Published in final edited form as:
Neuroinformatics. 2006 ; 4(2): 163–176. doi:10.1385/NI:4:2:163.

Neural query system: data-mining from within the NEURON simulator

William W. Lytton
Depts. of Physiology, Pharmacology and Neurology, SUNY Downstate Medical Center, Brooklyn, NY 11203, Dept. of Electrical Engineering, Polytechnic University, Brooklyn, NY 11201, July 1, 2005

Abstract

We have developed a simulation tool within the NEURON simulator to assist in organization, verification and analysis of simulations. This tool, denominated Neural Query System (NQS), provides a relational database system, a query function based on the SELECT function of Structured Query Language (SQL), and data-mining tools. We show how NQS can be used to organize, manage, verify and visualize parameters for both single cell and network simulations. We demonstrate an additional use of NQS to organize simulation output and relate outputs to parameters in a network model. The NQS software package is available at http://senselab.med.yale.edu/senselab/SimToolDB.

Introduction

Neural simulations are complex and difficult to organize. Large single neuron compartmental model simulations with active dendrites can require tens of thousands of parameters. Although the computer power is not yet available to organize these massive single neuron models into large networks, networks of several thousands of cells can be put together using simpler unit models, each with their own still-large parameter set. In networks, additional parameters arise from the perhaps millions of synapses with different weights, time constants, delays and thresholds. These massive parameter sets pose problems of parameter organization and verification.

Simulation is an experimental endeavor that produces large amounts of data that has to be organized, analyzed and correlated with associated parameter sets. In addition to being a producer of simulation data, neural simulation is also a consumer of experimental data. Here there is a need to compare data sets in order to demonstrate the adequacy of a particular simulation.

Both on the input side (parameter management) and on the output side (simulation results), there is a large amount of numerical data which is*de facto* stored in some sort of data structure, whether by design or happenstance. With the Neural Query System (NQS), we utilize relational rectangular tables to store this information. The use of a single tool for both parameters and data is parsimonious since data visualization tasks are comparable for both, and since these must be related to one another and to experimental data in the context of a full modeling project. The use of rectangular tables, inter-table relations, and a query language is a common framework that is well understood, familiar and easy to use. Where rectangular arrays are not well suited to a particular neural problems, NQS allows the array to be extended by providing object pointers. For example, in the case of the dendritic tree (Example 1 below), a row giving parameter entries for a particular compartment in a compartmental model of the tree can have a column containing a pointer to that compartment in the simulator's tree structure.

While parameter sets for simple models can easily be verified with a few print statements, those for very large models are more difficult to assess. A not-uncommon modeling pitfall is to complete a series of modeling runs only to look back at the parameter set and discover that some parameter, left over from a previous modeling effort, was set in a way that is not consistent with the current project. For this reason it is valuable to be able to assess parameters interactively, serially and graphically. NQS facilitates directed queries and visualization, the basic data-mining process, to evaluate parameters. Additionally, NQS makes it easy to store parameter sets that can then later be explored together with simulation results when looking for parameter/dynamics correlations.

NQS functionality could of course be offloaded to a standard database program. However, most database programs are not optimized for numerical processing and do not directly offer the sophisticated numerical analysis tools available in NEURON. Alternatively, a numerical analysis programs such as MATLAB could be used. MATLAB offers a*Database Toolbox* that can be used to interact with a standard database but is not itself a database substitute. Since it works with a standard database, it uses the basic Structured Query Language (SQL) language without numerical enhancements. In either of these cases, separate databasing of parameters would require writing additional code, possibly in a third scripting language, in order to coordinate fetching and putting and interaction with the simulator. Finally, maintaining on-line data storage within the simulator will permit more sophisticated applications in the future. For example, simulation results can be immediately compared to experimental data and optimization performed using tools such as NEURON's "multiple run fitter" or other simulation fitting tools such as are available in the GENESIS simulator (Bower and Beeman 1998).

NQS is compiled into the NEURON simulator as a module. The software is available at http://senselab.med.yale.edu/senselab/SimToolDB. We illustrate the use of NQS to analyze existing single neuron simulations (Example 1), to assist in development of network models (Example 2) and to relate simulation output to parameters (Example 3).

NQS functionality

NQS provides access to a variety of numerical tools that can be used for data-mining. Many of these are vector (array) manipulation tools built in to NEURON. These vector functions permit convolution, numerical differentiation and integration, and basic statistics. Additional data-mining tools have been added on by compiling C-language code that can be directly applied to the numerical vectors used as columns for a table (see Appendix C). Vector-oriented C-language code is readily available from a variety of sources (Galassi *et al*. 2003; Press *et al*. 1992). Such code can be compiled into NEURON after adding brief linking headers.

NQS handles basic databasing functionality including: 1. creating tables; 2. inserting, deleting and altering tuples; 3. data queries. More sophisticated databasing functionality such as indexing and transaction protection are not yet implemented. Databasing commands in NQS provide 1. selection of specified data with both numerical and limited string criteria ; 2. numerical sorting; 3. printing of data-slices by column and row designators; 4. line, bar and scatter graphs; 5. import and export of data in columnar format; 6. symbolic spreadsheet functionality; 7. iterators over data subsets or over an entire table; 8. relational selections using criteria across related tables; 9. mapping of user specified functions onto particular columns. The important NQS commands are listed in Appendix A.

NQS is not a full database management system since it is meant primarily as single-user system that does not have to handle problems of data-access control and data security. It also does not presently have the indexing and hashing capabilities of a full database management system, though these may be added in the future. NQS provides some spreadsheet functionality.

However, several features characterize it as a database rather than a spreadsheet system: the presence of a query language; the ability to handle many thousands of records; data structuring; handling of non-numeric data; capacity for relational organization across database tables.

Among basic databasing functions, querying is the most complex. A query language, although often regarded as a database component and thereby denigrated as a data-mining tool, is a critical aspect of data-mining. SQL, because of its commercial antecedents, is less numerically oriented than is desirable for scientific query. The NQS select command is designed to focus on numerical comparisons. Due to the importance of geometric information in neuroscience, inclusion of geometric criteria will be an additional feature that would be desirable in further development of NQS.

The NQS select() command is similar to the commands related to the WHERE and HAVING sub-functions of SQL's SELECT. NQS syntax naturally differs from that of SQL as it must follow the syntax of NEURON's hoc language. An NQS database table is a template in hoc. A new table is created by having an object pointer that is then made to point to a new instance of an NQS:

```
tab1 = new NQS("COLA", "COLB",…).
```

This would create an empty database table named "tab1" with column names given by the quoted strings provided. (tab1 is a pointer to an NQS instance.) Neuron uses the C/C++ dot-notation idiom:*e.g.*, tab1.select(…) does a select command on the contents of table tab1.

Having created the table it is then necessary to fill it with data. This typically requires some programming in order to either provide the data row-by-row using tab1.append() or column by column with tab1.setcol(). The data manipulation required for filling tables will be discussed in the examples below.

The NQS select() command takes any number of arguments in sets. Each set consists of a column name, a comparative operator such as '<' or '==' (listed in Appendix B) and one or two arguments depending on the operator. Multiple criteria in a single select() statement are handled with an implicit AND. A flag can be set to use OR on the clauses. A select() command that begins with a "!" will return the complement of the selected rows. A command can also begin with "&& " or "||" to return the union or respectively intersection of the selected rows with previously selected rows (*cf.* SQL UNION, INTERSECT, MINUS subcommands). Relational databasing (inner join) is done using the vector oriented EQW operator which references values previously selected from a column of the same name in a separate table. Relational databasing is therefore done serially: first selecting from table #1 and then ANDing the results with a separate select command on table #2.

Although the NQS select() was not designed to replicate the agglutinative syntax of SQL SELECT, much of this functionality can be replicated by serial application of NQS's select(), sort() and stat() functions.

Unlike SQL's SELECT, whose selected values are printed by default, the NQS selection simply stores tuples for further manipulation: typically printing, numerical operations or graphing. Following a select, the user is by default accessing the selected tuples when calling any subsequent commands (*e.g.*, print, sort, etc.). The tog command toggles back-and-forth between accessing the entire table and the most recently selected component. As noted above, multiple select commands can be used to gradually focus on data subsets. Alternatively, selected records can be exported as a new separate table for further exploration.

The full package consists of over 50 commands. Major commands are shown in Appendix A. A complete listing is given in the manual provided with the NQS package.

Example 1: NQS for compartment cell model analysis

Mapping onto a relational database involves a remapping from the native organization of the data or of the data collection method into a rectangular form. In the case of a dendritic tree, mapping onto columns and rows obscures the tree relationship which is then implicit in columns indicating section parent and branch order. We have used NQS to begin to analyze some of the models available in the ModelDB database at Yale (http://senselab.med.yale.edu/senselab/ModelDB; Davison *et al*. 2004; Hines *et al*. 2004). This application domain is similar to the neuroanatomical database tools developed by Ascoli *et al*. and to NEURON's "modelview" tool (Ascoli 2002; Ascoli *et al*. 2001a,b).

In practice, the creation of the database from morphology and conductance values is almost fully automated. The program first parses the tree to pick up morphological relationships. On another pass, it scans each compartment for the presence at any conductances available in that particular model, using a regular expression which looks for a "g" or "p" followed by a "max" or "bar." This picks up names such as gbar_naf or pcamaxT. User intervention is only called for where a model in ModelDB uses a variant name for a particular maximal conductance value that is not picked up by the standard regular expression. In the following, the construction of the databases is described step by step to illustrate the features of NQS.

Column headings are typically defined when creating a database table. NEURON's handling of the compartmental model of the dendritic tree is unusual in that it organizes the tree into unbranched *sections* which can then be further subdivided into individual *segments* which are the compartments of the compartmental model. (The rationale is that spatial discretization, number of compartments, can be easily changed without altering the section naming that follows the underlying topology.) Because some attributes of a model neuron belong to the segment while others belong to the section, it makes sense to create separate section and segment tables, SC and SG respectively. The tables are initially created with the following commands, which utilize hoc's object-oriented syntax to create two new objects, each a table:

```
SC = new NQS("NAME","PARENT","NCHILD", "NSEG", "ORDER", "CODE")

SG = new NQS("NAME", "X", "Y", "Z", "DIST", "DIAM", "SEG#")
```

The NAME and PARENT columns will be strings and must be declared as such with a strdec () command. This declaration allows queries on these columns to utilize regular-expression and string-match operators. The other columns will all take numerical values. For SC, NCHILD and NSEG will be used to store the number of child branches and number of segments respectively; ORDER for branch order and CODE reserved for a code to indicate type of section (eg soma, axon, proximal dendrite, etc.). For SG, X, Y, Z will give coordinates, DIAM will give segment diameter and SEG# is the segment counter within a section. Section NAME is used in both tables and can be used for relational joins.

Now that the tables have been created, they must be populated with data. This can be done either on a row by row basis, by setting individual columns, or by setting *row, column* cells individually. In the present case, NEURON's forall function is used to go through the dendritic tree. The NQS append() command will add rows corresponding to individual sections or segments. In this way, we can fill in all but the ORDER and CODE columns, which will be set subsequently. For example, the first row for SC might be added with:

```
SC.append("soma", "NONE",4,1,0,0)

(NAME: soma; PARENT: NONE; NCHILD: 4; NSEG: 1; ORDER: 0; CODE: 0)
```

The CODE column can then be set by utilizing rules to identify certain compartments by diameters (*e.g.*, a soma is large and an axon small), by location (*e.g.*, the Y location in an oriented cell indicates the cortical layer) or by other characteristics (*e.g.*, terminal sections have

no children). The ORDER column can be filled in iteratively by setting ORDER to 0 for the soma and then determining one or two higher-order child sections and an optional continuation section at each bifurcation. Note that child section identities are not directly available but can be found by selecting for a particular section in the PARENT column.

Now that the morphology database is built, we can use the select command to explore the data. For example, we may wish to select for dendrites at a range of 200–400 microns from the soma which with diameters of greater than 2 microns and then look at their Y locations.

```
SG.select("DIST", "()",200,400, "DIAM", ">",2)
SG.pr("NAME", "Y")
```

If we wanted to know the branch order of the segments which we just selected, we would need to use the relational property to connect with the SC table.

```
SC.select("NAME",EQW,SG) // EQW relates prior select command in SG
SC.sort("ORDER")
SC.pr("NAME", "ORDER")
```

In this example, we also sorted the output by lowest-to-highest ORDER before printing it. Here, EQW is the relational operator used to pick out NAME's in SC that were previously selected in the SG table.

Selected results can be graphed instead of printed, *e.g.*, to graph distance against diameter for the selected segments:

```
SC.gr("DIST", "DIAM")
```

In addition to the models basic morphology, we add information about voltage-sensitive ion channel densities in the individual segments to our table. The segment table is augmented to include columns for maximal sodium and delayed rectifier conductances by using the resize command:

```
SG.resize("gna bar", "gkdr_bar")
```

This is done algorithmically to pick up the 5–20 different channel types present in a typical compartment model. Maximal conductance values from individual compartments are then retrieved from the underlying model to populate these new columns. This extended SG table now contains physiological model information (channel conductances) as well as anatomical model information.

The complex compartmental model contributed to ModelDB by Poirazi *et al*. (2003a,b) includes 17 different conductances scattered over 357 compartments, making it difficult to understand by code perusal. In Fig. 1, all of the conductance densities for this model are organized graphically by columns, sorted by distance from the soma with the soma at the bottom. Horizontal width at a given location indicates conductance magnitude compared to its magnitude elsewhere in the tree. We can see that many of the conductances take on only one or two values. In the full graphic, color coding is also used to indicate the absolute conductance density compared with all of the conductances.

Fig. 1 only represents the starting point for exploration of the model, being a graphical rendering of the database. We could then go through and confirm certain observations by running statistics – for example, which channels are single-valued, which two-valued and which take on a range of values. Certain features of interest that jump out can be explored further – for example, what is special about the compartments that use nahha2/khha2 compared to the compartments that use nahha old/khha old. In this context, it might be valuable to connect into

separate tables containing abbreviated kinetic information for each conductance, *e.g.*, activation and inactivation thresholds with time-constants at threshold.

Example 2: Organizing network wiring

Real brain networks feature a bewildering array of neuron types, differing in dendritic shape and temporal response properties. The densities of each constituent neuron type is variable and the wiring between populations differs. Not only are there different neuron types, there are also different connection types with different signs (excitatory and inhibitory synapses) and different time courses (*e.g.*, $GABA_A$ vs. $GABA_B$ synapses). Such complexities make it difficult to design, implement and firm the wiring diagram for a particular network. If we are designing a hybrid network of realistic and artificial cells, we add further complexity by representing some of these neurons with detailed multi-compartment models and the rest as simplified integrate-and-fire neurons.

The wiring of a simple neural network is typically represented by a connectivity matrix which gives the weight between units. For example, Fig. 2 would be represented by the following matrix:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -.5 & 0 & 0 \\ 0 & 0 & 0 & -.6 & 0 & 0 & 0 & 0 & 0 \\ .4 & -.1 & 0 & 0 & 0 & .2 & 0 & 0 & 0 \\ .5 & 0 & 0 & 0 & 0 & 0 & 0 & -.8 & 0 \\ 0 & 0 & 0 & 0 & 0 & -.2 & 0 & 0 & .7 \\ 0 & 0 & .3 & 0 & 0 & 0 & -.7 & 0 & 0 \\ 0 & 0 & 0 & 0 & .6 & 0 & 0 & .9 & 0 \\ 0 & 0 & 0 & 0 & .8 & 0 & 0 & 0 & -.4 \\ 0 & 0 & -.3 & 0 & 0 & .1 & 0 & 0 & 0 \end{pmatrix}$$

Such matrices usually have many zeros, representing cells that are not connected. For this reason, the connectivity data is more parsimoniously represented as a sparse matrix, storing row, column, weight information only for non-zero entries:

ROW	COL	WEIGHT
POST	PRE	WEIGHT
0	6	−.5
1	3	−.6
2	0	.4
2	1	−.1
2	5	.2
3	0	.5
3	7	−.8
⋮	⋮	⋮

As indicated by the alternate column heading above, individual records identify POSTsynatic and PREsynaptic cells by number.

This sparse matrix is a 3-column connectivity database table. However, neurons are often connected via two different synaptic weights (AMPA and NMDA for excitatory connects;

$GABA_A$ and $GABA_B$ for inhibitory connects). Therefore we use an additional column to store a second weight (Chover *et al.* 2001). We then use a fifth column to define the location of the connection on the postsynaptic cell.

Connections table

POST	PRE	WT_0	WT_1	LOCATION
4	6	.5	.1	PROXIMAL_APICAL
⋮	⋮	⋮	⋮	⋮

The **LOCATION** field indicates the postsynaptic target location in terms of dendritic fields. Alternatively, this could be used to identify a specific dendritic segment or to identify a distance form the soma.

The *Connections* table is related via a unique **CELL#** to a *Cells* table. This identifying number is used in the *Connections* table to identify PRE and POST -synaptic cells. The *Cells* table gives cell type and location. Cell locations (here on a two dimensional grid) are used to calculate distances to set axonal delays.

Cells table

CELL#	TYPE	X	Y
⋮	⋮	⋮	⋮
4	Layer_5_Pyramid	30	20
5	Basket_Cell	40	30
6	Thalamocortical_Cell	30	30
⋮	⋮	⋮	⋮

The *Cells* table can then be related to a *Neurotransmitter* table using the shared **TYPE** identifier. This permits identification of the presynaptic neurotransmitter which determines the receptor model that must be utilized in the postsynaptic cell.

Neurotransmitters table

TYPE	ID_{WT0}	ID_{WT1}
Layer_5_Pyramid	AMPA	NMDA
Basket_Cell	$GABA_A$	$GABA_B$
⋮	⋮	⋮

In this example, the *Connections* table is joined to the *Cells* table to identify the presynaptic cell type: Cell#6 is identified as of TYPE “Thalamocortical_Cell.” The result from this joint query is then itself joined with the *Neurotransmitters* table: this reveals that WT_0 represents an AMPA connection and WT_1 represents an NMDA connection. At each step, the relational joins are done using the EQW operator.

The network database can then be used for both network parameter assignment and network parameter verification. In practice, it is often easiest to consolidate the *Connections* and *Cells* table in order to have immediate access to cell identities without the more complex syntax required for the relational join. Script-language procedures map to and from the database and NEURON’s internal network parameters, making it easy to use the database to set synaptic

weights based on cellular relations: types of cells, cell locations, types of transmitters, etc. These manipulations were used to build the simulation discussed in the next example.

Example 3: Relating simulation output to parameters

We have begun to use NQS to evaluate simulation results in complex network models. We explored a thalamocortical response based on the model of Bazhenov *et al*. (1998; available at http://senselab.med.yale.edu/senselab/ModelDB). The simulation was driven with 8 Hz stimulation to the thalamocortical cells, simulating the effect of strobe stimulation via the retina. Hundreds of simulations were run in an automated fashion with different parameter sets. An NQS table was then built with one set of columns giving values of the parameters being varied and another set of columns giving scalar measures of the results, including time to first spike after stimulation, number of spikes within various post-stimulation periods, number of spikes within half-cycle, peak instantaneous frequency, etc. Each measure was assessed independently for the thalamocortical and cortical cell population as well as for the lumped excitatory population and for the entire population (all 4 cell types in the model). We also assessed different sizes of stimulations and then looked at center field, edge field, and out of field responses independently.

We used the database to explore the relation of parameters to particular attributes of network behavior and to compare network patterns to those obtained electrophysiologically. In a sense, we are opening up the black box of an automated parameter search as would be performed using an optimization routine. By substituting human exploration for the rigidity of a fitness function, we can come across potentially revealing relationships that would be missed by a fully automated procedure. This exploration can then provide clues for developing a fitness function for further automated exploration.

The spr() command was used to combine response measures in order to calculate an overall oscillation strength score. The object pointers available in NQS were used to provide pointers to raster plots that gave a quick visual image of overall activity in the network. A lumped score that was being evaluated was then quickly assessed by visual inspection of the graphics associated with rows that gave high scores contrasted with those that gave low scores.

Given suitable measures and scores, we identified parameters that covaried with the measures, considering whether some combination of parameter alterations might best predict the alteration in a measure. In cases where altering a parameter resulted in an abrupt change from low to high oscillation, we reexplored with new simulations to look at the intermediate values. Note that this is the traditional data-mining process: an interactive process of exploration, repeatedly drilling down, stepping back and assessing different angles. However, the use of simulation offers a significant advantage over the usual data-mining setting: new data can be generated on demand. Manual exploration can also be complemented by formal methods such as principal component analysis to determine combinations of parameters most strongly affect a given measure.

Fig. 3 shows a summary figure of one measure of oscillation strength with variation in 3 parameters. Response strength is represented by the size of circles in response to changes in synaptic strength at corticothalamic (x-axis), retinothalamic (y-axis) and thalamocortical (diagonal axis) locations. This graph demonstrates that oscillatory response requires a certain minimum input strength to be expressed (the bottom row shows little response while the middle and top row show similar responses). The oscillation is relatively insensitive to corticothalamic strength but is graded with increasing thalamocortical strength. This result was confirmed across several oscillation measures.

Discussion

NQS provides a general tool for the myriad data-management tasks that are a major part of the simulation endeavor. Parameter management has been emphasized in the examples given. NQS can form the substrate for development of stereotyped model representations that could be used to move a given model from one simulator to another. As compared to the use of a model metalanguage such as that of Goddard *et al*. (2001) a database format would be less flexible but would provide more rapid access and exploration through data-mining tools. In addition to parameter management, NQS is also a useful tool for management of simulation output and comparison with experimental data. This would be particularly useful for running large network simulations with immediate matching to a database of physiological results.

Traditional data flow in current scientific methodology goes from experiment to data storage to data-mining, with the loop closing as data-mining insights are used to develop hypotheses that suggest new experiments. In this view, simulation would be considered as just another data-mining tool. An alternative view would regard realistic simulation as a tool apart since it alone can provide causal explanation rather than just correlation. From this perspective, NQS could serve as a central resource for organizing data in an experiment→simulation→hypothesis→experiment loop.

Acknowledgments

The author wishes to thank Mike Hines and Ted Carnevale for continuing assistance with NEURON; Tom Morse for suggestions and help with SimToolDB; Richard Adams for reading the manuscript; Mark Stewart and Dan Uhlrich for use of physiological data; and 3 anonymous reviewers for many helpful comments. This research was sponsored by NIH (NS045612 and NS032187).

Appendix A: Basic NQS commands

FUNCTION	USAGE	DESCRIPTION
append	append(TUPLE)	append tuple
setcol	setcol("A",VEC)	copy contents of VEC into COL A
apply	apply("FUNC", "A", "B",…)	apply FUNC to vectors for each COL
cp	cp(DB)	copy table
delect	delect()	move values from selected back to main table (used after manipulating selected tuples)
fill	fill("A",x1, "B",x2,…) fill("A",vec1,…) fill("A", "Z",…)	fill COLs with corresponding values copy vec1 to COL A copy COL Z to COL A
fillin	fillin("A",x1, "B",x2,…)	fill in-place after select(−1…) (avoids large data copies)
gr	gr("A") gr(A"A A", A"B A") gr(…,[OPTIONS])	plot COL A against sequential integers plot COL A (y) vs. COL B (x) choose color, line type, superimpose on graph
map	map("FUNC", "A", "B",…)	call FUNC with vectors for all COLs
pr	pr("A", "B",…[,MAX])	print selected COLs through tuple MAX
qt	qt(&x1, "A",&x2, "B",…)	iterate through tuples setting x1,x2 scalars
rd	rd("FILENAME")	read table from file
remove	remove(TUPLE)	remove selected tuple
select	select([OPTIONS]) select(−1,[OPTIONS])	see text select in-place instead of copying tuples (−1 option avoids large data copies; see fillin)

FUNCTION	USAGE	DESCRIPTION
sort	sort("COL")	sort all tuples using numeric order of COL
spr	spr([COMMAND STRING])	see text
stat	stat("COL") stat("COL", "min")	print out mean,min,max,stdev for selected COL print out min for selected COL
strdec	strdec("A","B",…)	declare that these COLs contain strings
sv	sv("FILENAME")	save table or selected tuples to file
tog	tog()	switch between full table and selected

Appendix B: Selection criteria available for NQS select

NAME	SYMBOL	action
Numeric		
NEG	<0	numeric less than 0
POS	>0	numeric greater than 0
NOZ	!=0	numeric non-zero
GTH	>	greater than given value
GTE	>=	greater than or equal to given value
LTH	<	less than given value
LTE	<=	less than or equal to given value
EQU	==	equal to given value
NEQ	!=	not equal to given value
IBE	[ )	within closed/open interval
EBI	( ]	within open/closed interval
IBI	[ ]	within closed/closed interval
EBE	( )	within open/open interval
String		
SEQ	=~	string identity
RXP	~~	regular expression matching
Vector		
EQV		equal values in two columns
EQW		value present in a given vector

Appendix C: Implementation notes

The Neural Query System is written as a module for the NEURON simulation system. Its implementation consists of two parts. First, interpreted code in NEURON's hoc language implements the routines called by the user. Second, compiled C code provides the array functions needed to allow select() and sort() to execute rapidly. Compiled code also allows rapid vector-based calculations for data-mining. Further vector-based algorithms written in C can be easily added. For example, a back-propagation artificial neural network algorithm was ported from C code and made available as an ANN tool that does not use the neural simulation engine of NEURON itself (Lytton 2002).

Each column of the table is represented internally by a vector (array) whose ordered values represent the numerical values for the associated row in that column. String functionality is provided by using the vectors to store numeric pointers to a linked list of strings (List object in NEURON).

After parsing its arguments, the select command calls a C-coded slct command that runs through all the rows of the columns of interest doing the appropriate comparisons and building an index vector of rows matching all criteria (AND; matching any criterion for OR). This index vector is then used to make a separate table of all columns for these selected rows.

Although adequate in speed and size for current purposes, there are areas where the current implementation falls short. Although NQS databases can be stored and re-read, the data being used currently is stored in memory rather than on disk. Accessing large databases in this way requires highly inefficient memory swapping. An improved design would use a variety of secondary indices to allow rapid access to records on disk and only load them into memory as needed.

The combination of vector-oriented numerical operations and database functionality is comparable to MATLAB's DATABASE TOOLBOX. However, the MATLAB product does not provide an internal select function but constructs SQL queries which are sent to the connected database.

References

Ascoli G, Krichmar J, Scorcioni R, Nasuto S, Senft S. Computer generation and quantitative morphometric analysis of virtual neuron. Anatomy & Embryology 2001a;204:283–301. [PubMed: 11720234]

Ascoli G, Krichmar J, Nasuto S, Senft S. Generation, description and storage of dendritic morphology data. Phil. Trans. R. Soc. Lond. B 2001b;356:1131–1145. [PubMed: 11545695]

Ascoli G. Neuroanatomical algorithms for dendritic modelling. Network-Computation in Neural Systems 2002;13:247–260.

Bazhenov M, Timofeev I, Steriade M, Sejnowski T. Computational models of thalamocortical augmenting responses. Journal of Neuroscience 1998;18:6444–6465. [PubMed: 9698334]

Bower, J.; Beeman, D. The Book of Genesis. Vol. 2nd ed.. New York: Springer; 1998.

Chover J, Haberly L, Lytton W. Alternating dominance of NMDA and AMPA for learning and recall: a computer model. Neuroreport 2001;12:2503–2507. [PubMed: 11496138]

Davison A, Morse T, Migliore M, Shepherd G. Semi-automated population of an online database of neuronal models (ModelDB) with citation information, using PubMed for validation. Neuroinformatics 2004;2:327–332. [PubMed: 15365194]

Galassi, M.; Davies, J.; Theiler, J.; Gough, B.; Jungman, G.; Booth, M.; Rossi, F. Gnu Scientific Library: Reference Manual. Vol. 2nd ed.. Cambridge, MA: Network Theory; 2003.

Goddard N, Hucka M, Howell F, Cornelis H, Shankar K, Beeman D. Towards NeuroML: model description methods for collaborative modelling in neuroscience. Phil. Trans. R. Soc. Lond. B 2001;356:1209–1228. [PubMed: 11545699]

Hines M, Morse T, Migliore M, Carnevale N, Shepherd G. Modeldb: a database to support computational neuroscience. J Comput Neurosci 2004;17:73–77.

Lytton, W. From Computer to Brain. New York: Springer Verlag; 2002.

Poirazi P, Brannon T, Mel B. Arithmetic of subthreshold synaptic summation in a model ca1 pyramidal cell. Neuron 2003a;37:977–987. [PubMed: 12670426]

Poirazi P, Brannon T, Mel B. Pyramidal neuron as two-layer neural network. Neuron 2003b;37:989–999. [PubMed: 12670427]

Press, W.; Flannery, B.; Teukolsky, S.; Vetterling, W. Numerial Recipes in C: The Art of Scientific Programming. Vol. 2nd ed.. Cambridge: Cambridge University Press; 1992.

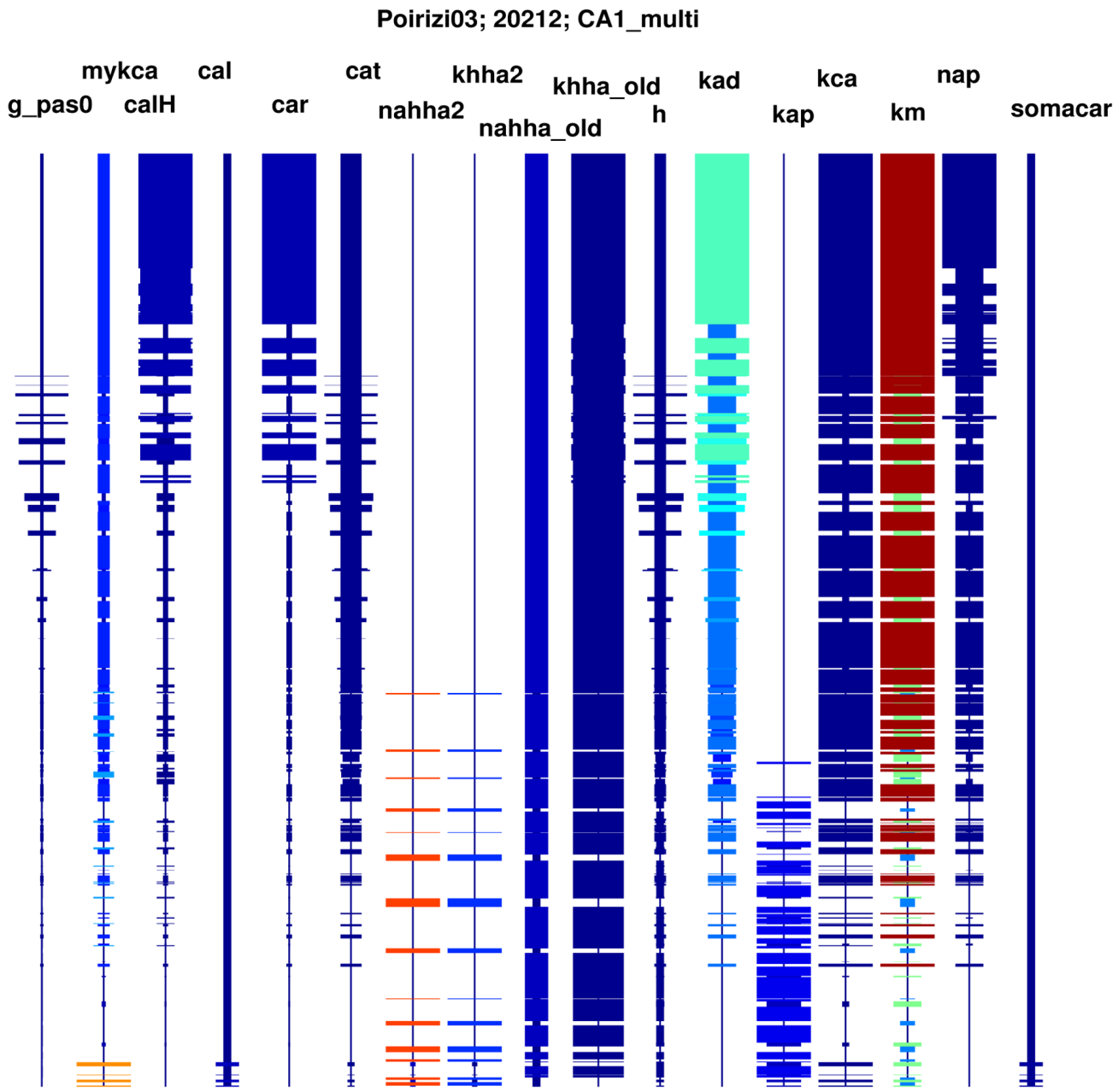

Fig. 1.
Graphical representation of columns showing maximal conductance values for model from Poirazi *et al*. (2003a,b)

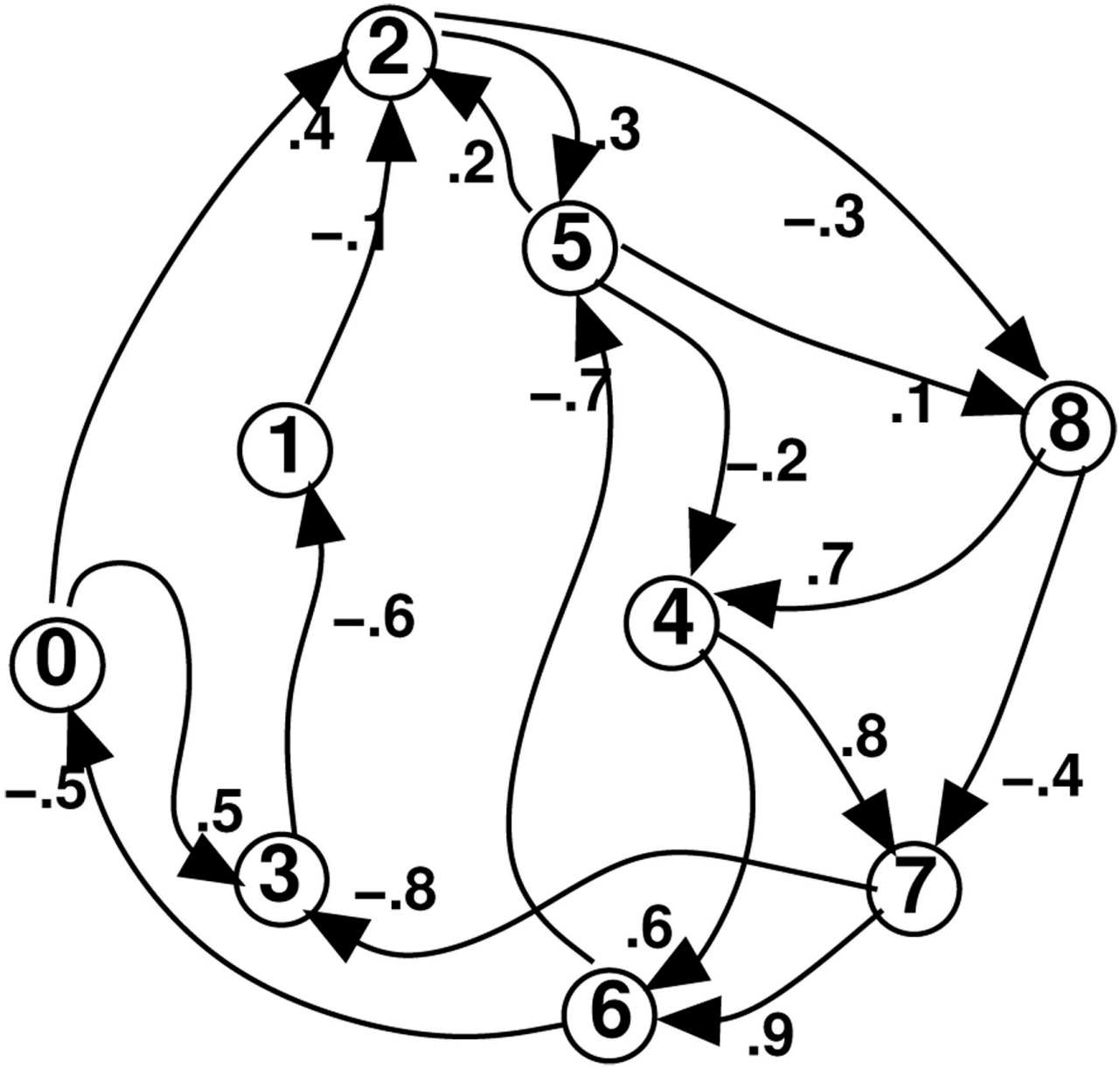

Fig. 2.
Ball and stick representation of a simple neural network

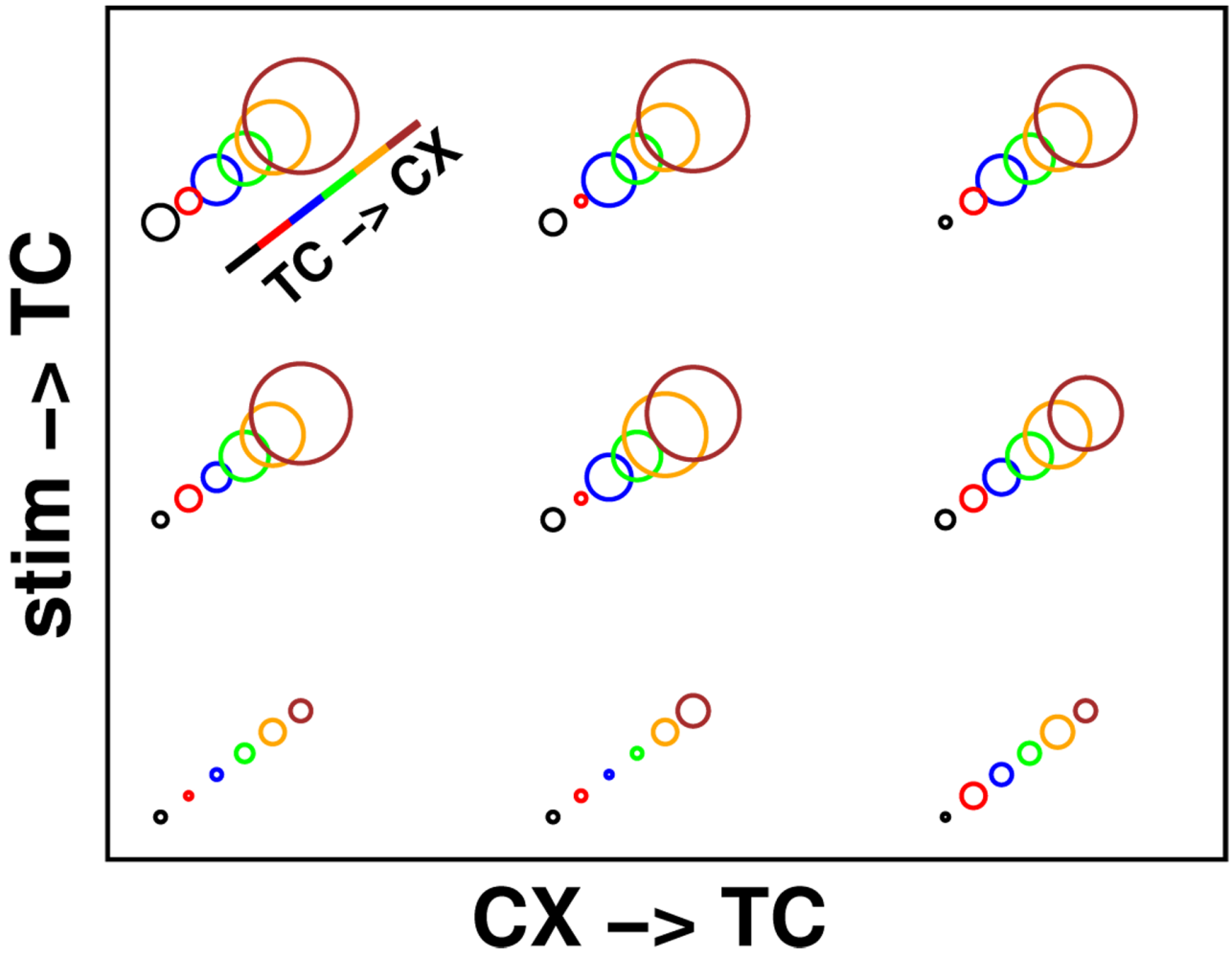

Fig. 3.
Oscillation strength with parameter variation in 3 dimensions