A NEURON model template and organizational tool

Marianne J. Bezaire, Calvin J. Schneider, and Ivan Soltesz

January 10, 2013

Todo list

Abstract

Keywords computational, modeling, NEURON, hippocampus, GUI, MATLAB, organizing, versioning

Introduction

- motivation: We were motivated to structure our code and produce these tools to increase the accessibility of our model, both to other modelers and to experimentalists. Model code is often difficult to read and poorly documented, the final models not as transparent as wanted - er, the biological relevance of them uncertain.

Here, we aimed to increase the accessibility to experimentalists by creating an AutoRig tool that characterizes various aspects of the model within experimental protocols. This includes ion channel IV curves and activation/inactivation curves, single cell current clamps, and paired cell recordings. Additionally, the RunOrganizer makes it easy to produce network level outputs that can be compared to experimental data .

a. Modeling at a detailed level (individual cells, detailed synapse and ion channel characterization) is quite common now i. At least 1176 models have been published using NEURON software as of January 2012 (nrnwebsite) ii. That number doesn't include the many models developed using other simulation methods, such as GENESIS (genesiswebsite) or MATLAB. iii. Many of these models are of networks, which incorporate detailed connectivity information (synapse strengths and kinetics) iv. These models also often include physiologically detailed cells that incorporate ion channel mechanisms (with characteristic activation, inactivation processes and dependence on voltage or other conditions). v. These details are added to enable the model to better represent biological reality, but we often don't fully appreciate them (characterize) b. Models generally can and should be made more transparent i. Increased transparency aids comprehension, validation, reproducibility 1. At the level of the code a. Consistent structure, organized/grouped logically 2. At the level of the model behavior a. breaking down the behavior of the model as we would do experimentally ii. But characterizing models can be time-consuming c. Characterizing models using computational analogs of common experimental procedures would enhance model transparency i. single cell recordings ii. paired cell recordings iii. ion channel characterization (IV curve, activation curve, inactivation curve) d. Organizing the model code not only makes it easier for other programmers to understand, it also enables automation of the characterization process

We also aimed to increase the accessibility to other modelers. The code has already been used by a variety of people , even as a learning tool . However,

1: really? single cell voltage trace, but no LFP

2: cite

3: cite

we feel that it can be made even more accessible and could possibly serve as a template for other modelers who will eventually share their code as well. We produced a detailed user manual and documented the code extensively. We organized the code and broke it into smaller modules or files where it made sense. We parallelized it and shifted the connection-making mechanism to compiled code to speed it up. We introduce a tool to organize the runs and results and to easily reproduce any run. As well as to produce standard output figures and analyses.

With this system, it becomes much easier to reproduce modeling results. By tracking both the code version and parameters used, we ensure that the exact model code can be rerun. This tool also makes it easy to see what models have already been run, to look up any particular simulation result, and to compare simulation results from different runs.

- scope: NEURON code (Carnevale and Hines, 2006) with MATLAB tool (The Mathworks, Inc, Natick, MA) and Mercurial versioning (Babenhauserheide et al., 2013) - how this solution compares or fits in with other tools (NeuroConstruct, NeuroML).

1 Methods

Model Code

Acquiring and setting up the code

The code for this model can be downloaded from ModelDB (Hines et al., 2004), accession # . After extracting the zip file, a folder containing the model code will be generated. The general organization of the folder is shown in Figure 1.

4: get accession number

Next, create a Mercurial repository for this model. If you haven't used Mercurial before, download it from mercurial.selenic.com and, after installing it, set up your user profile. Now you are ready to use Mercurial. At the command prompt, change the directory to the main folder for this model. Then enter the command:

```
hg init
```

at the command prompt. This will generate a new repository for the model. After creating the repository, then commit the latest version so that the changes are saved, by entering this command:

```
hg commit -m "First version of the ModelDB code"
```

Now, check the log to see that the version was committed by entering:

```
hg log
```

which should return a result like:

```
changeset:  0:0ca73aafe064
user:  Your Name <your.name@school.edu>
date:  Wed Nov 10 12:03:28 2010 -0800
summary:  First version of the ModelDB code
```

Mercurial uses a file called `.hgignore` to determine which file to track and which to ignore. As we walk through each of the various folders and files of the model, we will also comment on those folders containing files that are ignored by Mercurial. Within the main folder are 1) the main hoc file for the model, called , 2) the .mod files that define the subcellular mechanisms of the model, and 3) the folders containing the auxilary files of the model. The folders include the following:

5: name of file

- **cells** The definitions for each cell type
- **cellframes** The definitions for each cell type, but without the synapse information
- **connections** Files defining different connectivity combinations
- **synapses** Files defining different synapse strength and kinetics combinations
- **stimulation** Files defining different stimulation protocols
- **results** A folder where the results will be stored, with each set of simulation results stored in a new folder within the results folder

These components will be discussed in more detail in the following sections.

6: add more folder descriptions or delete folders

Figure 1: Diagram of the model directory.

Code Template

The model is set up with one main file. This file reads in a parameter file that specifies which other files will be loaded by the main file. The structure of the main file and its relation to the other files is diagrammed in Figure 2. The sections of the main file are as follows.

7: Explain about the celltype array somewhere

Load libraries This section loads the standard NEURON library as well as some class definitions. It defines a `ParallelNetManager` class (written by Michael Hines), which is used to set up a network that runs on parallel processors. This section loads a definition for the `ranstream` class, also written by Michael Hines, which is used to generate streams of random numbers in several procedures in the model. It also loads a `CellCategoryInfo` class used to

store cell type-specific parameters; one instance is created for each type of cell in the model. It then defines a procedure created by Michael Hines called the default_var proc. This procedure allows one to pass in arguments to the model at the command line in the execution command. Finally, this section loads a file that sets the customizable parameters of the model and a second file that sets some lesser varied parameters or parameters that cannot be set using the default_var proc.

Define cells and model size The first file in this section loads a list of cell types and numbers of each cell, populating that information into instances of the `CellCategoryInfo` class described above. The next file loads in the specified connectivity information: for possible combination of presynaptic cell type and postsynaptic cell type, the total number of connections, the weight of each connection, and the number of synapses per each presynaptic connection type are stored in vectors within the cellType array instance for that postsynaptic cell type. Next, a procedure is run that loads the definition for each cell type class. The next procedure calculates the size of the network. For each cell type, the number of cells listed in the chosen `cellnumbers_*.dat` file is added to a running count of cells. If certain cell types are subject to cell death, that is taken into account here. Even the artificial cells used for stimulation are included in the total cell count. Finally, a procedure is ran that calculates some parameters necessary for defining a 3D network.

The random number offset is also set here. This is a value used to ensure that each random number stream in the model is statistically independent. Two index values define the random number stream, the low index and the high index. To use an analogy to a track, the low index number determines which lane of the track you are in. Streams that use different low index numbers will never overlap. The high index number determines how far down the lane you start. Therefore, if the random number generators are going to be used 200 times during the simulation, and multiple generators are going to use the same low index number (be in the same lane), then the high index numbers should be more than 200 numbers apart (so that each generator starts over 200 numbers away from any other generator and that way they don't come to overlap after 200 uses). The random number offset defines how far apart the high index numbers will be.

The parameters file bears further explanation.

Set up parallel capability and write out run receipt - versioning tracking - run receipt The run receipt includes information about the run that will be useful for tracking or reproduction. For example, the code in the run receipt procedure executes a system command to grab the Mercurial version of the code that is in use and report that in the receipt. Since a modeler can update to a particular version of the Mercurial repository and then make changes on top of it, the run receipt function also checks for this. If it finds that changes have been made, it executes system commands to write out a list of files changed

8: explain how the offset varies with different uses of the random number generator

9: explain parameters file and refer to later section for RunOrganizer

(`hg_status.out`) and a line listing of each change made to the version of the code that is currently in use (`hg_diff.out`).

Create, identify, and position cells The cells were already assigned to a host processor as described above in the ?? section. Now, each processor creates the cells that were assigned to it by gid, and positions them using an algorithm that specifies their 3D coordinates given their gid and cell type.

Load balancing is an option not used in this model. However, it is described in a separate section.

Creating a topographical model This version of the model can be designed and run without regard to any model topography. However, the model can also be specified in three dimensions. Cells can be positioned uniformly within a three dimensional block, or uniformly within adjacent three dimensional layers . Connections can be constrained such that cells only make connections with cells within a certain distance, or preferentially make connections with close cells.

10: work on layer specification

11: Explain how this works

Connect the cells

Initialize and run the simulation; output result files The results of the simulation are saved in a folder with the same name as that particular simulation run, within the results folder of the model directory. The result files include:

- **spikeraster.dat** A list of spike times and GIDs of the cells that spiked is printed out. This is the main output of the simulation, used for most subsequent analysis of the model and simulation.
- **runtimes.dat** The amount of time (in seconds) spent on each section of the code by one processor (host 0) is printed out
- **voltage traces** traces
- **numcons.dat** Summary connection file that gives pre- and post- synaptic cell types and number of connections
- **connections.dat** File that gives pre- and post- synaptic cell gids and synapse types, optionally printed out

12: traces connections

- **sumnumout.txt** A summary file that gives the number of cell types in the model, total number of cells in the model, the total number of connections in the model, the total number of spikes by all cells during the simulation, and the total run time of the model. This file is loaded into the RunOrganizer and its values displayed there when the finished run is uploaded.
- **celltype.dat** File that gives cell name and gid range for each cell type

- **position.dat** Position file that gives the gid and x, y, and z coordinates of each cell, as well as the processor on which the cell resides
- **traces**

The code within these sections is extensively documented and should be referred to for further detail.

Figure 2: Diagram of the model code.

Parallelization and Simulation

choice of dt In this model, we use a fixed time step for the simulation. The value of `dt` (integration time step of the simulation) should be chosen with care: small enough for a reasonably accurate simulation but large enough that the simulation can be completed in a reasonable amount of time. We felt that a `dt` of 0.1 ms was sufficient for our simulations. However, in some cases a smaller dt may be necessary. This is especially true if the model network is highly active and currents may become large. .

13: perhaps talk about the Ca2+ bug here?

Parallelizing The most significant time-saver in the model was to parallelize it, such that multiple processors could solve equations for the model at the same time. To do this required using a parallel net manager, which set up the network such that each processor could know which processor owned every other cell, and also be able to talk to every other processor.

14: Vrest, vinit - what to do with them

For determining the connectivity of the model, it was necessary for each processor to know certain information about cells that were owned by other processors. Our way of dealing with this was to make these properties of the cell derivable from its gid, such that other processors could determine the needed information using only the gid of the cell. For example, the position of the cell... or the number of cells of each type?

load balancing Load balancing is a way of assigning cells to processors. Our code generally uses the round robin technique, in which cells are assigned uniformly to each processor such that each processor has approximately the same number of cells as any other processor, similar to how a dealer would deal cards out to each player in a game. However, if the cells differ widely in complexity, this may not be the most efficient way of assigning cells. It can be useful to take into account the complexity of the cells when assigning them to processors, such that the overall number of equations each processor has to solve are roughly equal. That is where load balancing comes in. To load balance the model, the cells of the model are created and the complexity of each cell type

is determined. Then, the model is re-created using the complexity information when assigning each cell to a processor, and the simulation proceeds.

Determining the model complexity takes some time, but only has to be done once per each version of the model where the cell definitions are changed. However, if the model is quite complex and the simulation time is long, it makes sense to invest a small amount of time in determining the complexity to save a large amount of time during simulation. In our case, the pyramidal cells are more complex than the interneurons and all the interneuron types are roughly equal in complexity. Because there are so many more cells than processors, the round robin distribution is sufficient to give each processor roughly the same amount of complexity and therefore we do not use load balancing. However, an option for load balancing is included in our model for explanatory purposes.

The `psolve` method of simulation

Ion Channel/subcellular mechanism Techniques

Ion channels are generally defined either using Hodgkin-Huxley formalism or kinetic schemes. While kinetic schemes may provide better accuracy, they require many more parameters than Hodgkin-Huxley style definitions. In many cases, Hodgkin-Huxley definitions can provide adequate fitting to experimental data with far fewer computations. These methods of defining ion channels are described in detail in Sterratt et al. (2011); Carnevale and Hines (2006).

- Type of ion channel models used and where they came from (ref and preparation). Temperature corrections. What is our rationale for using these models? Well, we want something that is biologically feasible is all. The behavior at the individual cell level is what is really important for us, and as long as we have a biologically feasible method of getting that behavior, we are satisfied even if it doesn't match a particular configuration found in some experimentally observed rat hippocampal neurons. - Syntax of the model - so that it is compatible with the GUI, g_max and myi_i. - Figures characterizing the ion channels. All here or just one with some in the appendix. Note that the later sections will explain how to create these figures for yourself.

- Not just ion channels; how do the calcium mechanisms work? If multiple calcium currents, then why? - How do the synapses work?

Model Cells

Within the `cells` folder is a file for each cell type in the model, named in the format "class_[celltype].hoc". Each cell file contains a class definition for the cell type following a standard template. The template includes:

- **public variables** those accessible outside of the class definition, within the main code, are defined first. They include properties of interest and those that need to be set in the main code, procedures and functions that need to be called from the main code, and references to morphological sections and lists of sections of the cell.

- **objects** some objects, the lists of synapse objects, are defined
- **external variables** those variables from the main code that need to be accessed by the cell class are then named
- **section creation** next, the morphological sections of the cell are created
- **initialization procedure** next, the initialization procedure is defined. This procedure is run immediately for each instance of the class upon instantiation.
- **function and procedure definitions** finally, each function and procedure called by the initialization procedure is defined.

The functions and procedures used include:

- `append_sections` This proc will append all the sections to their respective lists, such as lists for axon sections, lists for soma sections or dendritic sections, lists for presynaptic cell type possible synapse locations, or lists used to set the subcellular mechanisms
- `connect_sections` This proc will connect all the morphological sections to each other in the correct order
- `size_sections` This proc will set the diameter and length of each section or, alternatively, the diameter and 3D coordinates of the beginning and end of each section
- `insert_mechs` This proc will insert and set the parameters of the various subcellular mechanisms
- `set_nseg` This proc will set the resolution at which each section needs to be calculated during simulation to ensure a certain level of accuracy in the results
- `define_synapses` This proc will define the possible synapses for each presynaptic cell type. It will create a list of lists, where the top level of list specifies the possible presynaptic cell types and the second level of lists specifies the possible synapses for each presynaptic cell type. For each type, it iterates over all the possible synapse locations.

- synaptic input SynData - some figures of single cell current clamp data, plus associated ion channel currents. Note that these figures can be produced using the GUI explained later.

In the `cells` folder there is also a file per cell type detailing the axonal distribution of that cell type, named in the format "dist_[celltype].hoc". There are three numbers within the file, a, b, and c, which describe the axonal distribution of the presynaptic cell type. They correspond to an equation of the form:

$$P = \frac{1}{a}\exp\left(-\left[\frac{(D-b)}{c}\right]^2\right) \tag{1}$$

where the probability P is a number between 0 and 1 that describes the probability of a connection forming between two cells distance D apart.

There is also a `cellframes` folder containing class definition files for each cell type. These files are not directly used by NEURON. The purpose of this folder and these other class definition files will be explained in the SynData section.

Connectivity

Multiple possibilities exist for defining the connections between cells. Each one can be defined in its own file. Here, a connectivity file contains the procedure and function definitions as well as calls them to run. The name of the connectivity file is itself a parameter which is then used by the main code to load the proper connectivity file. - different options - ConnData - fastconn mechanism The connectivity part of the code takes the bulk of the setup time. This is because the total number of connections made is large, scaling exponentially with the number of cells in the model. Also, the decision of which cells to connect requires the computer to evaluate many possible connections, and this evaluation must take place for each connection formed . The decision of which connections to form depends on an element of randomness and, for topographical networks, the distance between cells and the axonal distribution. To speed up the connection process, I transferred the decision of which connections to make to a mechanism, so that it became compiled code. Hoc allows you to define new vector methods. So I defined a vector method that determined which cells to connect. I would pass in a bunch of parameters to the method by putting them in the vector. These parameters included the number of cells of the pre- and postsynaptic types, the number of connections to make, and for topographical models, the axonal distribution. Also, the gid ranges of each cell type which were used for calculating the position of each cell. Within the mechanism, these values were used to come up with a list of cell connections, in terms of the gids of the presynaptic and postsynaptic cell, as well as the synapse identifier (since more than one type of synapse or synapse location is available for each connection type). Then, these values were loaded into a results vector, which was returned from the vector method function. So in hoc, there was a results vector that was set to that return value. Then in hoc, the code iterates through that results vector and builds the chosen connections.

15: is that true?

A large amount of memory may be needed for this section. It will be needed to store the connections to be made in a vector that is passed between the hoc code and the fastconn vector method. The vector space must be set aside before using the fastconn method. It is important that a large enough vector be set aside, or else the fastconn method will error out . However, if too large of a

16: will it error out?

vector is set aside, the program will run out of memory. Therefore, the following formula is used to determine the vector size for the `conns2make` vector:

The number of connections to be made total (for that pre and postsynaptic cell type combination) is divided by the number of processors, assuming the connections will be spread out evenly across all processors. For buffer, the total number of postsynaptic cells is added to this number as well . Then, because for each connection, 3 data points need to be stored (the presynaptic gid, the postsynaptic gid, and ...?), and we want to double the saved space just to be safe (why?), we then multiply this number by 6 to get the total reserved vector length.

17: i don't remember why

- how the GUI can calculate the convergence and divergence.

18: allow conndata to be specified at the GUI level by divergence or convergence, which is then calculated out to total connection and saved in the file like normal.

Stimulation

Similar to the connectivity portion of the model definition, multiple strategies exist for stimulation, each in their own file. Each stimulation file contains the procedure and function definitions and calls them to run. The name of the stimulation file is introduced as a parameter to the main code and used to load the proper stimulation file.

NetStims are the artificial cells used for stimulation in the model. In general, they have parameters to specify their properties as random, Poission-distributed spike generators. However, they can also be used as single pulses by setting the number of spikes to 1. The properties used in these cells include:

- **start** The simulation time at which to start the NetStim spiking protocol, in ms
- **number** The number of spikes to execute
- **interval** The average interval at which to execute the spikes, in ms
- **noise** On a scale of 0 to 1, to what extent the interspike interval is randomly determined

It also has a procedure called `noiseFromRandom` that takes as an argument an instance of the RanStream class. The instance has its own high index and low index values that specify the random number stream. These index values should be chosen such that no instances of the RanStream in the model overlap. The instance also has a way to specify the type of random number generator used to set the interspike intervals. Examples of commonly used ones include a Poisson-distributed one and a normal one... This is explained in more detail in another section.

The NEURON website has an excellent explanation of the use of randomness for NetStims in parallel models (Randomness in NEURON models, http://www.neuron.yale.edu/neuron/node/5

Within this explanation is an example demonstrating the use of the `noiseFromRandom` procedure.

Note that if the number of spikes and the interval multiplied lead to a longer time than the simulation time, not all of the spikes set in the number field will occur. Also, note that the noise variable can be used to specify a minimum ISI even in the presence of some randomness. For example, if the ISI is set to 100 ms and the noise is set to 0.2, then the ISI will always be at least 20 ms, but the actual length will vary in a Poisson-distributed manner such that the mean ISI is 100 ms. To override these parameters and specify the precise spike times, the start time must be set to a negative number to essentially turn the NetStim 'off' and then a vector of spike times can be fed into the NetStim, as described later.

Various forms of stimulation can be applied to the model. Each one is defined in its own file, with the file name following the convention of '[stimulation type]_stimulation.hoc'. Some examples of stimulation type are given in the list below:

- **pulse** This method of stimulation applies a single pulse of excitation to the model. This comes in the form of many artificial cells spiking once at the same time . These artificial cells are connected to a subset of 'real' model neurons. The number of model neurons they connect to, and the number, strength, and kinetics of synapses they make on each one, can be altered and is editable at the level of cell type.

19: allow for flexibility in spiking?

- **spontaneous** This method of stimulation applies random, Poisson-distributed spiking activity in the artificial cells. It creates enough artificial cells for there to be independent stimulation of each 'real' cell in the network (or just those that are supposed to receive stimulation). . This form of stimulation is tonic in that it continues for the length of the simulation and has constant parameters throughout the simulation. As with the pulse method, the number, strength, and kinetics of the synapses onto the real model cells can be set per cell type.

20: what about realistic art cells that diverge/converge onto real cells?

- **spontburst** This method of stimulation is similar to the spontaneous method, except that it allows the stimulation to occur in set intervals throughout the stimulation. This means that the stimulation can last for a specified duration of time, and then be turned off for a specified duration of time before turning back on again. This pattern is repeated for the length of the simulation. This method of stimulation is useful for imposing an outside rhythm onto the model network.
- **thetaspont**
- **ripple**
- **singlecell**
- **vector** This method of stimulation allows precise control over the spike times of the artificial cell. For each artificial cell, there is a vector of spike

times. The vector data is loaded in from . It can be produced by another MATLAB script, for example the virtual rat environment. The data is organized by the gid of the cell, with each column giving the spike times.

21: where loaded, format

- **ecca3sintrain** This method is similar to spontaneous, except that the input patterns from the artificial cells are not random, Poisson-distributed spikes. Well, they are, but they are modulated by a gate, a sin-wave gate, so that selected times at the peak of the gate are more likely to produce a spike than those at the trough. Therefore, this pattern results in some level of spiking throughout the simulation, but the probability of spikes waxes and wanes with the modulating sine wave. There are two frequencies specified in this regime, the frequency of the modulation gate and the frequency of the spikes, which is the maximum frequency that you would see at the peak of the modulation gate.

- how they work with the connectivity options

22: Not all of these types should be included in the first instruction manual, but they should be added to the modeldb record as they are used in published work. Only pulse and spontaneous should be there at the start

Outputs

- gives examples of some outputs: spikeraster, numcons, voltage traces, input synapses for traced cells, runreceipt with times

RunOrganizer GUI

Acquiring the GUI

The GUI is available from two different locations. It can obtained from ModelDB's SimToolDB at http://senselab.med.yale.edu/SimToolDB/ . Alternatively, it can be downloaded from MATLAB's File Exchange at , entry # . Once the tool is downloaded to a folder, it should also have a Mercurial repository set up using the same commands as detailed above. The tool can be customized and it is likely that users will want to change some of the fields it tracks, so the Mercurial repository will be useful.

23: URL

24: MATLAB file number

Process Overview

Purpose of the GUI The RunOrganizer is meant to organize the simulations and assist in designing and running new simulations. It reinforces the following process: design a simulation by setting the parameters, then execute the simulation, then save the results, then analyze the results. And make the simulation results easy to get back to if necessary, and the simulation easy to repeat by tracking everything about it: the parameters used, the system run on, the version of the code, the particular connection and synapse sets. The RunOrganizer documents all of this, automating as much as possible so that you can spend more time thinking about designing and interpreting the model, and less time thinking about running it and storing files.

How the GUI works with Mercurial The RunOrganizer pulls in list of all available versions, allows you to specify one. Then, when you go to execute the code, it updates the current Mercurial version to the one you specified, both on your host computer and on the remote machine, if you are submitting the run to a remote machine. To update the code requires executing a specific Mercurial command:

```
hg update -C -r 1
```

where `1` is the version to which you would like to update. The `-C` command forces the files in the working directory to be updated to the specified version, even if there were uncommitted changes in the working directory. It also recompiles the mechanisms, calling `nrnivmodl`, as well as specifying the synapse definitions using the cell templates from the updated Mercurial version.

Note that it is possible to submit a new job with a specific Mercurial version to a batch queue while another job with a different Mercurial version is running or also waiting in the queue. This is undesirable as the Mercurial version will get updated to that specified by the job submitted most recently. The RunOrganizer has two safeguards in place to address this possibility. First, whenever a job is about to be submitted to a batch system, the RunOrganizer queries the queue list, pulls in the job names of any jobs waiting , and then compares the Mercurial version of those jobs to the one specified in the job to be submitted. If they are different, the RunOrganizer notifies you. Second, in the event that the version is updated and a previously submitted job is run with the wrong version, the use of that version will be documented in the run receipt. The version that was actually used will be noted in the RunOrganizer once the finished run is uploaded.

25: but not running, think if this makes sense for files loaded at end of run

To ready the remote machine to give the names of the runs waiting to be executed in the form the RunOrganizer wants, create a file on the remote machine that defines a new command called `testq`. So the name of the file should be `testq` and the contents of the file should read (all one line):

```
qstat -j ‘qstat -s p -u case | grep ’case’ | cut -f1 -d ’ ’ | tr ’\n’ ’,’ ‘
   | grep job_name | cut -f2 -d: | tr -d ’ ’
```

where `case` is replaced with the your username for the remote machine.

26: make sure apostrophes and quotes show up correctly

The RunOrganizer works in concert with a few other tools. These tools include the `AutoRig`, which allows clamping of cells to characterize ion channels, single cells, and synapses, `SynData`, which displays and allows easy updating of files defining the synapse kinetics of all synapses within the model, `ConnData` which displays and allows easy updating of files defining all the connections (numbers and weights) between cells of the model.

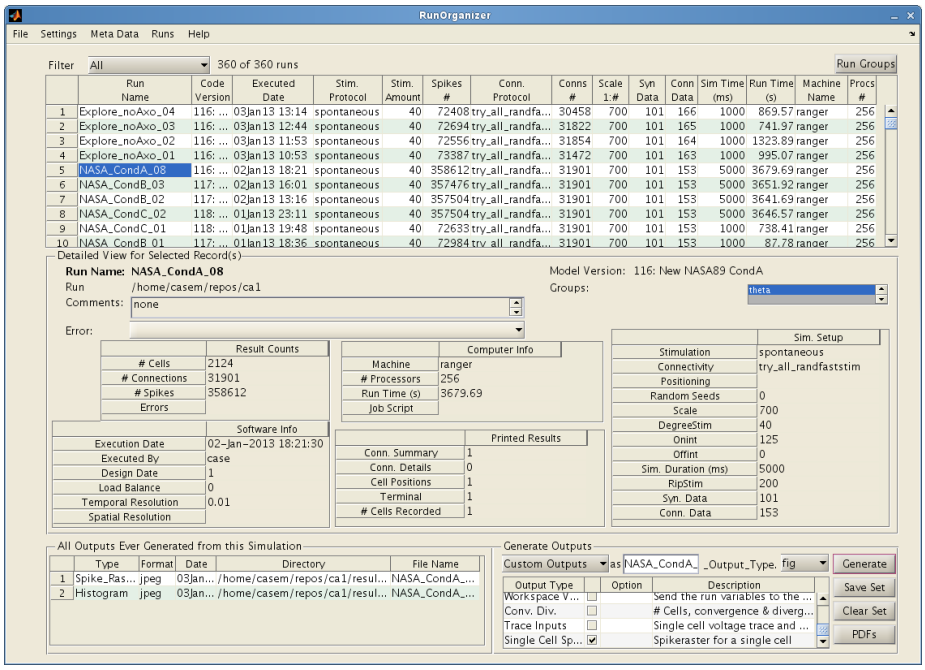


Figure 3: Screenshot of the RunOrganizer tool.

Walk thru GUI

A walk through the RunOrganizer

the list of runs All of the runs stored in the RunOrganizer can be viewed within the list view. The list view lists all of the runs, row by row, and displays several parameters associated with the runs. The parameters displayed in the list view can be customized, as described below in the 'Customizing the RunOrganizer' section.

the drop down to change the view The runs displayed in the list view can be filtered. The dropdown just above the list view gives options for various views:

- **All** Shows all runs stored in the RunOrganizer
- **Not Ran** Shows all runs that have been designed but have no execution date (have not been uploaded)
- **Ran** Shows all runs with an execution date (that have been uploaded)
- **Ran Without Error** Shows all runs with an execution date that do not have an error message (or, do have all the sumnumout fields?)

- **Errored Out** Shows all runs with a value in the error field
- **Find Error** Shows all runs that have an execution date, i.e., have been run, but do not have sumnumout values so that there must have been an error along the way. This view is useful for finding runs that need to have their error message updated but do not have that yet. The error updating is a manual process, as described later in the section on errors.
- **Similar to Selection Name** This view takes the name of the run currently selected, removes the characters after the last underscore, and find other runs that start with that same name. This takes advantage of the naming scheme for runs, where similar runs can be designed that have the same root name but different number suffixes after the underscore.
- **Group A, B, C** This view allows you to specify which of all the groups the run results should (or should not) be a member of.
- **Error A, B, C** This view allows you to specify which of all the errors the run results should (or should not be a member of.
- **Custom Filter** Create your own custom filter based on a single parameter. Enter the name of the parameter, the value to search for (or avoid), whether the parameter type is a string or a number, and optionally the type of search to do (= to find that value, avoid that value, etc).

the form view of the current run, which displays more parameters

the list of available analyses

the list of saved figures

the menubar The menu bar contains a number of options: Under the File menu:

- **New Repository** Track simulation runs in a new repository. Upon choosing this command, you will be prompted to pick the folder of the repository. If a Mercurial repository has not yet been established, you will be prompted to initialize one there.
- **Open** Saves the current data and then allows you to do one of three things: 1) open a different model directory 2) append archived or backed-up runs to the current dataset or 3) view an archived or backed up dataset on its own.
- **Export Data** Export the data in a tab-delimited format, which can be read by Excel.
- **Backup** Backup the RunArray and settings to a mat file, then open the folder of the file

- **Archive** Archive the selected RunArray records and settings to a mat file
- **Quit**

Under the Settings menu:

- **Parameter List**
- **Error List** This opens up a table of all the error values and allows you to edit them or add new ones. Note that, in general, existing error options should not be changed if they have already been used.
- **Outputs**
- **Machines** This allows you to add more remote machines, specifying how to connect to them and the requirements to submit jobs to the queues available on those machines (such as maximum run time and maximum number of cores)
- **General**

Under the Meta Data menu:

- **Parameters**
- **Performance**
- **Compare Runs** This allows the user to compare two or three runs side by side, looking at some of the most commonly used outputs (spikeraster, fft). It also shows which parameters each run had in common and their values, as well as which parameters varied between runs and their parameters.

Under the Runs menu:

- **Batch Upload Runs**
- **Upload One Run** Will upload the results of the currently selected run (overwrite if already loaded? fix subfolder issue). If the run was designed to be executed on another machine, it will log onto that machine and copy the run over if it exists.
- **Design Run** Blanks out the form view field values and makes them all editable so you can enter new run parameters in there, adds a Save button so you can save the run and see it in the run list.
- **Copy Run** Creates a fresh run record with all the same values as the selected one; it leaves the fields editable so you can change whatever you want before saving.
- **Edit Run** Displays an existing run in the form view and makes the fields editable so you can change parameters. Only works with runs that have not been executed yet (or sent for execution?)

27: what else should be changed

28: allow to see values of parameters used and combos of values

29: chart run times for various model sizes and parameters, per machine type and num proces

30: update this one

31: How to prevent duplicate name runs?

32: test new machine specifier on stampede or trestles

33: does this work?

34: code the edit run command

- **Delete Run** Deletes a run from the RunOrganizer, even if it has already been executed. Will ask for confirmation before deleting
- **Execute Run** Sets up the run (updates the repository to the specified version, recompiles the mechanisms, reprints the synapse info), then enters the execution command at the command prompt. If the machine is specified as a remote machine, the repository on the remote machine is readied (and the cell files copied over), a job script command file is produced and copied over, and then the jobscript is submitted to the batch queue.
- **Get Job Output File**
- **Execute Run Range** Instead of executing the selected run, it allows you to specify one parameter to vary and the values to use. Then, it creates a series of runs with similar names to the selected run, with all the parameters the same to the selected run except the one to vary. It then submits all of those runs to be executed.
- **Debug Commands** Rather than executing a run, this command sets up everything in preparation to be executed, but then writes out to the MATLAB desktop the commands to enter at the command line to set up debugging for the run. This is used for gdb for a serial run. For debugging in parallel...

35: What about duplicate name runs?

36: note the issue with overloading development queue

37: what to do

Under the Help menu:

- **RunOrganizer Help** Opens the instruction manual for the model and RunOrganizer
- **About RunOrganizer** Shows a dialog box that gives the version number and contact information

Design a run or copy a run First, create a new "run", which is a new record that corresponds to a particular simulation run. This causes the form fields to blank out and become editable. Specify the values wanted for each parameter. For those not specified, some default values will be used. The defaults are set in the `parameters.hoc` file. Once the values have all been specified, then click the `Save` button to add this run as a record in the RunOrganizer database. The run will appear in the list of runs that have not yet been executed. Runs that are not executed have blanks for several of the spaces, including the execution date, the runtime, and the total numbers of cells and spikes, etc. While specifying the run parameters, you also specified some execution parameters, such as the machine on which to run the simulation and how many processors to use.

Execute a run Once it is time to execute the run, choose the `Execute run` command from the `Tools` menu. What happens next will depend on what you entered in the `Machine` field of the run record. If you chose `localhost`, then the run will begin executing on your machine, using a command set by ... If

38: Make some way of specifying the localhost command, perhaps another jobscript

you chose a remote machine, the RunOrganizer will set up the run on that machine by updating the model code there and recompiling the mechanisms (actually it will do this for the local machine too) and updating the synapses. Then, it will create a job script on the remote machine and submit it to the batch queue. As the syntax used by each remote machine may differ, each machine has its own custom m-file that you should configure to ensure the commands are correct. Some sample m-files have been included for various machines (Stampede, Trestles, gateway??). Creating a new jobscript file is discussed in the appendix.

Upload a finished run The RunOrganizer does not stay in touch with the machine after job submission. Therefore, it is up to you to keep track of when the job finishes. After it is completed, use the RunOrganizer to retrieve the results. Oh, and the remote machine must have a directory set up just like the local one, with the same mercurial repository. So when you then click on the record in the RunOrganizer and say `Upload run`, the RunOrganizer will use the machine value that you entered, log onto that machine, and copy the folder with that runname onto your local machine. It will then read in some of the values from the run receipt and another output file with total counts (of cells and spikes, etc) and display those values in the fields in the run list and on the form view. It will update the values specified in the design with those actually used during the execution. In general, these values should always be the same. However, there are a few cases in which errors (not exactly, but irregularities) may cause different values to be used during execution. In that case, the values actually used should be displayed in the RunOrganizer, which is accomplished by overwriting the specified values with those reported in the run receipt.

Analyze a finished run When an uploaded run is selected in the RunOrganizer, the list of available analyses at the bottom right section of the RunOrganizer will be populated. Only the types of analyses available for that particular run will be shown. The resulting figures can be displayed on the screen as fig files or can be saved as a image files for later use. Once an analysis file has been saved, it is recorded in the list on the bottom left of the screen. That way you can keep track of which runs which figures came from. The files always include the runname in them. The files are saved in the results folder for that runname. The various outputs available are discussed in a later section, along with how to add new output types to the list that you have created yourself.

The data of the RunOrganizer

The SimRun class The data in the RunOrganizer is organized such that each run is its own record. The data of each run is stored in its own instance of the `SimRun` class. MATLAB has extensive documentation on classes, to which the reader is referred for more detail about classes. The `SimRun` class is a handle class (as opposed to a value class), which is useful for defining unique objects,

in this case, unique simulation runs. Within the class definition file are all the properties of the class. Methods for the class are also defined and include three functions:

- `SimRun` an initialization function
- `loadexecdata` a function that saves a bunch of execution-related properties into the SimRun instance, as well as does some updating with directory names: the local folder where the results will now be stored is saved in the ModelDirectory field, while the folder on the remote machine where the run was executed is listed in the RemoteDirectory field.

The SimRun class can be edited to meet your needs, as discussed later. It should be stored in its own folder, with the name of `@SimRun`, within the RunOrganizer folder. The definition file is reloaded each time you create the first instance of that class. If you edit the SimRun class definition but leave the RunOrganizer GUI open, instances of the previous class definition will still exist and so the class definition will not be updated. To update it, first close the RunOrganizer. Then enter `clear all` at the command line to remove the global variable `RunArray`, which holds instances of the SimRun class. Then relaunch the RunOrganizer.

Key fields in the SimRun class (that should never be removed) are:

- `RunName` The name of the run, this field uniquely identifies the run and is set by the user when first creating the SimRun instance. Its value should only be changed during the design phase. After the run is submitted, it should not be changed. However .... But, under no circumstances can it be changed after the executed run is uploaded back into the RunOrganizer.
- `ExecutionDate` This is the date and time that the run began executing. Its value is pulled from the run receipt, which writes out the date and time at the beginning of the code.
- `DesignDate` This is the date and time that the SimRun instance was first created, by saving data in the RunOrganizer.
- `DesignedBy` This field lists the username logged into the computer when the SimRun instance was first created.
- `Errors` This field is blank, set by the user at a later time point, from a drop down list.
- `Groups` This field is a text field that can contain multiple group names, one for each group the run is a member of. The user can manually add the run to various groups. The groups field is useful for searching various runs and displaying subsets of runs.
- `ExecutedBy` This is the name of the user who was logged in on the machine where this run was executed at the time of execution.

- `ModelDirectory` This field gives the directory where the model results are stored. Prior to uploading the results to the local computer, this field will contain the directory of the remote computer. After the results have been transferred to the local computer, this field is updated to the local computer and another field contains the path for the remote directory (RemoteDirectory)

39: maybe others, think about logic of this list

The RunArray The `RunArray` is an array of SimRun instances. This array stores all of the data from all of the runs. Each run has its own number within this array. However, that number can change if runs are deleted from the RunOrganizer (and hence the RunArray). Therefore, only the `runname` field uniquely and stably defines the run.

Backing up data

Reloading backed up data

Customize GUI

Change the SimRun class The definition of the SimRun class is in an m-file called `SimRun.m` within the `@SimRun` folder. You shouldn't need to update this file directly. Instead, within the RunOrganizer under Settings menu, choose Parameter List. Update the parameters there. Then, click the 'Print' button to generate a new SimRun file with the proper properties. After updating the file, you must clear all instances of the SimRun class before the new class definition will load. To do that, close the RunOrganizer. Then enter 'clear all' at the command line. Then, reload the RunOrganizer.

Change the fields displayed Within the Parameter List tool ... the list option, the form option. the file option sets what appears in the Parameters.hoc file...

How to create new views New view filters can be defined for the list view. To do this, first add the name of the new list view to the drop down. Then, define the filter criteria of the new view. To edit the drop down, first close the RunOrganizer. Then enter 'guide' at the command line. Choose the RunOrganizer.fig file. A figure of the RunOrganizer will appear. Within that figure, right-click the list view drop down and choose 'Property Inspector'. In the box that pops up, scroll down to the 'String' property. Click the icon next to the word 'String' to open up the list of views. Add your own. Then click 'OK'. Next, save the RunOrganizer.fig. Now the drop down has been updated. Next, you must add the code to the script file for the RunOrganizer. Open the RunOrganizer.m file. Within the `list_view_Callback` function, there is a `switch` statement that contains a `case` for each value in the

view drop down. Add a new `case` statement for the new view name. It must be spelled and capitalized exactly in the same way as the value you added to the drop down. Then, add a call to searchRuns that filters the runs according to the criteria of interest. The syntax for the searchRuns function is as follows `searchRuns(fieldname,fieldvalue,isnumber,searchstyle)`. For example, if you want to search for all runs with a Scale smaller than 500, you would enter the following: `idx=searchRuns('Scale',500,1,'<')`. This tells MATLAB to search through the `Scale` property of each simulation record, and find those with a value less than 500. Since the field we are searching, Scale, is a number, enter 1 as the third argument. If the field were a string, you would enter 0 there instead.

How to connect to new remote machines To connect to new remote machines, you must do two things. First, add the machine to the list by going to Settings > Machines. Make sure the update the queue information for the machine as well. Second, add a new jobscript m-file for that machine, so that it prints out the correct style of jobscript to be submitted to the batch queue when runs are submitted. This stuff may be covered in a different section...

How to edit SimRun data directly Each SimRun record is stored in an array of SimRun records, called the RunArray. The RunArray is a global variable within the RunOrganizer. Therefore, it can be accessed from the MATLAB workspace by simply making it a global variable in there as well. At the command line, enter:

```
» global RunArray
```

Then, the RunArray variable is available in the desktop. Enter RunArray at the command line to see a list of all the properties of the SimRun class. Or, enter RunArray(num) where num is the number of a SimRun of your choice to see the specific property values of that SimRun. You can also edit property values from here. Make sure to save the RunArray or do something in the RunOrganizer that will trigger a save event, such as batch uploading runs, uploading a single run, deleting a run, saving a newly designed run, submitting a run, setting an error value on a run, setting the comments on a run, or setting the groups a run is in, so that your changes are saved. To save the RunArray directly (not recommended... just do something in RunOrganizer). To access the most recently created SimRun, use the `end` indexing trick in MATLAB; just enter `» RunArray(end)` to access it. To access a particular run, select that run in the list view, then choose the 'workspace variables' output. The `ind` variable gives the index into the RunArray for that particular run. Note that backing up and restoring data from back up are covered in a separate section.

How to edit other settings `save data/MyOrganizer.mat myoutputs savedfigs machines myerrors groups`

Error List To add new entries to the list of possible errors, click the `Settings` menu and choose `Error List`. A window will pop up, listing all the errors currently available in the RunOrganizer. To add a new error, click the `Add Line` button. Then, in the blank line in the table, add the error category, phrase, and description. Once finished, click the `Save` button to store the new error list. Then close the error list window. The RunOrganizer will automatically update with the new error list.

Machine List The RunOrganizer can submit jobs to and pull results from remote supercomputers if data is provided about how to connect to the remote machine. To add a new remote machine to the list of possible machines, click the `Settings` menu and choose `Machines`. A window will appear that lists all the remote machines currently stored in the RunOrganizer. Add the information about the new machine and then click the `Save` button. Then, the new machine should appear as an option in the drop down list above the second table. Choose that machine from the list. Then, add the data about the possible queues available for that machine, including the maximum run time and maximum numbers of cores that can be requested. Click the `Save` button again. Next, you will need to add a jobscript file that specifies how to create a job script to submit to the machine when you want to run a simulation on it.

40: Describe how to write a jobscript file

Groups List

Outputs List The possible outputs that can be produced from the results can also be changed. However, this is covered in the next section which discusses outputs in detail.

41: write about adding to the groups list

Outputs

- how the naming of the outputs work - producing figures or image files - how only the possible outputs for that run are shown

There is a MATLAB structure that saves information about the possible outputs. It includes fields that populate the list of outputs in the RunOrganizer, fields with conditions for runs that must be satisfied for an output type to be produced, and the output file to execute. When adding a new output to the RunOrganizer, you must put the file in the `outputtypes` folder within the RunOrganizer directory and also add an entry to the MATLAB struct. Then, the next time the RunOrganizer is launched, that output will be available for the runs that meet its conditions.

- lists some of the outputs available and explains them

- **Spike Raster**
- **Axonal Dist.** This output can only be computed when the positions of each cell have been calculated and the detailed list of connections (including the gid of every presynaptic and postsynaptic cell for each connection) is

available . When those two conditions are met, this analysis is listed as an option in the output list of the RunOrganizer. This file computes the distance between each pair of connections. It then graphs the distribution of the distances as a function of presynaptic cell type, to give the effective axonal distribution of that type.

- **Conn. Matrix** This output is available as long as at least a summary of the total connections between each type of cell is available (numcons.dat), which is printed out if the PrintConnsSummary is set to 1. It gives the total number of connections between every combination of presynaptic cell type and postsynaptic cell type.

- **Histogram** This output just requires the spikeraster (and like the other outputs, a list of gid ranges per cell type). It analyzes the power of oscillations in the model. It takes an argument in the form of an oscillation period. It takes the FFT of each cell type spiking activity in the model and reports the power and frequency of the dominant frequency as well as the power of the oscillation at the frequency of interest. It not only displays the FFT spectrum but also the histogram of spiking activity double plotted over two periods of the frequency of interest, per cell type. It also shows an image of experimental results for theta oscillations that can be used for comparison.

- **Traces** This output is available as long as there are single cell voltage traces for each cell type in the model (at least one per cell type) . Then, this function displays an intracellular voltage trace from one cell instance for each type .

- **Cell Ranges** This output is always available as long as the output files cellType.dat and spikeraster.dat are available; they are automatically printed with each successful run. This output prints a table to the MATLAB desktop. For each cell type in the model, the table gives index of the cell type, the gid range of the cell type, the number of cells, the number of cells that didn't spike during the simulation, as well as the minimum, maximum, and standard deviation for spikes per cell of that type, as well as all spikes by all cells of that type.

- **Spikes** This output type requires a spikeraster figure to be showing. Upon selecting this output, a cursor will appear on the spikeraster for you to click twice, once marking the beginning of a duration of interest and once marking the end. For each cell type in the model, it will show the number of spikes made by that type and the number of cells of that type that contributed to the spiking during that time. Then it also gives the gids of the first and last cells to spike during that time.

- **Relative Timing** This output type requires a spikeraster figure to be showing. It measures the relative starting time of each cell type if a pattern of pulses is visible in the spikeraster. Once this output is selected, a dialog

42: file names

43: perhaps remove this in favor of the conn div?

44: what about something similar for gamma?

45: filename

46: make argument that allows to specify how many

47: perhaps give fraction of cells of that type that spiked and which cells spiked most and how much they spiked

48: add a requirement of a spikeraster

box will appear for you to enter the number of pulses in the spikeraster that you want to analyze. Then a cursor will appear over the spikeraster for you to select the pulses. Both the start and the end of each pulse must be defined by clicking. After all pulses have been defined, then the program will compute the average lag time of each type of cell to start the pulse, relative to the pyramidal cell starting the pulse. . Since not all cell types necessarily participate in each pulse, it is noted when a certain cell type misses some pulses. The standard deviation of the start time is also plotted. Within the MATLAB desktop, further results are reported. If a certain subset of cells is participating in the pulse each time, that can be determined from the table in the desktop, where it gives the number of cells participating in (all? then why is there a std) all pulses.

49: what about avgs of avg spike time per pulse or max spike time?

50: test this, some of the results don't make sense to me. use NASA_CondA_08

- **FFT** For each cell type, the FFT of the spiking activity of all cells of that type is calculated and plotted. The power and frequency of the peak overall are noted, as well as the peak within the theta range of $4 - 12$ Hz

- **Cell Inputs** This gives a connection matrix (convergence only) for a single cell. It is only available for those cells for which detailed outputs were obtained (the traced cells for which the detailed connection file was written out)

- **Trace Stats** For those cell instances whose voltages were traced during the simulation, the traces are averaged and presented here, one per cell type. The average, plus or minus the standard deviation, and the maximum and minimum voltages for each time point are graphed.

- **Workspace Variables** This option sends several variables to the MATLAB desktop workspace so that you can manipulate them ad hoc. The variables sent include: a spikeraster (times and gids of spiking cells), a cells struct that contains the names, technical names, numbers, and gid ranges of all cell types, a numcons matrix that gives the host processor, the pre and postsynaptic cell type (indices correspond to those given in the cells struct) and the number of connections between those types on that processor, and an ind variable that gives the RunArray index of the run, and a copy of the SimRun record called myrun.

- **ppstim vectors** This view ...

- **Conv. Div.** This view produces an output similar to the connection matrix, except with more data. It gives the total number of connections between each combination of presynaptic and postsynaptic cell type, as well as the weight of the connections, the divergence of each presynaptic cell type and the convergence onto each postsynaptic cell type. It also includes the number of each cell type where relevant.

- **Trace Inputs** Pick a gid out of those that have the intracellular trace and get two graphs. One is more of a summary. It gives the intracellular

voltage trace and histograms of the total excitatory inputs and total inhibitory inputs. It also gives the convergence onto the cell, in terms of connections, synapses per connection, and spikes per each cell type, as well as the kinetics and weight of each synapse type. A second figure gives the histogram of inputs for each presynaptic cell type, plotted over the intracellular voltage trace.

- **Single Cell Spike Train** The single cell spike train output is available for those cells that have an intracellular voltage trace . It plots their spike train and also performs an FFT on the spiking activity and, if available, the membrane potential trace.

  - Lists outputs produced from that simulation on the left side

51: print activity histograms on top of trace

52: should be for others, too, w/o MP fft

Create New Outputs

- How to create new output scripts, where to save them, how to update the output struct and add conditions

Error tracking

Error tracking is also important. The RunOrganizer allows tracking of simulations that ended in error. One of the most noticable symptoms of an error is that the output files from the simulation are not produced. This becomes apparent when the RunOrganizer is uploading an executed run. The run receipt is usually still printed out, so the execution date of the run is still filled out. This means that the run still shows up in the 'Executed Runs' view, though there are no results for it. For runs like these, they can be easily spotted using the view 'Find Errors', which displays runs that have an Execution Date but no information about total spikes or total number of connections and cells. When a run is selected, there is a dropdown list of errors that can be picked from to describe what went wrong. New errors can be added to the list, as described in section ??. An error can be picked from this list and it will be added to the SimRun record. There are various categories of error, RunTime, Programming ... The possible errors are:

RunTime

- **Error type**

Programming Time

- **Error type**

Sometimes errors can occur even before the run receipt is written out. In this case, the run appears as though it hasn't been executed, because no execution data is uploaded. For these runs,

53: what to do?

54: if error happened before run receipt, then no execution date, then even after you add error to RunOrganizer, it still shows up in the Not Ran list. fix!

55: generally, pull the jobscript file into the results folder when done.

AutoRig

The AutoRig tool (Fig. 4) simulates ephys done to characterize cells and channels. It allows you to compare model and experimental data by producing analogous data from the model.

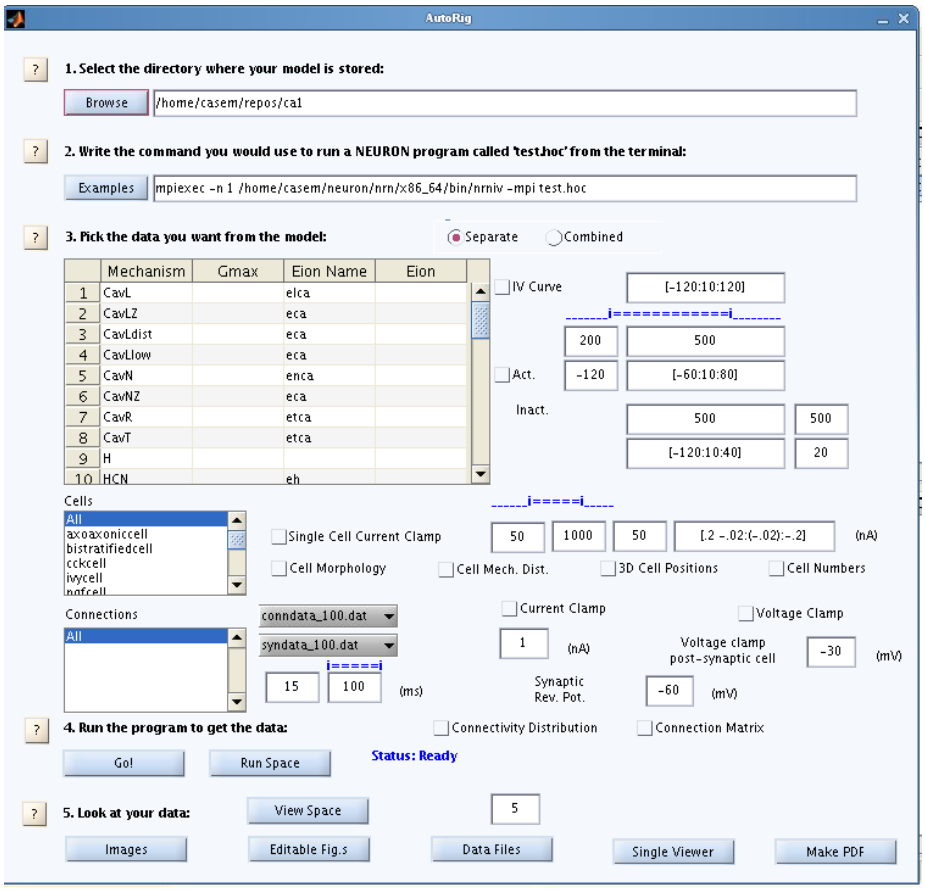


Figure 4: Screenshot of the AutoRig tool.

- purpose of the tool - works with code versioning in that it tells you which version it is using. It also records the version used in the runreceipt associated with each AutoRig run. Additionally, it prints out any changes made to the code in the separate version status and version difference files.

At the top of the AutoRig, push the button to specify the model directory, which will then display in the field next to the button, along with the current Mercurial version of that repository. Below that field, there is a field for entering in the command to run NEURON programs locally. Write the execution line exactly as if you were submitting a file called `test.hoc` to be executed. The

AutoRig program will then substitute in the necessary commands to run its own NEURON code.

With each execution of the AutoRig, the results will be written out into a separate results file. Within that file, a run receipt will also be written that gives the code version and the parameters used in the run.

Below these fields is a table listing all the channels. To the right of the table are checkboxes to specify which types of outputs to produce for the channels of interest. The channels of interest are specified by entering values in the table for their `Gmax` and `Eion` columns. The `Gmax` of the channel is the maximum conductance possible for that channel. In model code, it is sometimes called `gbar` and in text it is sometimes called g. This is in units of mho/cm^2. Then `Eion` gives the reversal potential of the current through this ion channel in mV. Once these values are entered, then if an ion channel simulation type is checked, anything checked will be run for this channel type. The ion channels are characterized by inserting them into a blank soma. The parameters used to characterize the soma are initially set to:

- **Soma Diameter** 16.8 μm
- **Ra** 210 ohm*cm
- **cm** 1 uF/cm^2
- **Celsius** 37^o

They can be easily changed by going to the Settings menu and choosing Ion Channel Settings, then updating them in the dialog box that appears.

Channels - lists all channels in model (mod files beginning with ch_) - useful for ion specific and nonspecific channels - running channel graphs: act/inact, IV curve

- **IV curve** This shows how the current through the channel depends on the voltage across the membrane.
- **Act./Inact.** This shows how the levels of activation and inactivation of the channel depend on the voltage

56: explain the measuring protocol

57: using tail currents? not steady state right

58: add something for time course of inact, recovery?

- the logic used to produce the graphs (bio logic)

- what about the model code is necessary for this to work: In order for this section of the AutoRig to work right, there are a number of requirements for code syntax. The channel definition mod file names must begin with `ch_`. Other mod files that do not define channels must not use that prefix. Further, the suffix defined within the mod file must correspond to the channel name. For example, a sodium channel defined in the file `ch_Nav.mod` must have the `SUFFIX` set to `ch_Nav`. Further, within the channel definition, the maximum conductance must be a `RANGE` variable with the name of `gmax`. There must be a

range variable called `myi` which calculates the total current from that mechanism .

More things to discuss about channels: 1. Defining ions 2. alpha and beta parameters 3. initialization and step-wise equations 4. Making values within NMODL available to hoc (and recording in hoc) 5. Specific tutorials available for these NMODL code examples on our website 6. Further information is available in the NEURON book (and other peoples' online tutorials)

59: should we keep myi?

Single Cell Recordings - lists all cells in model, taking the list from the files within the `cells` folder of the model repository, those files that begin with the prefix `class_`.

- The cell can be current clamped to a variety of values for a set length of time. The membrane potential of the cell will be recorded, as will the currents through all the ion channel mechanisms present in the cell. These traces will be displayed in a figure resembling experimental results.

- what about the model code is necessary for this to work

Morphology

60: Doesn't work

cell mech dists This option graphs the values of each subcellular mechanism as a function of location within the cell. For each cell selected, a separate figure is created for the axon, soma, and dendritic groups (apical or basal). Within that figure(s), the value of each mechanism is graphed as a function of distance from the 0 end of the soma. This includes the values of cm, Ra, and the g_max values of any channels as well.

3D

61: Doesn't work

cell numbers

62: Doesn't work

Paired Cell Recordings The connections between cells can be characterized using paired cell recordings. In the model, this means triggering the synapse associated with a particular presynaptic cell and measuring the response in the postsynaptic cell. The AutoRig can do both PSP (current clamp) and PSC (voltage clamp) measurements for paired recordings. It generally measures the recording at the soma of the postsynaptic cell. It allows you to specify the clamp conditions, the reversal potential of the synapse and, in the case of voltage clamp, the junction potential as well.

The AutoRig allows you to specify the number of synapses per connection, the weight and kinetics of each synapse type. This is done by specifying the ConnData (weight, number of synapses) and SynData (kinetics) sets used to set up the paired connection that you will measure. To form the connection, the AutoRig will randomly choose the specified number of synapses from the available synapses for that cell type (the possible locations of which are specified in the postsynaptic cell template). The AutoRig will perform 10 different

paired recordings with 10 different connection configurations, meaning 10 different combinations of synaptic connections.

To use the paired recording section, you first specify the ConnData set to use. This will cause the list of available pairs to be populated. You can record multiple pairs at once, but they must all have the same reversal and junction potential. Even pairs from artificial cells to real cells can be recorded.

The average kinetics and amplitude of the 10 connection configurations will be calulated and displayed along with the graph of the postsynaptic response, along with the reversal potential and junction potential and holding potential (or current) used.

63: ensure all outputs listed

For this part of the model to work, the code must have ConnDatas, SynDatas, and must use a synapse definition file that has an 'e' range variable that can be set to the reversal potential of the synapse.

Our model uses a two-state kinetic scheme for all of its synapses; the scheme has a time constant to specify the rise kinetics of the synaptic conductance and another time constant to specify the decay time frame of the conductance. The weight, set in the ConnData and used by the model within the connection formation process, is in μS and sets the maximum conductance of the synapse. The output of the synapse is the current in nA. The kinetics and weight are set on a per-synapse level, so it is important to view the overall kinetics and weight of the combined effect of all synapses to make sure that is what matches the experimental data. The synapses summate.

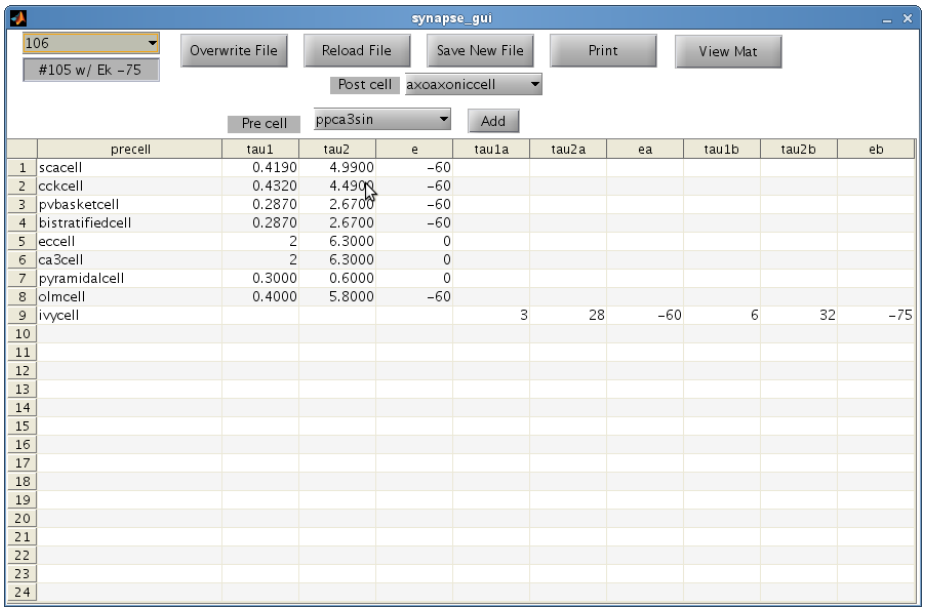
SynData

The synapse kinetics for all possible connections are specified in the cell definition file for the postsynaptic cell. However, we found that it can occasionally be convenient to view the synapse kinetics for multiple postsynaptic cells at once, and to edit them apart from the rest of the cell definition. Therefore, we also store the synapse kinetics in a separate file. The `synapses` folder contains files of synapse data. Each file defines all the synapse kinetics parameters for all connections in the model; different files contain different sets of synapse data. The synapse data from these files can be displayed in the `SynData` GUI. Within the GUI, files can be displayed, altered, and saved. Additionally, files can be 'printed'. Printing a file means inserting the relevant code for those synapse kinetics into each cell definition file.

Explain the `cells` and `cellframes` folders here. How the `cells` folder is ignored by Mercurial, but it contains the files used by the model to define the cells. All changes should be made to the `cellframes` files, which are tracked by Mercurial. After making changes to those files (or updating the Mercurial version), then the synapses should be printed again, even if the synapse set didn't change. This will refresh the definition within the `cells` folder.

Buttons

- **Overwrite File** Upon clicking this button, any changes made in the table are written to the current file (the file displayed in the drop down list).



	precell	tau1	tau2	e	tau1a	tau2a	ea	tau1b	tau2b	eb
1	scacell	0.4190	4.9900	-60						
2	cckcell	0.4320	4.4900	-60						
3	pvbasketcell	0.2870	2.6700	-60						
4	bistratifiedcell	0.2870	2.6700	-60						
5	eccell	2	6.3000	0						
6	ca3cell	2	6.3000	0						
7	pyramidalcell	0.3000	0.6000	0						
8	olmcell	0.4000	5.8000	-60						
9	ivycell				3	28	-60	6	32	-75
10										
11										
12										
13										
14										
15										
16										
17										
18										
19										
20										
21										
22										
23										
24										

Figure 5: Screenshot of the SynData tool.

- **Reload File** Upon clicking this file, the table is refreshed with the data currently in the file, so any changes made to the table without saving the file are lost.

- **Save New File** When this button is clicked, a new file is created with the values currently in the table.

- **Print** When this button is clicked, code is generated that defines the synapses for each postsynaptic cell and it is inserted into the synapse definition function (proc?) of the cell definition template for each postsynaptic cell type.

- **View Mat** This displays a table of all possible presynaptic and postsynaptic cell types. The cells give the total number of synapses defined for each combination.

- **Post cell** This is a dropdown list that specifies which postsynaptic cell is currently being displayed. Only the synapses for which that cell type is postsynaptic are displayed. All cells within the model should be listed in this dropdown list. (all cells with the class file in the cells folder?)

- **Pre cell** This dropdown list includes all possible cell types in the model. If you want to add a synapse for a particular presynaptic cell type, you can select that type here and then press the `Add` button

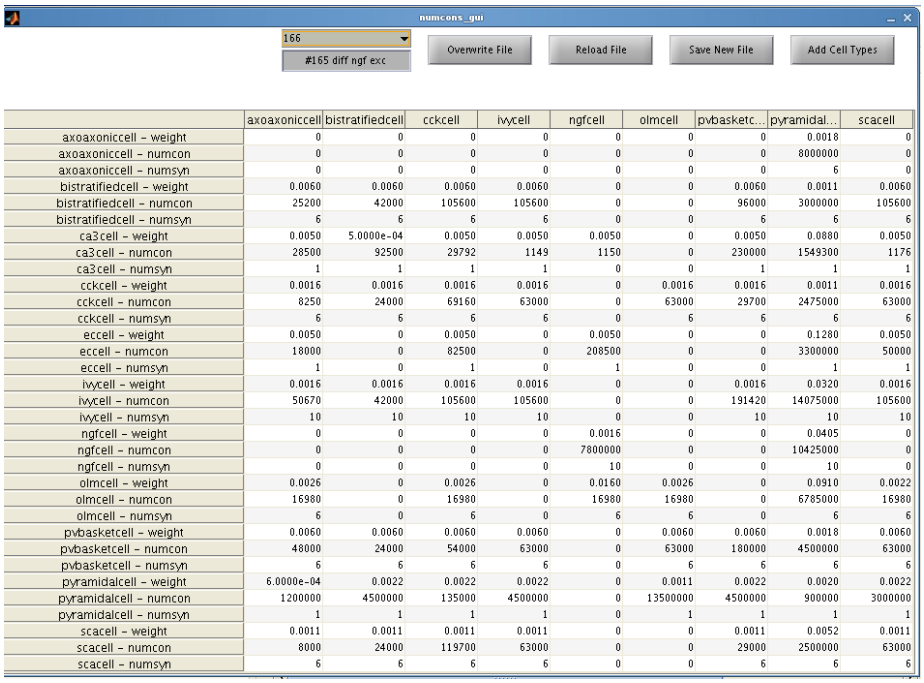
- **Add** This button will add a new entry to the current table for the selected presynaptic cell type. It saves you the time of typing in the presynaptic cell type and also ensures that the cell type is spelled correctly.

ConnData

The `numcons_gui` GUI (Fig. 6) allows you to view and specify connectivity sets, which define the number and strength of connections between all the cells of the model. For each possible presynaptic and postsynaptic cell type, the total number of connections between all cells of those types is specified, as well as the synapse strength (maximum synaptic conductance) in μS. At the top of the table are buttons to specify the connection sets. There is a drop down list of all existing connection set files (conndata_[num].dat files). When you pick a particular connection file, it loads into the table. It also updates the caption below the drop down, which displays the descriptive comment associated with that connection set. There are the following buttons as well:

- **Overwrite File** Upon clicking this button, any changes made in the table are written to the current file (the file displayed in the drop down list).
- **Reload File** Upon clicking this file, the table is refreshed with the data currently in the file, so any changes made to the table without saving the file are lost.
- **Save New File** When this button is clicked, a new file is created with the values currently in the table
- **Add Cell Types** ~~. If the file currently displayed doesn't include all cell~~ types (i.e., some cell types have been added since then), this button will append the previously excluded cell types to the table.

64: this button may not work



	axoaxoniccell	bistratifiedcell	cckcell	ivycell	ngfcell	olmcell	pvbasketc...	pyramidal...	scacell
axoaxoniccell – weight	0	0	0	0	0	0	0	0.0018	0
axoaxoniccell – numcon	0	0	0	0	0	0	0	8000000	0
axoaxoniccell – numsyn	0	0	0	0	0	0	0	6	0
bistratifiedcell – weight	0.0060	0.0060	0.0060	0.0060	0	0	0.0060	0.0011	0.0060
bistratifiedcell – numcon	25200	42000	105600	105600	0	0	96000	3000000	105600
bistratifiedcell – numsyn	6	6	6	6	0	0	6	6	6
ca3cell – weight	0.0050	5.0000e-04	0.0050	0.0050	0.0050	0	0.0050	0.0880	0.0050
ca3cell – numcon	28500	92500	29792	1149	1150	0	230000	1549300	1176
ca3cell – numsyn	1	1	1	1	0	0	1	1	1
cckcell – weight	0.0016	0.0016	0.0016	0.0016	0	0.0016	0.0016	0.0011	0.0016
cckcell – numcon	8250	24000	69160	63000	0	63000	29700	2475000	63000
cckcell – numsyn	6	6	6	6	0	6	6	6	6
eccell – weight	0.0050	0	0.0050	0	0.0050	0	0	0.1280	0.0050
eccell – numcon	18000	0	82500	0	208500	0	0	3300000	50000
eccell – numsyn	1	0	1	0	1	0	0	1	1
ivycell – weight	0.0016	0.0016	0.0016	0.0016	0	0	0.0016	0.0320	0.0016
ivycell – numcon	50670	42000	105600	105600	0	0	191420	14075000	105600
ivycell – numsyn	10	10	10	10	0	0	10	10	10
ngfcell – weight	0	0	0	0	0.0016	0	0	0.0405	0
ngfcell – numcon	0	0	0	0	7800000	0	0	10425000	0
ngfcell – numsyn	0	0	0	0	10	0	0	10	0
olmcell – weight	0.0026	0	0.0026	0	0.0160	0.0026	0	0.0910	0.0022
olmcell – numcon	16980	0	16980	0	16980	16980	0	6785000	16980
olmcell – numsyn	6	0	6	0	6	6	0	6	6
pvbasketcell – weight	0.0060	0.0060	0.0060	0.0060	0	0.0060	0.0060	0.0018	0.0060
pvbasketcell – numcon	48000	24000	54000	63000	0	63000	180000	4500000	63000
pvbasketcell – numsyn	6	6	6	6	0	6	6	6	6
pyramidalcell – weight	6.0000e-04	0.0022	0.0022	0.0022	0	0.0011	0.0022	0.0020	0.0022
pyramidalcell – numcon	1200000	4500000	135000	4500000	0	13500000	4500000	900000	3000000
pyramidalcell – numsyn	1	1	1	1	0	1	1	1	1
scacell – weight	0.0011	0.0011	0.0011	0.0011	0	0	0.0011	0.0052	0.0011
scacell – numcon	8000	24000	119700	63000	0	0	29000	2500000	63000
scacell – numsyn	6	6	6	6	0	0	6	6	6

Figure 6: Screenshot of the ConnData tool.

2 Results

2.1 Subcellular Mechanisms

2.1.1 Ion Channel Graphs

2.2 Cells

2.2.1 Distribution of Ion Channels in Cells

2.2.2 Cell Morphology

2.2.3 Single Cell Traces

2.2.4 Single Cell fI curves

2.2.5 Table of Single Cell Properties

2.2.6 Graph of Cell Numbers

2.3 Connections

2.3.1 Paired Cell Recordings

2.4 Network Structural

2.4.1 Graph of Cell Numbers

2.4.2 ConvDiv Table

2.4.3 3D Model Pic

2.4.4 Axonal Divergence

2.5 Network Functional

2.5.1 Activity Inputs to a particular cell

2.5.2 Record a cell

2.5.3 Spike Raster for levels of sprouting and sclerosis

3 Discussion

a. Summarize how this will be useful, can generate figures for use in publications b. Publically available tools i. AutoRig on SimToolsDB (or link to MATLAB Central - compiled and m-scripts) ii. NEURON and NMODL code on ModelDB iii. Tutorials on lab website c. Next steps i. Produce this in another format, perhaps Python? ii. Users can customize the AutoRig to produce additional validation 'experiments' such as: .... (ion channel dependence on pH, etc) iii. Be mindful of structure of NEURON models that you produce

Acknowledgements

Appendix

- talk about how to make a jobscript m-file for a new remote machine

References

Arne Babenhauserheide, Steve Losh, and David Soria Parra. Mercurial scm. http://mercurial.selenic.com/, Jan., 2013.

N.T. Carnevale and M.L. Hines. *The NEURON Book*. Cambridge University Press, Cambridge, UK, 2006.

M.L. Hines, T. Morse, M. Migliore, N.T. Carnevale, and G.M. Shepherd. Modeldb: A database to support computational neuroscience. *J. Comput. Neurosci.*, 17:7–11, Jul-Aug 2004.

David Sterratt, Bruce Graham, Andrew Gillies, and David Willshaw. *Principles of Computational Modelling in Neuroscience*. Cambridge University Press, Cambridge, UK, 2011.