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

March 8, 2005

Running NQS

Neural Query System (NQS) is implemented within the NEURON simulation program. This manual assumes familiarity with NEURON and with the `hoc` (aka `oc`) programming language.

NQS utilizes a compiled NMODL file (vecst.mod) which must be linked into the simulator with the following command:

```
nrnivmodl vecst
```

Then the program can be run with

```
nrngui nqs.hoc
```

nqs.hoc loads several other hoc files that are present in the directory.

In order to use NQS, it is necessary to populate the database with data. This can be done most easily by reading in from a tab delimited file. The following sequence will set up a new database and append data from the included file *sample.asc*.

```
objref nq
nq=new NQS()
nq.rdcols("sample.asc")
```

Basic commands

Now we can play with the data in our database. Below the input is in `type` and the output in *italics*

```
nq.size
```

3 x 202,202,202

Note that the lengths of each row are given explicitly. This is redundant for a relational database structure where every column must be of the same length. In NQS, however, columns of different length are allowed to permit storage of spike train times and other data instances that yield variable length data. In such cases, there is no notion of record and the usual record-based commands given below do not apply.

`nq.gets` shows column headers; `nq.pr` prints out part of all of the records.

```
nq.pr(5)
COLA COLB COLC
16.45 15.95 0.04729
15.95 15.85 0.2355
15.85 15.27 8.091
15.27 12.91 0.3045
12.91 11.87 8.816
11.87 6.785 6.694
```

Note that this prints records 0 through 5; all record access is 0-offset denominated. One can use *e.g.,* `nq.pr(-5)` to see the last 5 rows or `nq.pr(18,50)` to see a slice or `nq.pr("COLA","COLB",5)` to print out only selected columns.

One can append to the database as follows:

```
nq.append(53.7,42.1,8.609)
```

The query command is select() which takes a number of test operators (Table 1):

```
nq.select("COLA","()",100,200,"COLB",">",70,"COLC",">0")
26
```

The return value of 26 indicates that 26 records were selected in which the values in "COLA" were between 100 and 200, values in "COLB" were greater than 70 and values in "COLC" were positive.

Having run a `select()` command, additional commands will pertain to the selected portion. For example `nq.pr()` will now print out only the selected records. The output will be headed with "*Selected:*" to indicate this. `nq.tog()` will toggle back and forth between the most recently selected records and the full database. `nq.tog(1)` will show which is being currently accessed; `nq.tog("DB")` will make current the full database; `nq.tog("SELECTED")` will make current the selected portion ("`SELECTED`" here can be abbreviated down to 3 letters).

`select()` always applies to the full database. Other commands can apply to the full db or to the most recently selected portion. `select()` also has operations that pertain to strings (see Table 1).

For example, `sp.gr("COLA","COLB")` will graph the chosen columns against one another. If we then `sp.tog()` and regraph with a different color we can see the selected values against the background of the full database Fig. 1

Control of the graphics is through the GRVEC tool which is described elsewhere. In brief, `attrpanl(0)` will bring up an attribute panel which will allow you to toggle to `Mark` instead of line graph. The `Color` field will allow you to change the color from black (1) to red (2).

`nq.sort("COLC")` will sort in numerical order. `nq.sort("COLC",-1)` will sort in

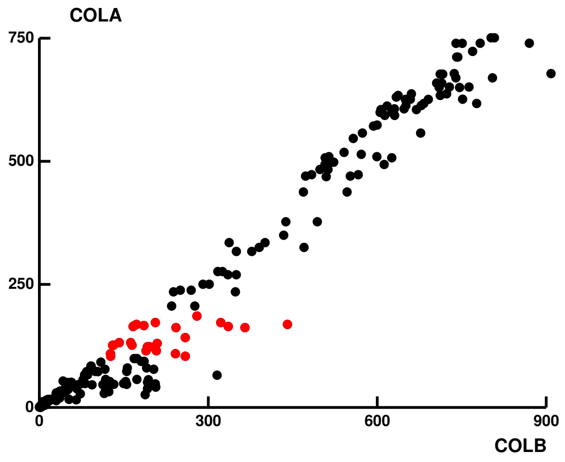

Fig. 1: Graph of COLA (ordinate) vs COLB (abscissa) from database nq. Full db in black, selected values in red

reverse numerical order.

The spreadsheet command, *e.g.,*

```
nq.spr("<SCR>.copy(<COLA>.c.mul(<COLB>).add(<COLC>).sub(3.1))")
```

will do vector operations (see NEURON's VECTOR manual page for details) on the columns, each of which is represented internally by a vector. The angle brackets indicate use of these vectors – if COLA was used instead of <COLA> the operation would pertain to a global numeric parameter named COLA. <SCR> references the scratch vector which can then be access as `nq.scr`. Note that the columns can similarly be accessed independently, *e.g.,* COLA as nq.v[0], COLB as nq.v[1], COLC as nq.v[2]. `nq.gets()` gives these index numbers next to the column names.

stat() will print out basic information about a column:

```
nq.stat("COLA")
```
max=750.879; min=0; mean=263.274; stdev=253.994;

Note that this is done by applying these NEURON vector dot operators to the underlying vector. This can be done selectively, *e.g.,* `nq.stat("COLA","var")` will return the variance since ".var" is a built-in operator. Note that additional dot operators can be readily added (compiled in) as needed – see *e.g.,* `xzero` in `vecst.mod` which counts zero crossings.

Similarly a user function that takes a vector and returns a value can be applied to a column using `applf(FUNC,COL)`.

Saving and reading to binary format are done with `nq.sv("FILENAME")` and `nq.rd("FILENAME")`. Reading a new database file (generally using a .nqs suffix) into nq will overwrite the previous data, *e.g.,*

```
nq.sv("aa.nqs") // saves to file aa.nqs
nq.rd("gfrpnq.nqs") // reads file into nq, overwriting prior data
```

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
User-defined		
FCN		user function(number) returns 1
FCS		user function(string) returns 1

Table 1: Selection criteria available for NQS `select`

will read the model data saved from the MFP package pertaining to the ModelDB model of *CA1 pyramidal neuron: dendritic spike initiation, Gasparini et al 2004*.

Additional details for `select` and `spr`

The NQS `select()` command takes any number of arguments in sets. Each set consists of a column name, an operator (Table 1) 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). Inner join for related databases is done using the vector oriented `EQW` operator which will reference values previously selected from a column of the same name in a separate database.

Although the NQS `select()` was not designed to replicate the agglutinative syntax of SQL `SELECT`, much of this functionality can be replicated by combining 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, such as 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 database 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 database for further exploration.

The full package consists of over 50 commands. The important commands are shown in Table 2. With the exception of `select` and `spr`, command arguments are straightforward. `spr` short for spreadsheet, allows values from various columns to be combined using the vector functionality available in NEURON. An example would be to calculate some statistic involving compartment diameter, distance, and an external parameter:

```
nqs.spr("<DATA>.copy(<DISTANCE>.c.sqrt.mul(paramA).div(<DIAM>))")
```

The angle brackets are used to indicate the names of columns in the nqs database. In this case a DATA column (which must have already been created) will get values calculated by multiplying the square root of DISTANCE by a scalar parameter (`paramA` – not in angle brackets) and dividing by DIAM. The "`.c.`" is NEURON vector notation indicating that a copy of the vector will be used. This copying ensures that the DISTANCE column will not itself be changed.

Review of general functionality

NQS handles basic DBMS functionality including: 1. creating tables; 2. inserting, deleting and altering tuples; 3. data queries. More sophisticated DBMS functionality such as indexing and transaction protection are not yet implemented. DBMS 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 database; 8. relational selections using criteria across related databases; 9. mapping of user specified functions onto particular columns.

Of all DBMS 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. Structured Query Language (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 to be added in further development of NQS.

FUNCTION	**USAGE**	**DESCRIPTION**
`append`	`append(TUPLE)`	append tuple
`apply`	`apply("FUNC","A","B",...)`	apply FUNC to vectors for each COL
`cp`	`cp(DB)`	copy database
`delect`	`delect()`	move values from selected back to main database (used after manipulating selected tuples)
`fill`	`fill("A",x1,"B",x2,...)`	fill COLs with corresponding values
	`fill("A",vec1,...)`	copy vec1 to COL A
	`fill("A","Z",...)`	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")`	plot COL A against sequential integers
	`gr("A","B")`	plot COL A (y) vs. COL B (x)
	`gr(...,[OPTIONS])`	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 database from file
`remove`	`remove(TUPLE)`	remove selected tuple
`select`	`select([OPTIONS])`	see text
	`select(-1,[OPTIONS])`	select in-place instead of copying tuples (-1 option avoids large data copies; see fillin)
`sort`	`sort("COL")`	sort all tuples using numeric order of COL
`spr`	`spr([COMMAND STRING])`	see text
`stat`	`stat("COL")`	print out mean,min,max,stdev for selected COL
	`stat("COL","min")`	print out min for selected COL
`strdec`	`strdec("A","B",...)`	declare that these COLs contain strings
`sv`	`sv("FILENAME")`	save database or selected tuples to file
`tog`	`tog()`	switch between full database and selected

Table 2: Basic NQS commands

Implementation notes

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.

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.

Each column of the database 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 database 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. First, all vectors are stored with double precision. It would be desirable to reduce the precision, particularly for those vectors whose values are being used as simple numeric pointers. These could be stored in single bytes, saving considerable space. Since a single byte is still too large for flags and other low-information identifiers, further consolidation could be achieved by storing information in bit-fields. Bit-masking in C would still allow rapid matching to identifiers.

Second, there is not currently any mechanism for including a columns of objects. This extension to an object-oriented database within the object-oriented NEURON simulator is straightforward and will be added in the future. Third, and most importantly, although NQS databases can be stored and re-read, the data being used currently is stored in memory rather than on disk. When large databases are built this will require memory swapping which is highly inefficient. Standard database design would optimize retrieval from disk by using a variety of secondary indices to allow rapid access. Rather than re-invent this particular wheel, it would be desirable to export and import to the formats of full databases that can then do disk management for large databases. These external databases could then be used to import data slices for further exploration with NQS. Ideally this would be done by having NQS construct calls to the external SQL interpreter, making external and internal queries seamless.