchrony.

- **Build functions to analyze your new connectivity.**
  Plot the connectivity. Calculate distribution of synaptic convergence and divergence. Find the set of all pairs of neurons with reciprocal synaptic connections (i.e. A→B and B→A). Highlight the reciprical synapses in your connectivity plots.

- **Modify probabilistic wiring to not include reciprocal synapses.**
- **Replace the use of `IntIbFire` with Hodgkin-Huxley kinetics.**
  This will require modifying the `connect` method as well as adding a section and the corresponding dynamics. Are you able to do this and maintain the basic synchronization properties we have seen?

intfire.mod

```
: dm/dt = (minf - m)/tau
: input event adds w to m
: when m = 1, or event makes m >= 1 cell fires
: minf is calculated so that the natural interval between spikes is invl

NEURON {
  ARTIFICIAL_CELL IntIbFire
  RANGE tau, m, invl
  : m plays the role of voltage
}

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

ASSIGNED {
  m
  minf
  t0(ms)
}

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

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

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



```
FUNCTION firetime()(ms) { : m < 1 and minf > 1
  firetime = tau*log((minf-m)/(minf - 1))
:  printf("firetime=%g\n", firetime)
}
```

net.py

```
from neuron import h
from matplotlib import pyplot, gridspec

h.load_file('stdrun.hoc')


class Cell:
  '''Integrate and Fire cell'''
  def __init__(self, _id, interval, tau=10):
    self.id = _id
    self.dynamics = h.IntIbFire()
    self._netcons = []
    self._spike_detector = h.NetCon(self.dynamics, None)
    self._spike_times = h.Vector()
    self._spike_detector.record(self._spike_times)
    self.dynamics.tau = tau        # ms
    self.dynamics.invl = interval  # ms

  def connect(self, other, weight=-0.3, delay=4):
    '''connects self -> other'''
    if isinstance(other, Cell):
      connection = h.NetCon(self.dynamics, other.dynamics)
    else:
      connection = h.NetCon(self.dynamics, other)
    connection.weight[0] = weight
    connection.delay = delay
    self._netcons.append(connection)
    return connection


class Network:
  '''a network'''
  def __init__(
      self,
      num_cells=100,
      interval_low=10,
      interval_high=15,
      weight=-0.005,
      delay=10,
      tau=10):
    '''create a network of size num_cells'''
    dinterval = (interval_high - interval_low) / float(num_cells)
    self.cells = [
      Cell(i, interval_low + i*dinterval, tau) for i in range(num_cells)
    ]
    self.wire(weight, delay)
```



```
  def wire(self, weight=-0.005, delay=10):
    '''all-to-all (non-self) connectivity'''
    for pre in self.cells:
      for post in self.cells:
        if pre != post:
          pre.connect(post, weight, delay)

  def set_weight(self, new_weight):
    '''set all the synaptic weights'''
    for cell in self.cells:
      for nc in cell._netcons:
        nc.weight[0] = new_weight

  def set_delay(self, new_delay):
    '''set all the synaptic delays'''
    for cell in self.cells:
      for nc in cell._netcons:
        nc.delay = new_delay

  weight = property(fset=set_weight, doc='synaptic weights')
  delay = property(fset=set_delay, doc='synaptic delay')


def plot_raster(ax, network):
  '''uses pyplot to plot the spike times.'''
  for cell in network.cells:
    if cell._spike_times:
      ax.vlines(cell._spike_times, cell.id - 0.45, cell.id + 0.45)

h.cvode_active(True)


def create_readout_cell(network, weight=1e-5, delay=4, tau=5):
  output_cell = h.Section(name='output_cell')
  output_cell.L = output_cell.diam = 10
  output_cell.insert('hh')
  syns = []
  ncs = []
  for cell in network.cells:
    syn = h.ExpSyn(output_cell(0.5))
    syn.e = 0
    syn.tau = tau
    nc = h.NetCon(cell.dynamics, syn)
    nc.weight[0] = weight
    nc.delay = delay
    syns.append(syn)
    ncs.append(nc)
  return output_cell, (syns, ncs)
```



```
def plot_raster_and_output_mv(network, t, output_v):
  '''plot a raster plot and the readout cell timecourse'''
  fig = pyplot.figure()
  gs = gridspec.GridSpec(3, 1)
  ax = fig.add_subplot(gs[0:2, :])
  plot_raster(ax, network)
  ax.set_ylabel('cell id')
  ax.set_xticks([])
  ax.set_xlim([0, h.t])
  ax.set_ylim([-0.5, len(network.cells) - 0.5])
  ax = fig.add_subplot(gs[2, :])
  ax.plot(t, output_v)
  ax.set_xlim([0, h.t])
  ax.set_xlabel('t (ms)')
  ax.set_ylabel('output cell (mV)')
  pyplot.show()


if __name__ == '__main__':
  network = Network(
    num_cells=100,
    interval_low=10,
    interval_high=15,
    delay=4,
    weight=-0,
    tau=10)

  output_sec, output_conns = create_readout_cell(network, weight=5e-6)

  output_v = h.Vector()
  output_v.record(output_sec(0.5)._ref_v)
  t = h.Vector()
  t.record(h._ref_t)

  h.finitialize(-65)
  h.continuerun(1000)
  plot_raster_and_output_mv(network, t, output_v)

  '''
  can run other experiments after this is done using the network

  e.g.

  network.weight = -0.005
  h.finitialize(-65)
  h.continuerun(1000)
  plot_raster_and_output_mv(network, t, output_v)
  '''
```


Parallel Computing with MPI

Getting started

The first step is to get to the place where you can run the "hello world" level program test0.hoc by launching in a terminal window with

```
mpiexec -n 4 nrniv -mpi test0.hoc
```

and see the output

```
...
I am 1 of 4
I am 3 of 4
I am 0 of 4
I am 2 of 4
```

MSWIN

The NEURON setup installer has already put all the required software on your machine to use MPI. I.e. a subset of MPICH2 compiled under CYGWIN.

1) start an rxvt terminal window
2) start the mpd daemon

```
mpdtrace # is it already running
mpd&
mpdtrace # it will persist until you do an mpdallexit or close the terminal
```

3) launch the program (mpiexec command above) in the directory containing test0.hoc (or give a full path to test0.hoc)

Mac OS X and Linux

Unfortunately MPI can't be a part of the binary installation because I don't know if, which, or where MPI was installed on your machine. So you have to install MPI yourself, check that it works, and build NEURON from the sources with the configure option '--with-paranrn'. See the "installing and testing MPI" section of the Hines and Carnevale (2008) paper, "Translating network models to parallel hardware in NEURON", J. Neurosci. Meth. 169: 425-465. The paper is reprinted in your handout booklet. Or see the ModelDB entry

Going further

The ring model from the above ModelDB entry is a good next step. See also the documentation for the ParallelContext class, especialy the subset of methods gathered under the ParallelNetwork heading. A large portion of the ParallelNetManager wrapper is better off done directly from the underlying ParallelContext though it can be mined for interesting pieces. A good place to find the most recent idioms is the NEURON implementation of the Vogels and Abbott model found in the Brette et al. ModelDB entry. However, to run in parallel, the NetCon delay between cells needs to be set greater than zero.

NEURON hands-on course

Listing of test0.hoc

```
objref pc
pc = new ParallelContext()
{
printf("I am %d of %d\n", pc.id, pc.nhost)
}
{pc.done()}
quit()
```


Using the Neuroscience Gateway Portal

Exercise

You can do this exercise on your own, or you might prefer to form a team of 2-4 people to work on it.

On your own computer:

Whether you're working on your own or in a team, everyone should start by downloading the zip file for the Jones et al. 2009 model from ModelDB. Also get a copy of their paper and examine it to discover what is being modeled and what results are produced.

Expand the zip file and examine its contents to figure out how to run a simulation (look for a "readme" file).

Each member of the team shoudl compile the mod files and run a simulation on their own PC with 1 processor.

What output files were generated?

How long did your run take to complete?

What file contains this information?

Now run simulations with 2 processors and 4 processors and record the run times.

For each number of hosts, plot the run time vs. log2(number of hosts). Compare your results with those of your teammates. Do these graphs demonstrate strong scaling or weak scaling?

Using NSG:

Each member of the team should work through the tutorial at https://kenneth59715.github.io/NSGNEURON/

In that tutorial you will

1. Upload the model's zip file to NSG.
2. Set up a task that runs a single simulation on 1, 2, 4, or 8 processors on Comet using NEURON 7.5 (each member of the team should choose a different number of processors).
3. Execute the simulation.
4. Download output.tar.gz, and expand and explore the contents of this file.

Analyze the results

How long did your run take to complete on Comet?

Share your run times on Comet with each other and with at least one other team.

For each number of hosts, plot the average run time vs. log2 number of hosts. Does this plot demonstrate strong scaling or weak scaling? How do these run times compare to the run times you and your team members got with your own hardware?

Examine the other output files. What kind of data is contained in `Mu_output.dat`? How about `out.dat`? Examine the model's source code to determine if your guesses are correct.

Make a graph that plots the results in Mu_output.dat, and another graph that plots the results in out.dat.

Idioms

Iterating over all segments

```
forall for (x) print secname(), x
forall for (x,0) print secname(), x // leave out 0 and 1, i.e. the 0 area nodes
```

Data Structures

```
begintemplate Temp
        public str, obj, x
        strdef str
        objref obj
        x = 0
endtemplate Temp

objref temp
temp = new Temp()
temp.x = 2
temp.str = "hello"
temp.obj = new SectionList()

print temp, temp.x, temp.str, temp.obj
```

Adding a Graph so it works with the Standard Run Library

```
begintemplate P
        public flush, plot, begin, view_count
        objref this
        proc flush() { print this, "flush", t }
        proc plot() { print this, "plot", t }
        proc begin() { print this, "begin", t }
        func view_count() { print this, "view_count"   return 1}
endtemplate P

flush_list.append(new P()) // call flush every step
graphList[0].append(new P()) // call plot and flush every step
```

Arrays of strings

```
/*
// The standard run library includes this template which defines a String class:
begintemplate String
        public s
        strdef s
        proc init() {
                if (numarg() == 1) {
                        s = $s1
                }
        }
endtemplate String
*/

objref sobj[5]
for i=0, 4 sobj[i] = new String()
for i=0, 4 sprint(sobj[i].s, "Number %d", i)
for i=0, 4 print sobj[i].s
```


Neat stuff to do after picking a Graph line into the vector clipboard

You can then apply the Vector methods, as in these examples.

`hoc_obj_.c.printf`
prints a copy of the vector

`hoc_obj_.c.deriv(1,1).printf`
prints the euler derivative

`hoc_obj_.c.deriv(1,1).indvwhere(">", 1e-5).printf`
prints the indices

```
hoc_obj_.ind(hoc_obj_.c.deriv(1,1).indvwhere(">",
1e-5).add(1)).printf
```
prints the peak values

```
hoc_obj_[1].ind(hoc_obj_.c.deriv(1,1).indvwhere(">",
1e-5).add(1)).deriv(1,1).printf
```
prints the time intervals between synaptic discontinuities

The following aren't hoc idioms, but they can be helpful.

NEURON's interpreter window

EMACS

These EMACS commands work for command line editing from the console (MSWin: NEURON's interpreter window).
^P previous line (up arrow may also work; this crude history function can be applied repeatedly to scroll through prior commands)
^A front of line
^E end of line
^B backward character
^F forward character
Long command lines can be constructed by revision + accretion (recall a prior line, make changes and add new stuff to it).

Vector/matrix: reading data

Exercises

1) Run the plotdata.hoc file in the vec directory. This reads the contents of data4.dat and displays all the lines in a Graph. Look at the hoc file and the format of the data file. Notice that the first number is the number of rows of data (Strings like #9 are not numbers. Numbers are strings that start with a 0-9, ., or +-. Modify the plotdata.hoc file so it uses a File chooser to read the file.

The following hints may help with this exercise. For information about File.chooser, in the NEURON console, type

```
help File
```

or else navigate there manually with the web browser.
Is a file open for the File referenced by f? Type `f.isopen()`
Define the chooser to be a read type: `f.chooser("r")`
Pop up the chooser: `f.chooser()` : and select data4.dat
Is a file open?

2) Save these data using the Print&FileWindowManager/PrintFile/Ascii menu item into the file temp1.dat . Look at the format of the temp1.dat file. Fix the temp1.dat file so it can be read with plotdata.hoc.

3) The data are in a Matrix. Prove this to yourself by typing `m` and then `m.printf`. Save the matrix by opening the file with `f.wopen("temp2.dat")` and then `m.fprint(f)`. Look at the contents of the file and verify that it can be read with plotdata.hoc.

4) Pick one of the data lines into the clipboard and save it using the NEURONMainMenu/Vector/SaveToFile menu item as temp2.dat . Verify that temp2.dat can be read correctly with the NEURONMainMenu/Vector/RetrieveFromFile. Fix the temp2.dat file so it can be read with plotdata.hoc.

An observation

There are probably more data formats than there are programs that write data files. Data in ASCII can generally be read with a hoc program (see File.scanvar, File.gets, sscanf, Vector.x and Matrix.x). Some formats are very complicated and are in binary, e.g. PClamp binary data files. Cases like this can only be handled with model descriptions like clampex.mod. After the data are in a Matrix or Vector set, they are generally fairly easy to display with the GUI or manipulate with simple hoc programs.

Vector/matrix: subtracting linear response

Exercises

1) Run plotdata.hoc in the course/vec directory. This reads the contents of data4.dat and displays four curves: the voltage responses to hyperpolarizing current pulses of -9, -18, -27, and -36 nA. Scale these curves according to the magnitude of the current pulse to see if the response was linear.

Hints

A good way to test a Vector method is to pick a curve into the clipboard, apply the method, and either printf the result or plot it in a NEURONMainMenu/Display tool. For example, if we pick the 36nA curve and put it in a Vector/Display and then pick the 9nA curve, scale it with

```
hoc_obj_.mul(4)
```

and then put it in the tool, we suddenly realize that the curves need to be shifted to 0 before being scaled. One way to shift would be merely to select a point on the resting curve and subtract that value from the entire curve, ie

```
hoc_obj_.sub(hoc_obj_.x[10])
```

If the resting value is noisy, we could subtract the mean value over a short range as in

```
hoc_obj_.sub(hoc_obj_.mean(5,15))
```

To modify a column in a matrix, get it into a Vector with m.getcol(i) and put a Vector into the column with m.setcol(i, vector). i.e one can scale a column in one statement with

```
m.setcol(i, m.getcol(i).sub(m.getcol(i).mean(5,15)).div(9*i))
```

but long nested chains get easier to write than to read and it is generally better to do only one or two Vector operations per statement.

2) This has nothing to do with data but it will give you practice in Vector manipulations. Set up a simulation with several action potentials, e.g. with a long but low amplitude current pulse in an HH patch. Run it with the variable time step method. Now, plot log10(dt) as a function of t.

Hints

Use the clipboard. The t values are in hoc_obj_[1]. Calculate the dt values in hoc_obj_ (a synonym for hoc_obj_[0]) and use a Vector/Display to plot it.

Some pertinent Vector methods are

`vec.c`	return a clone (new identical copy) of the vector instance
`vec.deriv(1, 1)`	v.x[i+1]-v.x[i] -> v.x[i] , size is 1 less than before

`vec.resize(vec.size-1)`	reduce size of vector by 1
`vec.log10`	log10(v.x[i]) -> v.x[i] , log10(0) is an error
`vec.remove(index_vector)`	remove all elements at values given by index_vector
`index_vector = vec.where("==", 0)`	indices where v.x[i] == 0

If removing 0 elements seems too complicated, it is less elegant but just as effective to add 1e-9 to every element before taking the log.

Simulation control: a family of simulations

Modeling projects often involve executing a series of simulation runs, analyzing the output of each run, and summarizing the results. This exercise will have you examine how the location of an excitatory synapse affects the peak depolarization at the soma. In doing this, you will learn:

- how to set and determine the position of a point process under program control
- how to use the Vector class to collect and analyze simulation output
- more about managing models with the CellBuilder

Conceptual Model

The idea behind this exercise is a thought experiment that explores the relationship between synaptic location and the size of the EPSP observed at the soma. In this experiment you attach an excitatory synapse to the cell at various locations along a particular dendrite, activate the synapse, and record the peak amplitude observed at the soma. After collecting these data, you plot somatic EPSP amplitude vs. position of the synapse along the dendrite.

Simulation

Computational model

This exercise uses the ball and stick model you have already seen.

In the exercises/simulation_families directory, start NEURON by running init.py (python -i init.py). This particular init file

```
from neuron import h, gui

# load the ballstk cellbuilder
h.load_file("ballstk.ses")

# eventually becomes a custom GUI
h.load_file("rig.ses")
```

makes NEURON read the session file for a CellBuilder that contains the specification of the model. You will use this CellBuilder to adjust the model parameters.

> **Note:** Make sure the CellBuilder's Continuous Create button is checked. Otherwise the sections of the ball and stick model will not exist.

Managing the spatial grid with the CellBuilder

1. Geometry/Specify Strategy : select d_lambda for the all subset. Also make sure that no section has an overriding strategy.
2. toggle Specify Strategy *off*
3. make sure that d_lambda = 0.1 space constant at 100 Hz

How many segments were created?


```
for sec in h.allsec():
    print('%s nseg = %d' % (sec, sec.nseg))
```

Where are the nodes located?

```
for seg in dend:
    print('%g %g' % (seg.x, seg.x * dend.L))
```

If these locations aren't particularly "esthetic," you can assign nseg a new (larger) value manually (odd multiples of 5 are nice). You *could* do this with a Python statement like
`dend.nseg=25`
but this would be a potential cause of confusion, so you should specify nseg through the CellBuilder instead.

> Remember, you are using the CellBuilder with Continuous Create *on*. This means that, if you change the model specification in the CellBuilder, or even just toggle Continuous Create off and on, any changes to cell properties that you made outside of the CellBuilder (e.g. by executing Python calls at the >>> prompt) could be overridden.

Using the computational model

1. BUILD THE GUI

Set up a graphical interface that lets you apply an AlphaSynapse to the model (time constant 1.0 ms, peak conductance 0.005 umho) while observing Vm at the middle of the soma.

2. TEST THE MODEL

Put the synapse at the proximal end of the dendrite, turn on Keep Lines in the voltage graph, and run a simulation. Then reduce the peak synaptic conductance to 0 and run another. Use View = plot to get a better look at somatic Vm.

What's wrong with these responses? (hint: increase Tstop to 50 ms and run another simulation)
Change dendritic e_pas to -65 mV (use the CellBuilder's Biophysics page!) and try another run. Does this help? Why?

3. INITIAL EXPLORATION OF THE MODEL

Place the synapse at several different positions along the dendrite. Find and plot the peak amplitude of the somatic EPSP vs. synaptic location.

You will need a procedure that moves the synapse to a specified location. I have provided putsyn.py, which contains a function (`putsyn()`) that takes a single numeric argument in the range [0, 1] (i.e. the desired synaptic location).

`putsyn()` does these things:

1. Verifies that the requested location is actually in the range [0, 1].
2. Places the synapse in the section (uses the Point Process .loc() function).
3. Since point processes are always placed at the nearest node, and nodes are located at 0, 1, and the center of each segment, `putsyn()` must determine the actual location of the synapse (uses .get_segment().x). This is assigned to a global variable called `h.synloc`.
4. Executes the statement `h.run()` (equivalent to clicking the Init & Run button).

Load putsyn (`from putsyn import putsyn`), and then invoke putsyn() with a couple of different arguments to see what happens. Use the voltage axis graph's crosshairs to find the peak amplitude of the epsp at the soma.

You might also want to append the statement `from putsyn import putsyn` to the end of init.py for future use.

4. SWITCHING TO PRODUCTION MODE

In principle, you could type putsyn() many times, with different numerical arguments, measure the epsp peaks manually, write down the synaptic location and the corresponding peak depolarization, and then plot the results by hand, but that would be a poor use of your time. It's much better to learn how to automate repetitive tasks.

Here's an outline of one approach to automating this particular modeling experiment:

For each node along dend
    move the synapse to that node
    run a simulation
    determine the most depolarized somatic Vm produced by a synapse at that location
    save this value and the synaptic location in a pair of vectors
Plot the vector of peak values vs. the vector of synaptic locations

Use a text editor to create a function called `profile` that implements this approach. Put it in a file called myprofile.py, and then use the command
`from myprofile import profile`
to make it available to your model.

You already have `putsyn()`, which takes care of the second and third items in this outline.
It may be helpful to know about :
the Vector class in general Vector record() Vector max() Vector append() Vector plot() the Graph class

Here is a skeleton of one possible implementation of such a function.

5. THINGS TO TRY

1. Compute the actual EPSP amplitudes by subtracting the -65 mV baseline from the soma responses and plot the results.

2. Plot peak EPSP amplitude as a function of anatomical distance along dend in microns.

2. What would happen if the somatic HH currents were blocked? Use the CellBuilder to reduce gnabar_hh and gkbar_hh to 0. Make sure to change el_hh to -65 mV before running a new series of simulations (why or why not? and what if you don't?).

Compare these results with what you saw when HH currents were not blocked. Do spike currents in the soma enhance all EPSPs, or does the nature of the effect depend on synaptic location?

```
""" putsyn.py for somatic epsp as a function of synaptic location

       last modified 7/25/2018 RAM

   based on putsyn.hoc

       last modified 7/15/99 NTC
"""
from neuron import h
h.load_file('stdrun.hoc')

# define a global variable for the synapse location, set it to fake value
h('synloc = -1')


def putsyn(x):
        if not (0 <= x <= 1):
                raise Exception('location not in the range [0, 1]')

        # move the 0th AlphaSynapse to the segment that contains x
        h.AlphaSynapse[0].loc(x, sec=h.dend)
        synseg = h.AlphaSynapse[0].get_segment()
        # discover where it actually went (center of that segment)
        h.synloc = synseg.x
        h.run()
```


Here is a skeleton of one possible implementation of the final function.
Ordinary comments are indicated by #,
things that remain to be done are indicated
by remarks inside triple quote pairs.

In broad outline, the function **profile()** walks the synapse along the length of **dend**. At each node (the 0 and 1 ends plus the center of each segment), the time course of somatic Vm is computed and stored in the vector **vm**, which is then examined to find its maximum value. The synaptic location and the peak amplitude of the somatically observed epsp are then stored in the vectors **location** and **amplitude**, respectively. Finally, the graph **g** displays a plot of **amplitude** vs. **location**.

```
from neuron import h, gui
from putsyn import putsyn

g = h.Graph()       # for plot of amplitude vs. location

vm = h.Vector()     # to hold the time course of somatic Vm
                    # evoked by a synaptic input
""" use Vector class .record() to "attach" vm to soma(0.5).v """


def profile():
        global g

        # if you're familiar with matplotlib or other Python graphics libraries,
        # it may be more convenient to use Python lists and those libraries
        # but everything can be done in pure NEURON
        g = h.Graph()
        location = h.Vector()    # store locations along the dendrite
        amplitude = h.Vector()   # store peak amplitude at each location

    # .allseg() loops over all nodes, including 0 and 1
        for seg in dend.allseg():
                putsyn(x)
                """ at this point, vm should contain a record of soma(0.5).v """
                """ find maximum element in vm """
                """ append this to amplitude vector """
                """ append x to location vector """

        """ plot amplitude vs. location """
```

Model Control: Arbitrary forcing functions

It is often useful to make some model parameter follow a predetermined time course during a simulation. For example, you might want to drive a voltage or current clamp with a complex waveform that you have computed or measured experimentally, or have a synaptic conductance or equilibrium potential change in a particular manner. You *could* do this at the interpreter level by inserting the necessary hoc statements inside the main computational loop, but this is cumbersome and imposes extra interpreter overhead at each time step.

The record and play functions of the Vector class are a better alternative. These are accessible using hoc statements and also through the GUI.

To help you learn how to generate and use arbitrary forcing functions, in this exercise you will use the GUI to make a ramp clamp. You will

- set up and test a voltage clamp with a standard step command
- generate the desired waveform
- use this waveform as the command for the voltage clamp

But first, you need some membrane with HH channels--

Physical System

A patch of excitable membrane

Model

Hodgkin-Huxley g_{Na}, g_K, and g_{leak} in parallel with membrane capacitance

Simulation

Start NEURON with its standard GUI with /course/arbforc as the working directory. Then use Build / single compartment from the Main Menu. This creates a single section with a total surface area of 100 um^2 and nseg = 1. It also brings up a Distributed Mechanism Inserter (a small window with checkboxes) that you can use to specify which mechanisms are present.

> An aside: the total current in units of [nA] (nanoamperes) through a patch of membrane with area = 100 um^2 is numerically equal to the current density in units of [mA/cm^2]

The membrane of this single-compartment model has cm = 1 uf/cm^2, but it lacks ion channels. Use the inserter to endow it with ion channels (hh), and then set up instrumentation to experiment on it.

The tools you'll bring up with the NEURON Main Menu:

- RunControl
- Voltage axis graph for a plot of v vs. t

- a PointProcessManager configured as an IClamp to deliver a stimulus pulse (verify that the membrane is excitable)

Set up and test a voltage clamp

Turn the point process into a single electrode clamp (SEClamp) that has these parameters:
dur1 = 1 ms, amp1 = - 65mV
dur2 = 3 ms, amp2 = - 35mV
dur3 = 1 ms, amp3 = - 65mV

You need another graph to show clamp current vs. t
(Graph / Current axis, then use the graph's Plot what? to specify SEClamp[0].i)

Run a simulation to test the voltage clamp. If voltage control is visibly poor, divide rs by 10 and try again. Repeat until membrane potential is well controlled.

Generate the ramp waveform

You want a voltage ramp that starts at v_init mV when t = 0, and rises at a constant rate.

This could be done in hoc by creating a vector with the right number of elements (one for each time step in the simulation), and then assigning the desired value to each element. However, it is more convenient to use the Grapher (NEURON Main Menu / Graph / Grapher) Read about it and try this simple example:
Use Plot what?, and enter the expression `sin(t)` into the "Variable to graph" field.
Then L click on the Grapher's Plot button.

The Grapher can plot any function that you specify in hoc. Once the desired waveform is plotted, you can use Pick Vector to copy it to NEURON's Clipboard.

Let's make a ramp that starts at the holding potential v_init and depolarizes steadily at a rate of 1 mV/ms for 50 ms. To do this, set the following parameters in the Grapher :

PARAMETER	VALUE	COMMENT
Indep Begin	0	t at start of ramp
Indep End	50	t at end of ramp
Steps	50/dt	NEURON's GUI defined dt for us
Independent Var	t	
X-expr	t	
Generator		leave blank

Next tell the Grapher what to plot (use Plot what?, and enter the expression `v_init+t*1` into the "Variable to graph" field).

Plot and examine the resulting waveform (may need to use View=plot).

When you are satisfied with the result, use Pick Vector to copy the ramp into NEURON's Clipboard.

Use the waveform as the command for the voltage clamp

The play function of the Vector class can be exercised in hoc, but it is more convenient to use the GUI's Vector Play tool
(NEURON Main Menu / Vector / Play).

This tool has a "Specify" button that brings up a menu with several items. For this exercise, the most important are "Variable Name" and "Vector from Clipboard". "Vector from Clipboard" will deposit the ramp waveform into the Vector Play tool. Use the tool's View=plot to verify that this has happened. Then use "Variable Name" to specify SEClamp[0].amp1

You are almost ready to test the ramp clamp -- but first you should increase SEClamp[0].dur1 to something BIG. Anything >=50 ms will do, and if you plan to play with other waveforms, you might as well make it much larger, say 1000 ms (why is this necessary?).

Finally, L click on Vector Play's "Connected" button, and then run a simulation.

When everything is working, save the configured SEClamp, the graph of clamp current vs. t, the Vector Play tool, and the Grapher to a new session file (call it rampclamp.ses).

Exercises

1. Try changing dt.
Turning on Keep Lines in the voltage and current plots will let you compare the new and old results side-by-side.
In the RunControl window, cut dt in half.
What happens to the time course of Vm and clamp current?
What do you think would happen if you increased dt to 0.05 ms?

Why does this occur?

Don't forget to restore dt to 0.025 ms when you're done.

2. Try changing the ramp's dv/dt. In the Grapher's graph window, invoke Change Text and edit the expression, then Plot the new waveform and copy it to the Vector Play tool. Try 2 mV/ms, 3 mV/ms, whatever -- you can't fry this cell!

3. How do you get a plot of clamp current vs. Vm?
Answer: use a Phase Plane graph (Graph / Phase Plane).
For the x axis expression enter v (OK for this simple one-compartment model, but SEClamp[0].amp1 would be a more flexible choice for a cell with many sections if you want to move the clamp away from the default section).

4. Try a different driving function, e.g. `v_init+sin(2*PI*t/10)*5`

An interesting variation: reconfigure the PointProcessManager as an IClamp and drive the amplitude of the current it delivers with a sinusoid or some other waveform.

5. Try the ramp clamp with a different cell, e.g. the ball and stick model used in previous exercises.
Use NEURON to execute the init.hoc file in the /course/arbforc subdirectory.
Incidentally, I have included the statement

```
dend nseg = 27
```

in the init.hoc file for improved spatial accuracy.
After playing with the cell under current clamp, close the IClamp PointProcessManager and

retrieve the session file rampclamp.ses (a good copy of this file already exists under the name rampclamp.se).

> An important aside: when rampclamp.ses was saved, its SEC referred to a section with the name "soma". Therefore it will work with any cell that has a section called "soma". If you try to use it with a cell that does NOT have such a section, it won't work and you'll get an error message.

First try a 1 mV/ms ramp applied to the soma, then try a 2 mV/ms ramp.

Can you improve control of Vm by cutting the SEClamp's access resistance (rs)?

See what happens when you move the SEClamp out onto the dendrite. It might be instructive to bring up a space plot that monitors Vm along the entire length of the model; in this case, you may also want to speed things up by reducing Points plotted/ms from 40 to 20.

Hints & tips

0. The Grapher can display more than one function at a time--just invoke Plot what? more than once. Avoid visual confusion by making the traces have different colors (Color/Brush).

1. The forcing function can be far more complicated than a one-liner. Create a hoc file that contains your own function, xopen() the file, and then plot it. Simple scaling and baseline shifts can be accomplished in the Grapher itself (Change Text).
Example:

```
func whatnot() { local abc
  if ($1 < 10) abc = sin(2*PI*$1/10)
  if ($1 >= 10 && $1 < 20) abc = (sin(4*PI*$1/10) > 0)
  if ($1 >= 20) abc = exp(($1-20)/30)*sin(2*PI*(($1-20)/10)^2)
  return abc
}
```

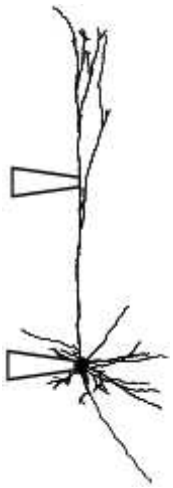
2. Alternatively, you could create a vector graphically with the MakeCurve tool (NEURON Main Menu / Vector / Draw). This is best suited to waveforms that consist of a sequence of rectangular steps (and linear ramps, if you activate the Vector Play tool's "piecewise continuous" option by clicking on Specify / Piecewise continuous).

3. Experimental data (e.g recordings of membrane potential or clamp current) can also be used as forcing functions. File i/o can be done through the GUI with the Clipboard, or under program control with these Vector class methods: fread, fwrite, vread, vwrite, scanf, printf, scantil.

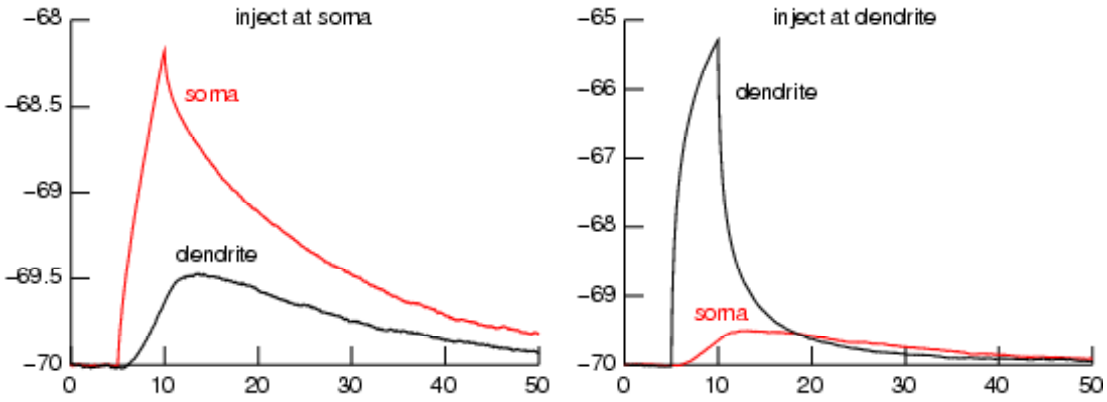
Optimizing a Model

Physical System

Two patch electrodes were attached to a pyramidal cell in layer 5 of the neocortex. One electrode was attached to the soma, and the other was attached to a dendrite in the apical tree. The resting potential of the cell was -70 mV, and its I-V relationship was linear from -90 to -50 mV.



This figure shows the response of the cell to injected current pulses (0.1 nA x 5 ms).



Two experimental protocols were used.
Protocol 1 (left): somatic current injection.
Protocol 2 (right): dendritic current injection.
Each trace is the average of four recordings.

At the end of the experiment, the cell was fixed and stained, and detailed anatomical measurements were made.

Experimental evidence (Stuart and Spruston, 1998) suggests that membrane conductance in the apical dendrites of pyramidal neurons increases with distance from the soma in a way that can be described by the sigmoidal function
g = A0 + A/(1 + exp(k*(d - p)))

where p is the distance from the soma, A0 is the conductance at the soma, A0 + A is the maximum conductance as p becomes very large, d is the distance at which g is halfway between A0 and A0 + A, and k governs the maximum slope of the sigmoid.

We will use the same k as Stuart and Spruston did (k = 1/(50 um), but the other biophysical parameters (not just A0, A, d, but also Ra, cm, and membrane conductance in the apical dendrites and elsewhere) are to be estimated from the experimental data.

The exercise

The task is to adjust the biophysical parameters of a model cell so that its response to injected current matches the experimental data. In this exercise you will learn how to use the Multiple Run Fitter (MRF), NEURON's GUI tool for automating optimization.

Here's an outline of the steps that are involved in accomplishing this task.

1. Create a specification of the model cell based on its anatomy and an intial guess at its biophysical properties, including the spatial distribution of ion channels.
2. Set up a "virtual experimental rig" to use with this cell for the purpose of recreating the experimental protocols.
3. Create and configure an MRF that will simulate the protocols and adjust the model parameters to minimize the differences between the simulation and experimental results.
4. Use the MRF.

Time is short and we need to focus on the MRF itself, so instead of building everything from scratch, we'll start with some preconfigured building blocks. These are:

cell.hoc
: A model specification that combines cellular anatomy with a "reasonable first guess" at its biophysical properties (what channels are present and how they are distributed in space). A first draft was created with the CellBuilder (very handy for setting up the g_pas gradient in the apical dendrites!) and exported to a hoc file. The hoc file was revised so that we can use its built-in procedures for assigning values to Ra, cm, g_pas, A0, A, and d.

  In the original version of cell.hoc, these parameters were assigned fixed numerical values that were buried inside procedures, and some of them were "local" variables that would be inaccessible to the MRF. The revision involved adding "surrogate parameters" called Ra_, cm_, g_pas_, A0_, A_, and d_ that the MRF can access.

  Here's how the cell model was set up, and a description of the specific changes to cell.hoc (for leisure reading after you have finished the rest of this exercise).

rig.ses
: Creates a "virtual experimental rig" for replicating protocols 1 and 2: a RunControl, a voltage axis graph, and a pair of PointProcessManagers configured as IClamps. IClamp[0] is attached to soma(0.5), and IClamp[1] to dendrite_1[9](0.5).

init_opt.hoc

Does the following:

- Loads nrngui.hoc, cell.hoc, and rig.ses. This creates the model cell and the virtual experimental rig.
- Defines starting values for the parameters that the MRF will adjust.
- Defines set_biophys(), which uses procs in cell.hoc and the surrogate variables (Ra_ etc.--see description of cell.hoc above) to assign the model's biophysical parameters.
- Uses an FInitializeHandler to ensure that set_biophys() is called before each simulation run.

Enough words already--let's get going!

Go to course/optimize and use NEURON to run init_opt.hoc

Check the parameters of the two IClamps.
See what happens when you run simulations with both IClamps delivering 0.1 nA, and after changing either IClamp's amp to 0.

This is an outline of how to proceed from here.

1. Configure a MultipleRunFitter to do a "run fitness" optimization.
2. Load experimental data into the Run Fitness Generator.
3. Tell the MRF what parameters to adjust.
4. Perform the optimization.

Reference

Stuart, G. and Spruston, N. Determinants of voltage attenuation in neocortical pyramidal neuron dendrites. *J. Neurosci.* 18:3501-3510, 1998.

NEURON hands-on course

Listing of init_opt.hoc

```
load_file("nrngui.hoc")
load_file("cell.hoc")
load_file("rig.ses")  // instrumentation

// model cell and instrumentation exist
// everything below this point is related to optimization

// assign starting values to the parameters defined in cell.hoc
// that will be adjusted by the MRF
Ra_ = 100
cm_ = 1
g_pas_ = 1/10000
A0_ = 1/10000 // ~ g_pas at prox end of apical tree
A_ = 9*A0_ // difference between g_pas at distal end and prox end
           // if apical tree were infinitely long
d_ = 1000 // distance at which g_pas is halfway between A0_ and A_

proc set_biophys() {
  // the following procs are defined in cell.hoc
  biophys() // may change Ra and cm
  geom_nseg() // so must adjust spatial grid
  // spatial grid may have changed so must call biphys_inhomo()
  // instead of g_pas_apicals_x()
  biophys_inhomo() // set up gradient of g_pas in apicals
}

// make sure that set_biophys() is called before finitialize() is called
objref fih
fih = new FInitializeHandler(0, "set_biophys()")

// load_file("mrf.ses")
```


Overview

We will use an MRF to adjust the parameters of a model to try to get the best match to data obtained with two experimental protocols. To this end, we must set up two Generators--one for each protocol.

For each Generator, we must specify

- a list of "protocol constants" that describe the experimental conditions (think "independent variables")
- a list of the "observed variables" ("dependent variables")
- the experimental results

In the context of this exercise, the protocol constants are IClamp[0].amp and IClamp[1].amp, the observed variables are soma.v(0.5) and dendrite_1[9](0.5), and the experimental results are the recordings of these variables.

Let's start by setting up the Generator for protocol 1.

Configure an MRF to do a "run fitness" optimzation

In the NEURON Main Menu toolbar, click on
Tools / Fitting / Multiple Run Fitter

Release the mouse button and an MRF appears. Drag it to a convenient location on your screen.

We need a tool to perform a "run fitness" optimization.

Create a Run Fitness Generator by clicking on the MRF's
Generators / Add Fitness Generator / Add Run Fitness

Release the mouse button, and the right panel of the MRF shows an item called "Unnamed single run protocol".

Give the Run Fitness Generator a descriptive name.

This is the Run Fitness Generator for protocol 1, in which current is injected into the soma, so change its name to "iclamp soma".

1. Click on
   Generators / Change Name
   "Change" should appear to right of the Generators button.
2. In the MRF's right panel, double click on "Unnamed single run protocol"
3. Type "iclamp" in the dialog box's edit field, then click its Accept button.

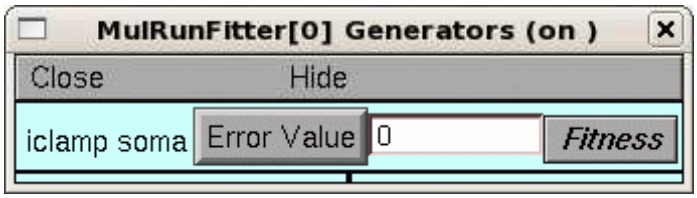
The right panel of the MRF will show the Generator's new name.

We need to see this Generator.

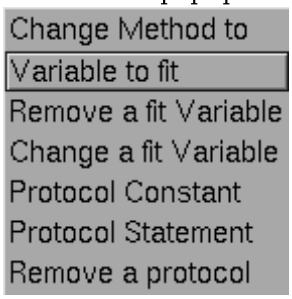
We have to see the Generator before we can get our experimental data into it.

1. Click on Generators / Display
   Now "Display" appears to the right of the Generators button.

2. Double click on "iclamp soma", and up pops up a tiny window titled "MulRunFitter[0] Generators".



3. Tell it the name of the dependent variable.
   Click on the iclamp soma Generator's Fitness button, and select the item "Variable to fit" from the popup menu.



   This brings up a "variable name browser" that looks and works just like a graph's "Plot what?" tool.

4. Click inside the edit field of the variable name browser and type
   ```
   soma.v(0.5)
   ```
   Then click on its Accept button.

The variable name browser will go away, but our Generator looks unchanged. We need to make the MRF redraw it.

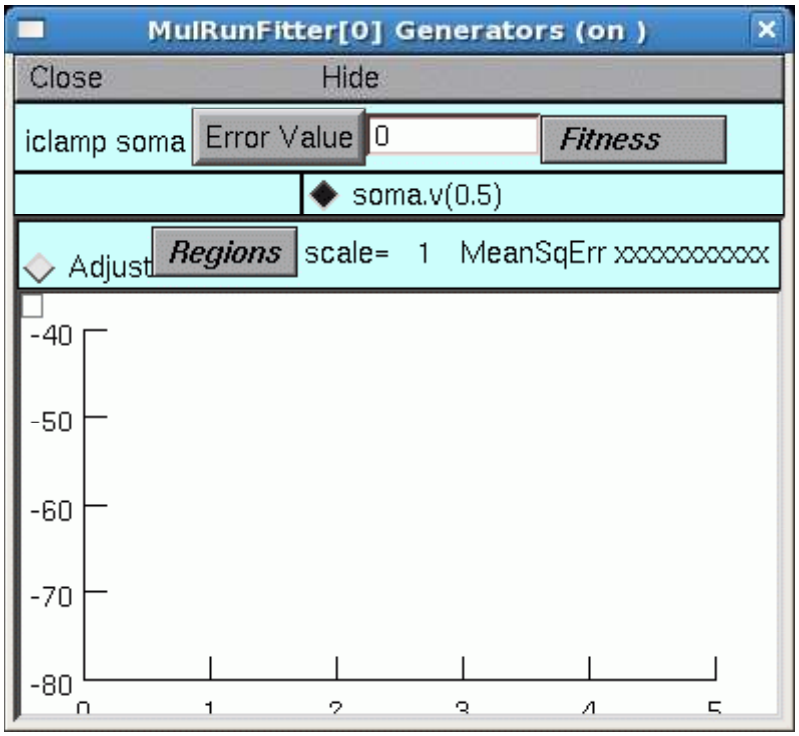
But first, save the MRF to a session file! I called mine mrf.ses

Redrawing the iclamp soma Generator

Click on the iclamp soma Generator's "Close" button (**NOT** the MRF's Close button!). To see the Generator again, make sure the MRF is in "Display" mode, then click on "iclamp soma" in its right panel.

> If you made a mistake and clicked on the MRF's Close button, both the MRF and the iclamp soma Generator went away. But since you saved the MRF to a session file, it's easy to restore--just xopen mrf.ses

Here's what the redrawn Generator looks like. Notice that the blue area above the graph has three rows. The top row shows the name of the protocol on the left. The middle row shows the name of the variable to fit on the right.



Next to do: load data into this Generator.

[Outline | Next]

NEURON hands-on course

Loading data into the Run Fitness Generator

This actually involves two related tasks: loading data into the Generator, and testing the Generator.

There are four data files. Each one has a name that starts with the letter *n* and ends with *.dat* The first part of the file name tells where current was injected, and the second part tells where voltage was recorded. The file we want now is called
nisoma_vsoma.dat
because it shows how current injected at the soma affected membrane potential at the soma.

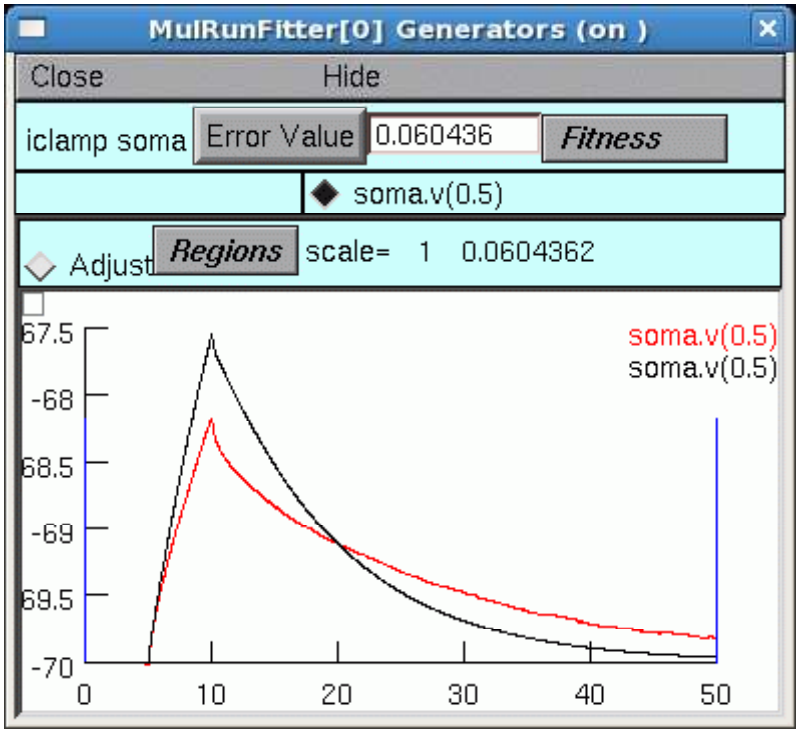
Read nisoma_vsoma.dat into NEURON's clipboard, then paste it into the iclamp soma Generator.

1. NEURON Main Menu / Vector / Retrieve from File
2. Navigate the directory tree and choose nisoma_vsoma.dat
3. In the iclamp soma Generator, click on Regions / Data from Clipboard

The Generator's graph area should now contain a red trace that shows the time course of membrane potential at the soma elicited by injecting a 0.1 nA x 5 ms current pulse at the soma.

Testing the Generator

Time to test the iclamp soma Generator. Click on its Error Value button. This should launch a simulation, producing a black trace that shows the trajectory of the simulated soma.v(0.5), and reporting an error value (see below).



Adding a second variable to fit

This protocol also produced a recording of dendritic membrane potential, so let's add that as the second variable to fit.

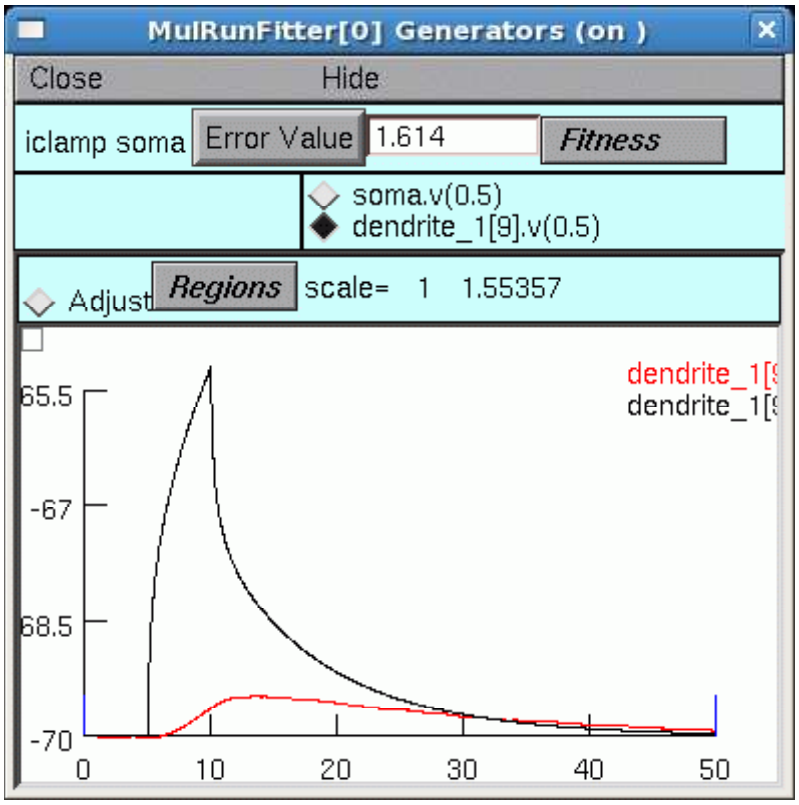
You already know how to do this--click on Fitness / Variable to fit
The name of the variable to add is dendrite_1[9].v(0.5)

If you don't see this new variable in the top panel of the Generator, maybe it's hiding behind something. Grab the bottom margin of the Generator's window and drag down a bit, and you should see a new radio button with "dendrite_1[9].v(0.5)" right next to it.

Having added a new variable, we must also add the corresponding experimental data.

Click on the dendrite_1[9].v(0.5) radio button and the Generator shows an empty graph.

Read nisoma_vdend.dat into NEURON's clipboard, then paste it into the Generator.
Now click on Error Value again and the Generator should look like this:



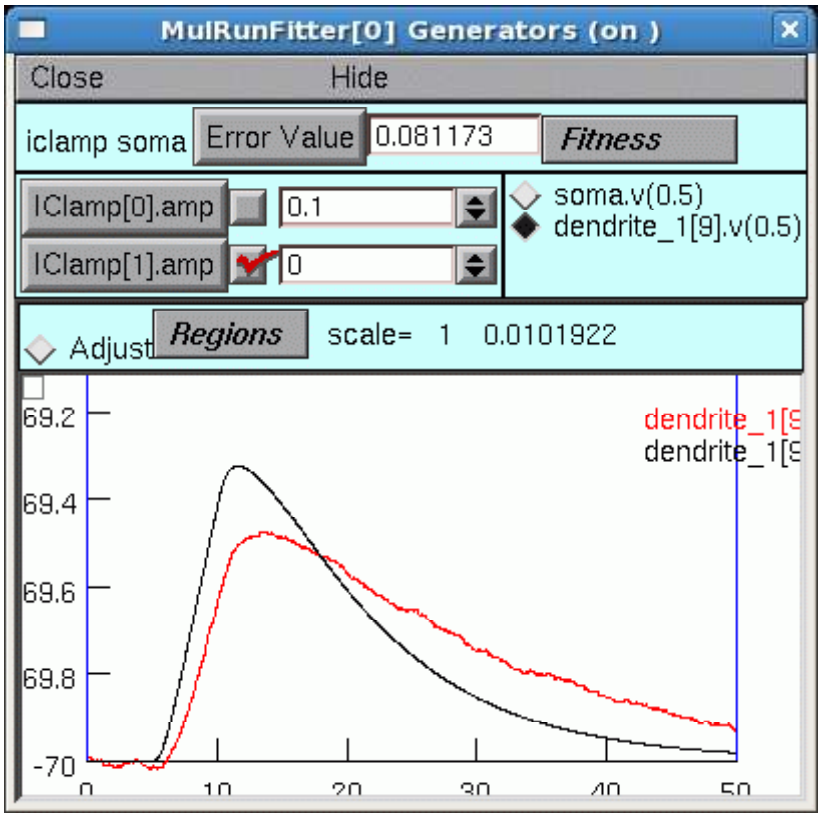
The experimental data (red) look OK, but the simulation looks like current is being injected into the dendrite.

And it is. We forgot about using protocol parameters to tell IClamp[1] that its amp should be 0 for the "iclamp soma" protocol.

Easy to fix. Click on Fitness / Protocol Constant, then use the "variable name browser" to add IClamp[0].amp.
Do the same for IClamp[1].amp.

Close the Generator and open it again, and notice that both IClamp amp parameters are 0.1. Enter 0 in the numeric field next to the IClamp[1].amp button, then save the MRF to a session file.

Finally click on the Error Value button and compare the experimental and simulated membrane potentials. Here's what the voltage in the dendrite looks like.



Use the radio buttons to switch back and forth between the graphs of soma.v(0.5) and dendrite_1[9].v(0.5).

Hint: to rescale the vertical axes, use the graphs' "Set View"

Next: specify the model parameters that are to be adjusted.

[Outline | Previous | Next]

NEURON hands-on course

Specifying the parameters that are to be optimized

Let's review the parameters that we want the MRF to adjust.

Parameter	Description
Ra	cytoplasmic resistivity
cm	specific membrane capacitance
g_pas	specific membrane conductance everywhere except apical dendrites
A0	minimum value of the sigmoidal function of distance that governs specific membrane conductance in the apical dendrites
A	A + A0 is the maximum value of the sigmoidal function
d	distance at which specific conductance in the apical dendrites is halfway between A0 and A + A0

But we're not going to have the MRF access these directly. Instead, we want to tell the MRF to control the surrogate variables Ra_, cm_, g_pas_, A0_, A_, and d_.

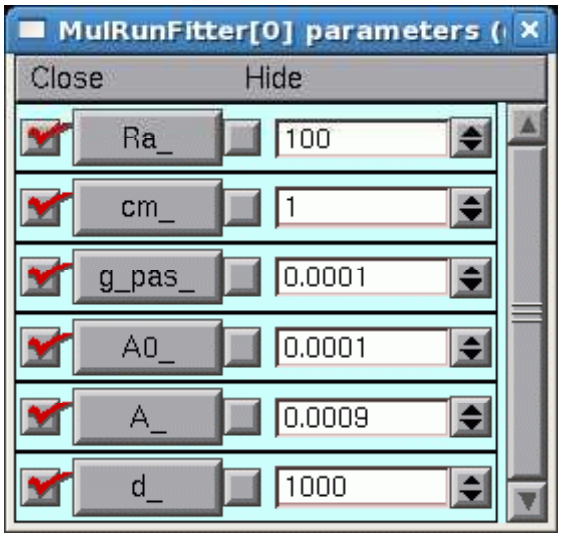
1. Click on the MRF's Parameters / Add Parameter
   This brings up a variable name browser (where have we seen that before?).
2. Click in the edit field of the variable name browser, type Ra_, then click on the Accept button.
3. Do the same for cm_, g_pas_, A0_, A_, and d_.

The surrogate variables will appear in the left panel of the MRF.

Save the MRF to a session file!

Viewing (and changing) parameter values

Click on the MRF's
Parameters / Parameter Panel



Change cm_ and see what happens. In the Parameter panel, increase cm_ to 2, then click on Error Value in the iclamp soma Generator. To restore cm_ back to its original value, click on the checkbox to the right of the cm_ button.

The checkboxes to the left of the parameter buttons specify which parameters the MRF is allowed to adjust. A check mark means the MRF can alter that parameter's value.

At this point we have set up the iclamp soma Generator, which emulates protocol 1, and told the MRF what parameters to optimize.

The next step is to try optimizing the model with this Generator.

[Outline | Previous | Next]

Optimizing the model with protocol 1

Test the MRF

First, test the MRF by clicking on its Error Value button.

Nothing happens--the number in the field next to the Error Value button is still 0.

We have to tell the MRF to use our Generator. Look at "iclamp soma" in the right hand panel of the MRF. See the little - (minus) sign? That means we haven't told the MRF to use the iclamp soma Generator.

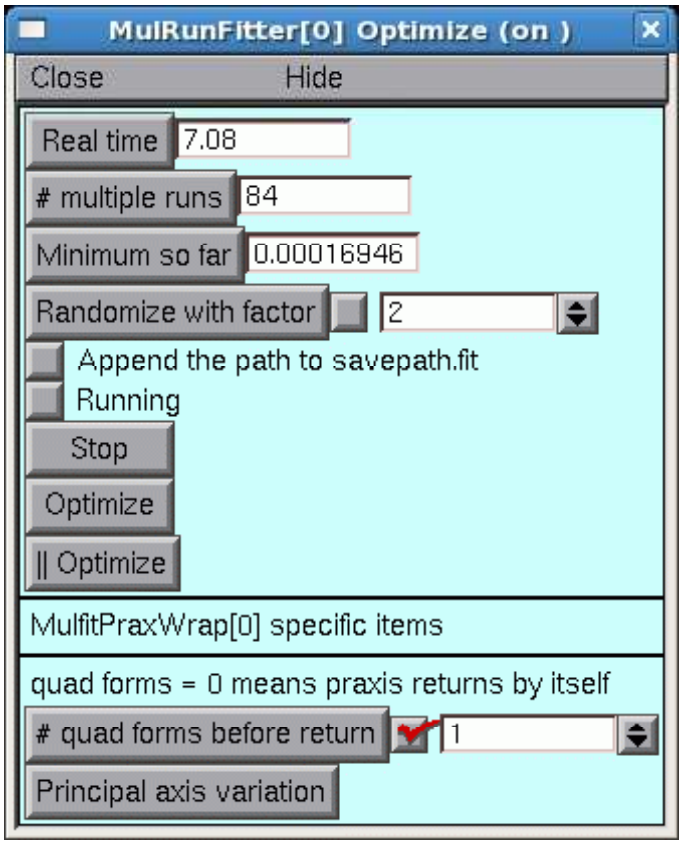
To fix this, in the MRF click on
  Generators / Use Generator
and note the appearance of "Toggle" next to the Generators button.
Double click on "iclamp soma" in the right panel of the MRF, and the - changes to a + (plus).

Now when we click on the MRF's Error Value button, the iclamp soma Generator will run a simulation and contribute to the total error that appears in MRF's error value field.

Choose and use an optimization algorithm

In the MRF click on
  Parameters / Select Optimizer / Praxis

This brings up a MulRunFitter Optimize panel, which we'll call the "Optimize panel". Change the "# quad forms before return" (numeric field near the bottom of the Optimize panel) from 0 to 1.



Now click on the Optimize button in this panel.
When the MRF stops, note the error value, then click on Optimize again.
And again.

Does it seem to be stuck?
Watch the values in the parameters panel--do any of them occasionally go negative?

Try constraining the parameters

It would be meaningless for any of the actual biophysical parameters (Ra, cm, g_pas, A0, and A) to become negative. And a negative value for d (distance at which membrane conductance is halfway between A0 and A) would also make no sense.

So all of the parameters are positive definite. To apply this constraint, bring up the MRF's Domain panel by clicking on its
  Parameters / Domain Panel

In the MulRunFitter Domain panel click on
  group attributes / positive definite limits

Now do a few more optimization runs.
The error decreases very gradually, and NEURON's interpreter prints a lot of complaints about parameters trying to go negative.

What else can we try?

The PRAXIS optimizer often benefits from logarithmic scaling of parameters. This seems to be most helpful when two or more parameters are very different in size, i.e. when they differ by orders of magnitude. Which is the case in this problem.

To apply logarithmic scaling to all the parameters, in the MulRunFitter Domain panel click on
group attributes / use log scale

Click on Optimize once more . . . much nicer!

> Range constraints and log vs. linear scaling can also be set for individual parameters. Just double click on a parameter in the Domain panel, then change the contents of the edit field in the window that pops up--see the "DomainPanel" discussion in the Programmers' Reference entry about the MulRunFitter.
>
> For an expanded discussion of parameter constraints, see **D. Constraining parameters** at
> http://www.neuron.yale.edu/neuron/docs/optimiz/func/params.html

More things to try

Add a Generator for protocol 2

Set up another Generator that uses the data obtained by injecting current into the dendrite. You can save yourself some effort by cloning the iclamp soma Generator, and then revising the clone.

1. In the MRF, click on
   Generators / Clone
   then double click on "iclamp soma" in the MRF's right panel. The name of the new Generator will have a - sign in front of it.
2. Change the name of the new Generator to "iclamp dend".
3. Display the new Generator. Notice that it has controls for specifying the protocol constants, and radio buttons for viewing the graphs that show soma.v(0.5) and dendrite_1[9].v(0.5).
4. Change the protocol constants so that they are appropriate for protocol 2.
5. Get protocol 2's experimental data into this Generator. These are in files called nidend_vsoma.dat and nidend_vdend.dat. Use
   NEURON Main Menu / Vector / Retrieve from File
   to read the somatic membrane potential recording into NEURON's clipboard, then make sure the Generator's soma.v(0.5) button has been selected, and click on
   Regions / Data from Clipboard
   Follow similar steps to retrieve the dendritic membrane potential recording and paste it into the Generator.

Use the Generators

See if using the iclamp dend Generator by itself does a better job of optimizing this model. Be sure to
Generators / Use Generator
and then "toggle" the Generators so you are using the iclamp dend Generator, and not the iclamp soma Generator. Also, for a fair test, before starting to optimize be sure to restore the parameters to their original values (Ra 100, cm 1, g_pas = A0 = 0.0001, A_ 0.0009, d_ 1000).

Finally, try using both Generators together and see if you get a better result, or at least faster convergence.

Hints:

1. If you think the optimizer may have fallen into a local minimum, or is in a parameter region where the error surface is very shallow, try randomizing the parameters. In the MRF Optimize panel, click on
   Randomize with factor
   (a factor of 2 is generally sufficient) once or twice, then run another series of optimization simulations and see how soon the error falls below a predetermined level, and what the new parameter values are.
2. You might find it interesting, and maybe even useful, to capture a record of parameter values and associated errors. To turn on "path logging", click on "Append the path to savepath.fit" in the MRF Optimize panel.

[Outline | Previous]

NEURON hands-on course

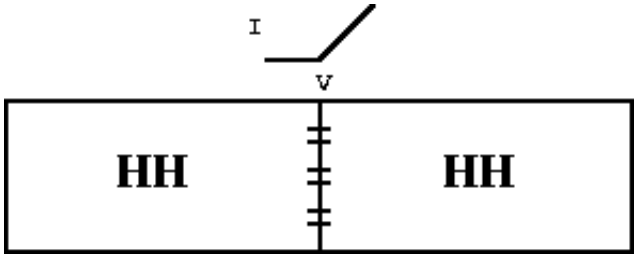
Rectifying Gap Junction

Physical System

Not in this universe.

Model

Rectifying gap junction between two Hodgkin-Huxley axons. When the internal voltage of the left side is greater than the internal voltage of the right side, current can pass. Otherwise the conductance is 0.



Simulation

This exercise shows how an understanding of the underlying NEURON representation of neuron cable sections in terms of equivalent circuits can become the basis for hitherto impossible simulations.

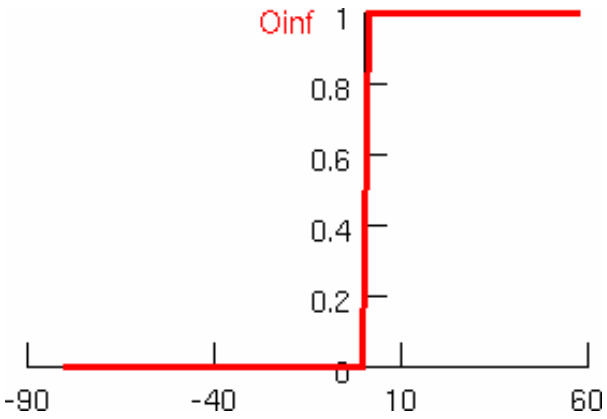
Representation

A Model description for a rectifying gap junction is rgap.mod. This membrane mechanism must somehow be wrenched out of its normal context of a channel between the inside and outside of a point on a section so that it can represent a channel between the insides of two sections.

However, I have a problem. I want you to be able to run the completed example from this page but this disallows the running of "special" versions of NEURON on unix machines. (There would be no problem under mswin since the downloaded hoc file that is to be executed could change to the proper directory "$(NEURONHOME)/../course/lincir2" and dynamically load the nrnmech.dll file).

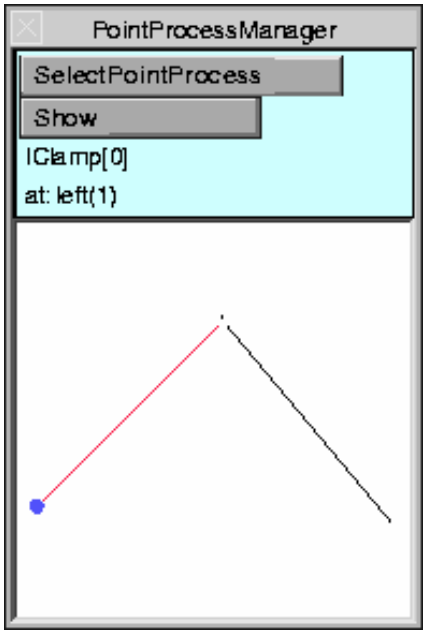
So, to allow all machines to run this example from the web, I am using a Channel Builder to specify the properties of the gap junction channels in terms of a simple HH-style mechanism with a single gating variable: g = gmax * O, where O' = (Oinf - O)/Otau

- Oinf = 1/(1+exp(-100*v)) so the steady state gap conductance is almost a step function at v = 0



- Otau = 0.01 ms, which is "instantaneous" compared to membrane time constant

We use three cellbuilders to create the two HH axons and a gap section. By specifying the orientation of each section and moving the 0 location of the root, we can arrange the sections so that combined appearance will turn out to look like



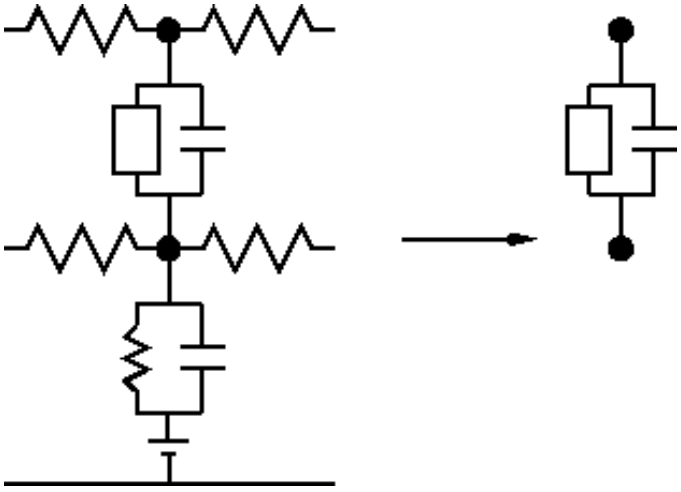
which nicely indicates the intended relation between the structures and makes it convenient to select the site of stimulus current injection as well as points for a time plot and two separate paths for a space plot.

Happily, the use of the "Reposition" tool in the "Topology" panel of a "Continuously Create"ing CellBuilder constantly updates the location of the instantiated sections and this is immediately re-plotted in a shape scene.

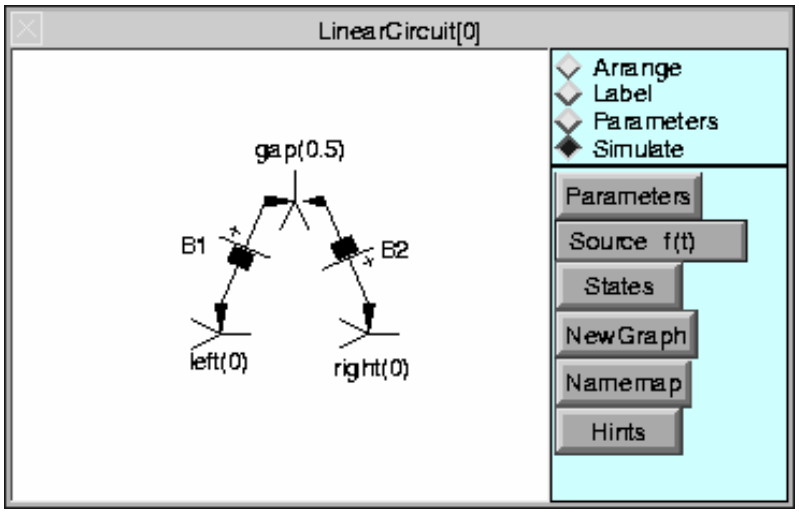
I'm using 1000um x 10um sections with hh channels and "d_lambda" segmentation for the "left" and "right" cables. For the gap section, I am using an area of 100 um2, cm as close to 0 as the cell builder allows (the default implicit method in NEURON allows cm=0 but that is another story), an rgap channel with maximum conductance of 1 mA/cm2 == 1 uS, ena=0 (ions mistakenly don't show up in the CellBuilder so that can only be done with a NEURONMainMenu/Tools/DistibutedMechanism/Viewers/ShapeName tool or with an interpreter

statement) and an extracellular mechanism.

The extracellular mechanism for the gap section is a key idea in this exercise since it gives us a non-trivial extracellular node (without an extracellular mechanism, the outside voltage next to the membrane is defined to be 0). This means that we can use available channel and point process mechanisms to create an arbitrary two terminal conductance component which, in conjunction with the LinearCircuit builder, can be attached between any two points in a network of cells. We defeat the normal usage of the extracellular mechanisms by setting xg to 0 so that there is no longer a default path from the extracellular node to ground. Thus the normal equivalent circuit of a single compartment section has been subverted to look like



Now the Linear Circuit Builder can be used to connect these pieces.



The batteries are necessary in order to introduce electrical current variables as first class states of the system. These current states allow consistency of an otherwise overdetermined system. That is, connecting two voltage nodes by a short circuit introduces the equation v1 = v2 and the only way to make this consistent with the two current balance equations for the nodes is to calculate the current flowing through the short circuit simultaneously with all the other membrane potentials. The batteries themselves have a default potential of 0 mV. This is a nice trick to use when one wishes to measure the current through a short circuit.

Completed example
See how an action potential initiated on the left propagates into the right section. However an action potential initiated on the right is blocked by the gap junction.

Clearly, the linear circuit builder would be inadequate to handle more than a few gap junctions. In hoc one would manage such simulations by writing a "GapJunction" template that manages a gap section with an rgap and extracellular mechanism along with a LinearMechanism instance to connect the gap junction to the desired locations. With that class as the basic object, it would then not be too much trouble to create a GUI tool (perhaps using a Shape scene) that managed interactive creation of the gap junctions.

```
: rectifying gap junction. Can only be used in conjunction with
: extracellular mechanism and the LinearMechanism class

NEURON {
      POINT_PROCESS RectifyingGapJunction
      RANGE g, i
      NONSPECIFIC_CURRENT i
}

PARAMETER {
      g = 0 (microsiemens)
}

ASSIGNED {
      v (millivolt)
      i (nanoamp)
}

BREAKPOINT {
      if (v > 0) {
            i = g*v
      }else{
            i = 0
      }
}
```


Electrotonic Analysis with NEURON

What's all this, then?

NEURON has a powerful, convenient, and flexible set of tools that facilitate analysis of electrotonic architecture. These tools compute the following :

input impedance Z_N	(local voltage change)/(local current injection)
transfer impedance Z_c	(local voltage change)/(remote current injection) equal to (remote voltage change)/(local current injection)
voltage transfer ratio k	(voltage downstream)/(voltage upstream) Identical to current and charge transfer ratio in the opposite direction.
voltage attenuation A	(voltage upstream)/(voltage downstream) Identical to current and charge attenuation in the opposite direction.
electrotonic distance L (see Note below)	natural log of A

These tools are based on the *electrotonic transformation* and their use led to the discovery of *passive normalization* (see citations).

> NOTE: The electrotonic distance computed by NEURON is defined by attenuation, but the classical definition is (anatomical distance/length constant). These two measures of electrotonic length are identical for an infinite cylindrical cable. However, the measure computed by NEURON always has a simple, direct relationship to attenuation, regardless of cellular anatomy, whereas the classical measure only has meaning in cells that meet several very specific constraints (such as the "3/2 power branching criterion"), and even then it does *not* have a simple relationship to attenuation. The new definition of electrotonic distance also preserves the direction-dependence of attenuation, which the classical definition obscures.

Exercises

Start `neurondemo` and select the Pyramidal cell model. Examine the side view of the anatomy of the cell in the Shape plot. Rotate, zoom in, and check it out from different vantage points (if you have any questions, here's how).

NEURON's tools for electrotonic analysis are gathered into four different "styles":

- Frequency
- Path
- log(A) vs. x
- Shape

They are accessible through NEURON Main Menu / Tools / Impedance. In this exercise you will start to learn how to use each of them. To save screen space, close a tool when you are done with it.

The Frequency tool

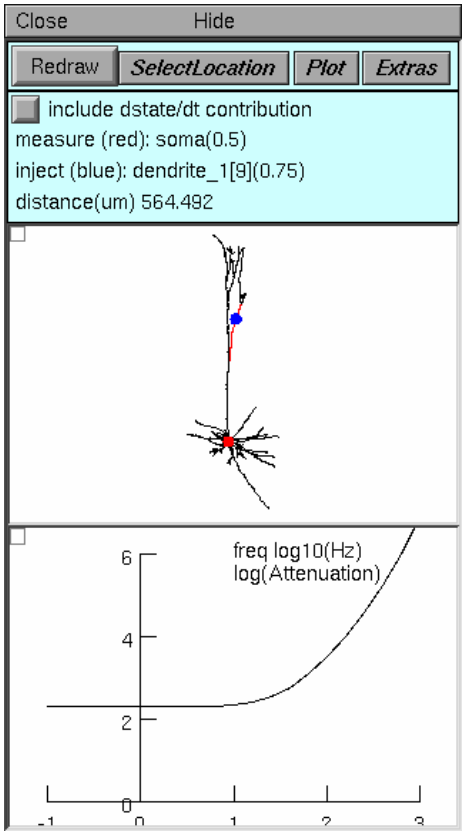
The Frequency tool can be used to study electrical coupling between any two points in a cell. Suppose an interesting signal is generated at some location in a neuron, e.g. by a synapse or by active conductances. An electrode is attached to the cell, but not necessarily where the signal is produced. The electrode may be used with a current clamp to inject current and record fluctuations in

membrane potential Vm, or with a voltage clamp that records the clamp current Ic that is needed to regulate Vm.

This experimental situation brings several questions to mind, such as

1. How accurately does observed Vm reflect what is actually happening at the site where the signal is generated?
2. How does the location of a synapse or active conductance change its effect on Vm elsewhere in the cell?
3. If the cell is voltage clamped, what fraction of the current generated by the signal source will the clamp capture?
4. If current is injected through the electrode to change local Vm, how does this affect Vm at the site of the "interesting signal"?
5. What does frequency do to the spread of signals between two points?

Bring up the Frequency tool: NEURON Main Menu / Tools / Impedance / Frequency



Top panel controls operation of the tool, and tells what is happening. Note particularly:

- Select Location enables the use of a mouse in the middle panel to change the sites of current injection and voltage measurement, or to swap these locations.
- Plot brings up a menu of items that can be plotted as functions of frequency
  - Log(Attenuation) plot natural log of voltage attenuation (shown here)
  - Zin and Ztransfer plot input or transfer impedance
  - Vmeasure/Vinject plots voltage transfer ratio, i.e. the reciprocal of voltage attenuation

Middle panel shows anatomy of the cell, indicating where current is injected (blue dot) and where Vm is measured (red dot). Zoom in or out as needed to see the whole cell.

Bottom panel in this example shows natural log of voltage attenuation from the injection site to the measurement site as a function of frequency over the range 0.1 to 1000 Hz. For most cells, attenuations at DC and 0.1 Hz are nearly identical.

Things to do:

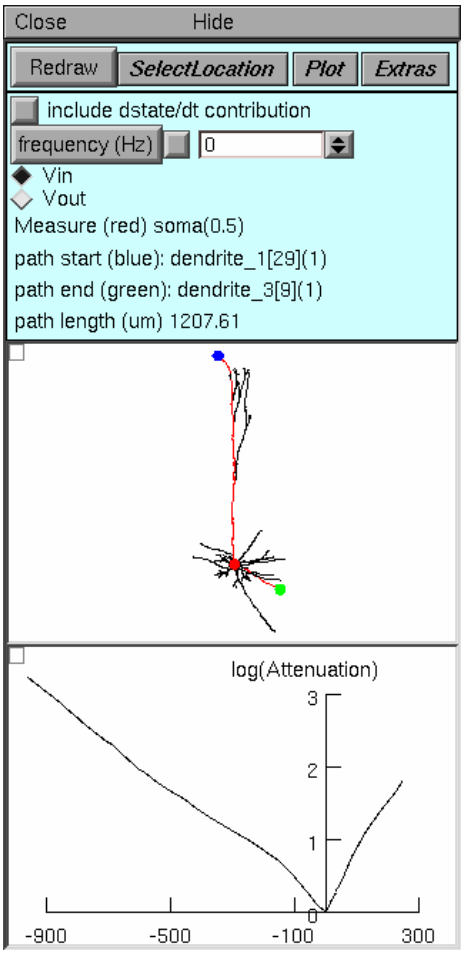
1. Move the injection and measurement sites to different locations in the cell and Redraw.
2. What happens to attenuation when the injection and measurement sites are swapped?
3. How do Zin and Ztransfer vary with frequency and location in the cell?
4. Is Ztransfer sensitive to direction (swap the inject and measure sites to find out)? How does this differ from attenuation, and why?

The Path tool

The Path tool is useful when the general location of the signal source is known, or when there are several independent signal sources at different locations. It performs the same kind of analyses that the Frequency tool does, but it allows the user to examine signaling between an electrode and a

region instead of a specific location.

Bring up the Path tool: NEURON Main Menu / Tools / Impedance / Path



Top panel. Note particularly

- The frequency field editor, which sets the frequency at which ln(A) is calculated.
- The Vin and Vout buttons, which specify the direction of signal flow relative to the electrode.

Middle panel. Click on the Shape Plot to set the location of the electrode (red dot), the path start (blue dot), and the path end (green dot). The direct path between the start and end is highlighted in red. Anatomical distance along this path is the independent variable against which analysis results of are plotted.

Bottom panel. For all points on the path, shows natural log(A) for voltage spreading toward (Vin) or away from (Vout) the electrode, according to the choice set in the Top panel. The path start is on the left of the horizontal axis, and the end is on the right. If the electrode is located on the path, this is the distance from the electrode in um; if the electrode is not on the path, this is the distance to the location on the path that is closest to the electrode.

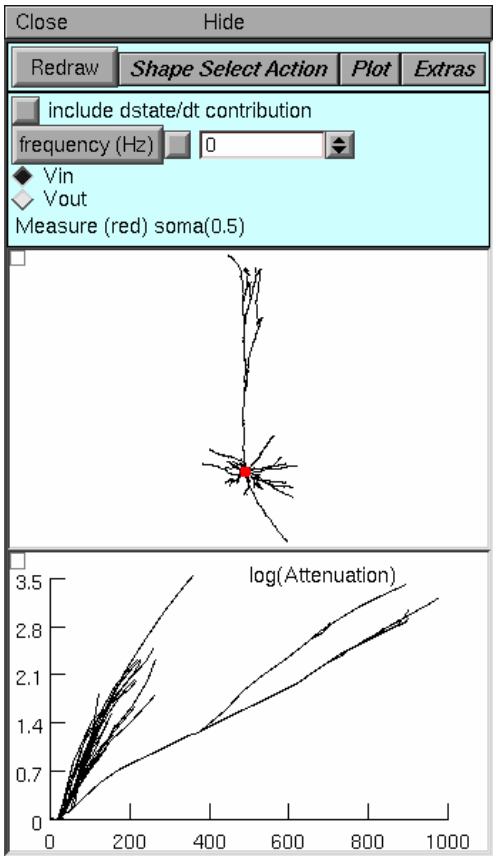
Things to do:

1. Leave the measure location at the soma, and move the path start and end to the positions shown above.
2. Examine the spatial profile of voltage attenuation in the somatopetal (Vin) direction. Where does Vin attenuation increase most rapidly with distance (where is the plot steepest): along the apical path or along the basilar path?
3. Switch to the Vout (somatofugal) direction. What happens to the magnitude and spatial profile of attenuation? Now where does it increase most rapidly?
4. How does frequency affect attenuation in the Vout direction (compare 0 Hz and 30 Hz)? in the Vin direction?
5. What does membrane resistance do to attenuation? Bring up the "Distributed Mechanism/Manager/Homogeneous spec" window and use its MechType button to view the parameters of the passive mechanism.
   The axon of the cell, which has the hh mechanism but not the pas mechanism, should be black in the shape plot for this manager.
   Change g_pas from its default value of 0.0001 (Rm = 10000 ohm cm^2) to 0.00002 (Rm = 50000 ohm cm^2). Then press the Redraw button in the Path tool window. Does increasing Rm make attenuation more or less sensitive to frequency?
6. How does cytoplasmic resistivity affect attenuation? Notice the axial resistivity in the NEURON Main Panel. This should be 100 ohm cm. Try doubling or halving this value and see what happens to attenuation. Don't forget to press Redraw in the Path tool window.

The log(A) vs x tool

This tool is the ultimate extension of the approach used by the Path tool: it shows the log of voltage attenuation for each point in the cell relative to the electrode or reference point.

Bring up the log(A) vs x tool: NEURON Main Menu / Tools / Impedance / log(A) vs x



Top panel. The only significant difference from the Path tool is the button labeled Shape Select Action. This button enables two important operations in the Shape Plot (middle panel). The first is "Move electrode" which lets you move the electrode to a new location by clicking on a neurite. The second is "Show Position" which helps you discover the mapping from the Shape Plot to the log(A) vs x plot: click on a neurite to see both it and the corresponding line in the log A vs. x plot turn red. The bottom panel's graph menu has a "Show position" item that does the same thing in the opposite direction.

Middle panel. Since attenuation is computed over the entire cell, the only location the user specifies is the position of the electrode (red dot).

Bottom panel. For every section throughout the cell, this panel shows ln(A) for voltage spreading toward (Vin) or away from (Vout) the electrode. The abscissa is the distance in um along the direct path from the ***soma*** (not the electrode) to each point. To discover which neurite corresponds to a line in this graph, click on the menu box (square in upper left corner of this graph) and select the "show position" item. Then click on a line to see it and the corresponding neurite turn red.

Things to do:

Leave the electrode at the soma.

1. Which of the traces in the bottom panel correspond to the
    - basilar dendrites?
    - primary apical dendrite and its major branches?
    - distal apical tuft?
    - axon?
2. How does the plot change with frequency? g_pas? axial resistivity?
3. For the Vin direction, in what part of the cell does attenuation increase most rapidly with distance? What about the Vout direction?

Follow these steps to discover passive normalization for yourself!

1. Leave the electrode at the soma.
2. Switch the plot to voltage transfer (click on the Plot button and select the V(measure)/V(inject) item).
3. Click on the Vin radio button, then click on Redraw.
4. Use the bottom panel's View = plot so you can see the entire range of y values.

This graph now shows a plot of the somatic response to a 1 mV signal as a function of the distance between the soma and the location where the signal is being applied to the cell. If synapses were

voltage sources, this is how the somatic PSP would vary as a function of synaptic location. Based on this result, you would expect synaptic efficacy to decline most rapidly with distance in the basilar dendrites.

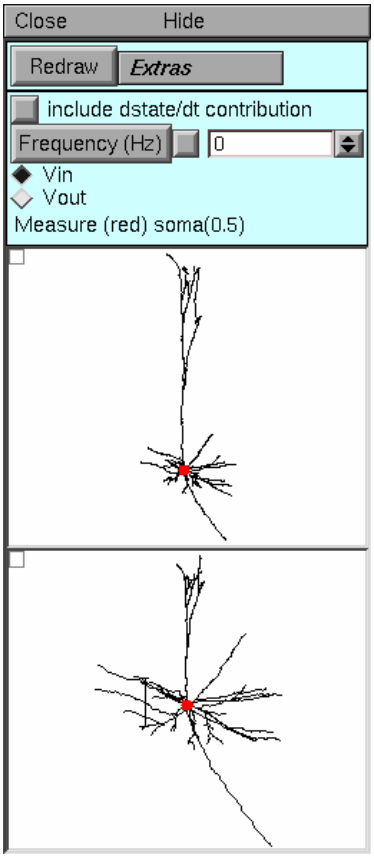
But synapses aren't voltage sources. They're much more like current sources. In other words, a synapse would deliver nearly the same current to a neuron regardless of where it is attached to the cell. Therefore voltage transfer ratio in the Vin direction (from synapse to soma) does not predict the relationship between synaptic efficacy and synaptic location. Instead, **the best predictor of synaptic efficacy is normalized transfer impedance.** This is identical to the voltage transfer ratio in the ***Vout*** direction!

So just click on the ***Vout*** radio button and you see that a synapse attached to a basilar dendrite will produce nearly the same somatic PSP no matter how far it is from the soma! This is the phenomenon that David Jaffe and I call passive normalization : variation of somatic PSP amplitude with synaptic distance is reduced ("normalization"), and it doesn't require active currents to happen ("passive"). For more information, see our paper.

The Shape tool

Perhaps the most intuitive representation of electrotonic architecture is to redraw the branched anatomy of the cell in a way that preserves the relative orientation of the branches, using line segments that are proportional to natural log(A) between adjacent points instead of the anatomical branch lengths. These ***neuromorphic renderings*** of the electrotonic transform warp the anatomy of the cell so that the proximity of points to each other is a direct indication of the degree of electrical coupling between them: tightly coupled points appear close to each other, and points that are electrically remote from each other are shown farther apart. The overall form of a **neuromorphic figure** parallels cellular anatomy, so it is easy to identify structural features of the cell, such as basilar or apical dendrites and particular dendritic segments or branch points.

Bring up the Shape tool: NEURON Main Menu / Tools / Impedance / Shape



Top panel. The controls for the Shape tool are very simple. Because of the direct visual parallels between the Shape plot (middle panel) and the form of the neuromorphic rendering (bottom panel), there is no need for special functions to demonstrate the correspondence between lines in these two panels. There is no Plot button because there is no way to represent Zin or Ztransfer by changing branch lengths in the neuromorphic figure.

Middle panel. This shows the anatomy of the cell and the location of the electrode or reference point (red dot), as in the log(A) vs x style.

Bottom panel. This displays the neuromorphic rendering of one of the components of the electrotonic transform. The distance of a point from the site of the electrode is proportional to the natural logarithm of attenuation for voltage spreading toward (Vin) or away from (Vout) the electrode, according to the selection in the Top panel. The calibration bar represents one log unit of attenuation, i.e. the distance that signifies an e-fold decay of voltage.

Things to do:

Leave the electrode at the soma.

1. Examine the Vin and Vout transforms at 0 Hz. How does the overall electrotonic extent of the cell vary with direction of signal transfer? Which parts of the cell are responsible for most attenuation in the Vin transform (ignore the axon)? in the Vout transform?
2. Change the frequency to 1, 3, 10, 30, and 100 Hz. At what frequency do you first see a noticable increase of electrotonic extent? Does this frequency depend on the direction of signal transfer? Note: to ensure similar sensitivity for detecting relative changes in the Vin and Vout transforms, first apply View = plot at 0 Hz.
3. Move the reference point to different locations and see what happens to the Vin transform. Do the same for the Vout transform. Can you explain the effect of changing the reference point? Hint: see Fig. 2 in Carnevale et al. 1995.

Citations

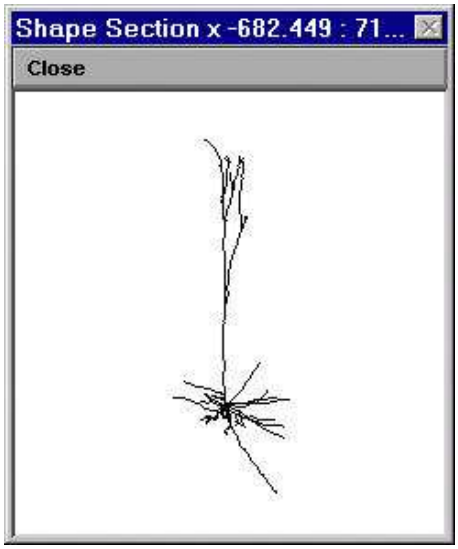
Carnevale, N.T., Tsai, K.Y., Claiborne, B.J., and Brown, T.H. The electrotonic transformation: a tool for relating neuronal form to function. In: Advances in Neural Information Processing Systems, vol. 7, edited by G. Tesauro, D.S. Touretzky, and T.K. Leen. Cambridge, MA: MIT Press, 1995, p. 69-76. Posted in html format at http://www.neuron.yale.edu/neuron/static/papers/NIPS94/nipsfin.html or see this local copy.

Jaffe, D.B. and Carnevale, N.T. Passive normalization of synaptic integration influenced by dendritic architecture. Journal of Neurophysiology 82:3268-3285, 1999. Preprint available from http://www.neuron.yale.edu/neuron/static/papers/jnp99/pasnorm.pdf

---

Working with Shape plots

This is the Shape plot for the Pyramidal cell model in `neurondemo` as it first appears on the screen. Its X axis is horizontal, the Y axis is vertical, and the Z axis is perpendicular to the page. Shape plots can be rotated, magnified or reduced, and translated.



Rotation

1. R click in the graph window and select 3D Rotate.
2. Lift the mouse carefully off the pad, so that its cursor remains in the graph window.
3. Press the L mouse button and hold it down.
4. Use the A, X, and Z keys to examine the cell from three orthogonal directions.

   A top view down the Y axis

   X side view down the X axis, i.e. from the right

   Z side view down the Z axis (standard view seen above)
5. Place the mouse back down on the pad, press the L mouse button, and run the mouse around on the pad to see the cell rotate wildly.
6. Restore the standard view by pressing Z while holding the L mouse button down.
7. Use ^X, ^Y, or ^Z (^ = Ctrl key) to rotate the cell in 10^{o} increments around the X, Y, or Z axis.

Other operations

Use the secondary menu of the Shape plot window to

- zoom in or out by 10% increments
- click and drag to open a NewView that focusses on a particular area
- zoom in or out continuously (Zoom in/out) by clicking and dragging to R or L
- click and drag the image around the window (Translate).
  Note: you can also move the image by holding down the shift key and then doing click and drag

When you're done

be sure to set View/Section so you don't inadvertently cause other changes to the image.

Hopfield Brody synchronization (sync) model

The exercises below are intended primarily to familiarize the student with techniques and tools useful for the implementation of networks in *Neuron*. We have chosen a relatively simple network in order to minimize distractions due to the complexities of channel kinetics, dendritic trees, detailed network architecture, etc. The following network uses an artificial integrate-and-fire cell without channels or compartments. There is only one kind of cell, so no issues of organizing interactions between cell populations. There is only one kind of synapse. Additionally, suggested algorithms were chosen for ease of implementation rather than quality of results.

Although this is a minimal model, learning the ropes is still difficult. Therefore, we suggest that you go through the entire lesson relatively quickly before returning to delve more deeply into the exercises. Some of the exercises are really more homework projects (eg design a new synchronization measure). These are marked with asterisks.

As you know, *Neuron* is optimized to handle the complex channel and compartment simulations that have been omitted from this exercise. The interested student might wish to convert this network into a network of spiking cells with realistic inhibitory interactions or a hybrid network with both realistic and artificial cells. Such an extended exercise would more clearly demonstrate Neuron's advantages for performing network simulations.

• Standard intfire implementation (eg IntFire1 from intfire1.mod))

Individual units are integrate-and-fire neurons.

The basic intfire implementation in neuron utilizes a decaying state variable (*m* as a stand-in for voltage) which is pushed up by the arrival of an excitatory input or down by the arrival of an inhibitory input ($m = m + w$). When *m* exceeds threshold the cell "fires," sending events to other connected cells.

```
if (m > 1) { ...

            net_event(t)    // trigger synapses
```

• IntIbFire in sync model

The integrate-and-fire neuron in the current model must fire spontaneously with no input, as well as firing when a threshold is reached. This is implemented by utilizing a *firetime()* routine to calculate when the state variable *m* will reach threshold assuming no other inputs during that time. This firing time is calculated based on the natural firing interval of the cell (*invl*) and the time constant for state variable decay (*tau*). When an input comes in, a new firetime is calculated after taking into account the synaptic input ($m = m + w$) which perturbs the state variable's trajectory towards threshold.

• Cell template

IntIBFire is enclosed in a template named "Cell." An instantiation of this template provides

access to the underlying mechanism through object pointer *pp*. Execute the following:

oc> objref mycell

oc> mycell = new Cell()

oc> print mycell.pp, mycell.pp.tau, mycell.pp.invl

The Cell template also provides 3 procedures. *connect2target()* is optionally used to hook this cell to a postsynaptic cell.

- **Network**

  The network has all-to-all inhibitory connectivity with all connections set to equal negative values. The network is initially set up with fast firing cells at the bottom of the graph (Cell[0], highest natural interval) and slower cells at the top (Cell[ncell-1], lowest natural interval). Cells in between have sequential evenly-spaced periods.

How it works

The synchronization mechanism requires that all of the cells fire spontaneously at similar frequencies. It is obvious that if all cells are started at the same time, they will still be roughly synchronous after one cycle (since they have similar intrinsic cycle periods). After two cycles, they will have drifted further apart. After many cycles, differences in period will be magnified, leading to no temporal relationship of firing.

The key observation utilized here is that firing is fairly synchronized one cycle after onset. The trick is to reset the cells after each cycle so that they start together again. They then fire with temporal differences equal to the differences in their intrinsic periods. This resetting can be provided by an inhibitory input which pushes state variable *m* down far from threshold (hyperpolarized, as it were). This could be accomplished through an external pacemaker that reset all the cells, thereby imposing an external frequency onto the network. The interesting observation in this network is that pacemaking can also be imposed from within, though an intrinsic connectivity that enslaves all members to the will of the masses.

- **Exercises to gain familiarity with the model**

  - **Increase to 100 neurons and run.**

    Many neurons do not fire. These have periods that are too long -- before they can fire, the population has fired again and reset them. Notice that the period of network firing is longer than the natural periods of the individual cells. This is because the threshold is calculated to provide this period when *m* starts at 0. However, with the inhibition, m starts negative.

  - **Narrow the difference between fast and slow cells so as to make more of them fire.**

    Alternatively, increase the delay.

- **Reduce the inhibition and demonstrate that synchrony worsens.**

  With inhibition set to zero, there is no synchrony and each cell fires at its natural period.

- **Increase cell time constant.**

  This will destroy synchrony. Increase inhibitory weight; synchrony recovers. This is a consequence of the exponential rise of the state variable. If the interval is short but the time constant long, then the cell will amplify small variations in the amount of inhibition received.

Beyond the GUI -- Saving and displaying spikes

- **Spike times are being saved in a series of vectors in a template: sp.vecs[0] .. sp.vecs[ncell-1]**

  Count the total number of spikes using a *for* loop and *total+=sp.vecs[ii].size*

- **We will instead save spike times in a single vector (*tvec*), using a second vector (*ind*) for indices**

  oc> load_file("ocomm.hoc") // additional routines

  oc> savspks() // record spike times to tvec; indices to ind

  oc> run() // or hit run button on GUI

- **Make sure that the same number of spikes are being saved as were saved in sp.vecs[]**

  oc> print ind.size,tvec.size

  - **Wise precaution -- check step by step to make sure that nothing's screwed up**

  - **Can use *for ... {vec.append(sp.vecs[ii])* vec.sort tvec.sort vec.eq(tvec)} to make sure have all the same spike times (still doesn't tell you they correspond to the same cells)**

- **Graph spike times -- should look like SpikePlot1 graph**

  oc> g=new Graph()

oc> ind.mark(g,tvec) // throw them up there

oc> showspks() // fancier marking with sync lines

Synchronization measures

- **Look at synchronization routine**

oc>syncer()

oc>for (w=0;w>-5e-2;w-=5e-3) {weight(w) run() print w,syncer()}

- **Exercise*: write (or find and implement) a better synchronization routine**
- **Graph synchronization**

```
oc> for ii=0,1 vec[ii].resize(0)         // clear
oc> for (w=0;w>-5e-2;w-=5e-3) {
        weight(w)
        run()
        vec[1].append(w)
        vec[2].append(syncer())
    }
oc> print vec[1].size,vec[2].size         // make sure nothing went wrong
oc> g.erase()         // assuming it's still there, else put up a new one
oc> vec[2].line(g,vec[1])         // use "View = plot" on pull down to see it
oc> vec[2].mark(g,vec[1],"O",8,2,1)         // big (8) red (2) circles ("O")
```

  - **Make sure that the values graphed are the same as the values printed out before**

- **Exercises**
  - **enclose the weight exploration above in a procedure**
  - **write a similar routine to explore cell time constant (param is called ta; set with tau(ta)); run it**
  - **write a similar routine to explore synaptic delay (param is called del; set with delay(del)); run it**

- *** write a general proc that takes 3 args: min,max,iter that can be used to explore any of the params**

  (hint: call a general *setpar()* procedure that can be redefined eg *proc setpar() { weight($1)* } depending on which param you are changing

Procedure interval2() in ocomm.hoc sets cell periods randomly

- **can be used instead of interval() in synchronize.hoc**
- **randomizing cell identities is easier than randomizing connections**
- **with randomized identities can attach cell 0 to cells 1-50 and not have interval uniformity**
- **To replace interval() with interval2(), overwrite interval():**

  oc> proc interval () { interval2($1,$2) }

- **Run interval() from command line or by changing low and high in GUI panel**
- **Check results**

  oc> for ii=0,ncell-1 printf("%g ",cells.object(ii).pp.invl)

- **Exercise: check results graphically by setting wt to 0, running sim, and graphing results**

Rewiring the network

All of the programs discussed in the lecture are available in ocomm.hoc. The student may wish to use or rewrite any of these procedures. Below we suggest a different approach to wiring the network.

- **procedure wire() in ocomm.hoc is slightly simplified from that in synchronize.hoc but does the same thing**

```
proc wire () {
  nclist.remove_all()
  for i=0,ncell-1 for j=0,ncell-1 if (i!=j) {
    netcon = new NetCon(cells.object(i).pp,cells.object(j).pp)
```



```
      nclist.append(netcon)
    }
  }
```

- **Exercises**
  - **rewrite wire() to connect each neuron to half of the neurons**

    suggestion: for each neuron, pick an initial projection randomly

    eg

    ```
    rdm.discunif(0,ncell-1)
    proj=rdm.repick()}
    if (proj < ncell/2) {
      // project to 0->proj
    } else { // project to proj->ncell-1
    ```

    This algorithm is not very good since cells in center get more convergence
  - *** rewrite wire to get even convergence**

    suggestions: counting upwards from *proj*, use modulus (%) to wrap-around and get values between 0 and ncell-1
  - **run(), graph and check synchrony**
  - **generalize your procedure to take argument *pij=$1* that defines connection density**
  - *** assess synchrony at different connection densities**

Assessing connectivity

- ***cvode.netconlist(cells.object(0).pp,"","")* gives a divergence list for cell#0**
- ***cvode.netconlist("","",cells.object(0).pp)* gives a convergence list for cell#0**
- **Exercise: use these lists to calculate average, min, max for conv and div**

Graphing connectivity

- **use *fconn(prevec,postvec)* to get parallel vecs of pre and postsyn cell numbers**
- **use *postvec.mark(g,prevec)* to demonstrate that central cells get most of the convergence (if using original suggestion for *wire()* rewrite)**
- **use *showdiv1(cell#)* and *showconv1(cell#)* to graph connections for selected cells**
- *** Exercise: write a procedure to count, print out and graph all cells with reciprocal connectivity, eg A->B and B->A**
- *** Exercise: modify your *wire()* to eliminate reciprocal connectivity**

Animate

- **Use *animplot()* to put up squares for animating simulation**
- **Resize and shift around as needed but afterwards make sure that "Shape Plot" is set on pulldown menu**
- **After running a simulation to set tvec and ind, run *anim()* to look at the activity**

***** Difficult or extended exercises

Last updated: Jun 17, 2003 (16:35)

Numerical Methods Exercises

Exercise 1:

Consider the differential equation $\frac{dy}{dt} = y^2$.

Note: $y = 0$ is a fixed point; i.e. the derivative there is 0. This means if y is ever 0, then y would stay 0 forever. Thus if y is ever negative, then y stays negative for all time. Likewise if y is ever positive, then y stays positive for all time.

Analytically solve this differential equation using the method of separation of variables subject to the initial condition $y(0) = -1$. Note that the solution has a singularity at $t = -1$.

What is the value of $y(1)$?

Thing to consider: Can every differential equation be solved this way? Can the Hodgkin-Huxley equations?

Exercise 2:

Use the Explicit Euler method to numerically solve the differential equation $\frac{dy}{dt} = y^2$ subject to the initial condition $y(0) = -1$.

Implement the algorithm yourself; do not use an ode solver library.

Graphically compare the solutions you obtain from several different choices of timestep with each other and with the exact solution you found in exercise 1. Plot the error at $t = 1$ versus the time step. Describe the relationship between the two.

Bonus challenge: implement the algorithm in a way that allows easily switching to solve a different differential equation.

Bonus challenge: implement the algorithm in a way that solves systems of differential equations. Test with $\frac{dx}{dt} = -y$ and $\frac{dy}{dt} = x$ where $x(0) = 1$ and $y(0) = 0$. For this example, plot x and y versus t. In a separate graph, plot y versus x. Describe the shape of the trajectory in this second plot.

Exercise 3:

Repeat the main part of exercise 2 (not the bonus challenges) using the Implicit Euler method instead.

Hint: Your code will have to solve a quadratic. Remember that the solution to $ax^2 + bx + c = 0$ is $x = (-b \pm \sqrt{b^2 - 4ac})/(2a)$.

Hint: To decide whether to add or subtract, recall the observation in exercise 1 that $y < 0$ for all time.

Bonus challenge: Repeat the same exercise except with the Implicit Midpoint Method: $y_{n+1} = y_n + dt\, f(\frac{1}{2}(y_n + y_{n+1}))$. Although the formula and process is similar to Implicit Euler, the error convergence is different. How might this difference in error convergence rates affect your choice of integration method?

Exercise 4:

A major component of exercise 3 was using the quadratic formula to solve for y or for Δy. Most algebraic functions, however, do not have closed form solutions.

In particular consider the function: $f(x) = e^x - x - 2$

Since $f'(x) = e^x - 1$ is positive for $x > 0$, f is monotonic on that domain, so it has at most one root (place where it is 0) there. Further as f is continuous, $f(0) = -1$, and $\lim_{x \to \infty} f(x) = \infty$, by the intermediate value theorem there must be at least one root on $x > 0$. Therefore there is exactly one root on $x > 0$. A similar argument shows there is exactly one root on $x < 0.$

Plot the graph of f on an appropriate domain to convince yourself that there are in fact exactly two roots and to identify their approximate locations.

Newton's Method is one strategy for locating roots. Starting with a point x_0, we can define a sequence $x_{n+1} = x_n - f(x_n) / f'(x_n)$. Intuitively, this algorithm chooses the next point to be where the function would be 0 based on its slope if it were a linear function. Under appropriate conditions on f, the sequence will converge to a root.

Use Newton's Method to approximate *both* roots of $f(x) = e^x - x - 2$.

Exercise 5:

Newton's Method required us to know the derivative of the function. If that information is not available, it may be approximated by the forward difference

$$f'(x) \approx (f(x+dx) - f(x)) / dx$$

or by the central difference

$$f'(x) \approx (f(x+dx) - f(x-dx)) / (2\,dx)$$

for small values of dx.

For both approximations, estimate the value of $f'(1)$ for $f(x) = e^x - x - 2$ for several choices of dx. How does the errors compare for the two approaches as $dx \to 0$?

Modify your Newton's Method code from exercise 4 to work for an arbitrary function by approximating the derivative using the approximation from this exercise that converges most quickly.

Exercise 6:

We cannot simply choose arbitrarily small choices of dx to get better approximations. While this would work in exact arithmetic, computers have limited precision due to how they represent numbers internally (usually IEEE 754).

Consider the following Python session:

```
>>> 1 + 1e-10 == 1
False
>>> 1 + 1e-100 == 1
True
```

Clearly in both cases the two quantities are not equal, but in the second case we have numerical equality.

Find the value of *machine epsilon*, which is defined as the smallest number that when added to one is not numerically equal to 1.

Find two positive numbers a and b with b less than machine epsilon, but $a + b \neq a$ numerically.

