

University of Nevada

Reno

**Brainlab: a toolkit to aid in the design, simulation,
and analysis of spiking neural networks with the
NCS environment**

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

Rich Drewes

Dr. Frederick C. Harris, Jr., thesis advisor

May, 2005

UNIVERSITY
OF NEVADA
RENO

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

Rich Drewes

entitled

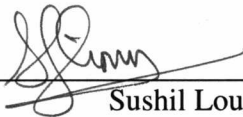
**Brainlab: a toolkit to aid in the design, simulation,
and analysis of spiking neural networks with the
NCS environment**

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE



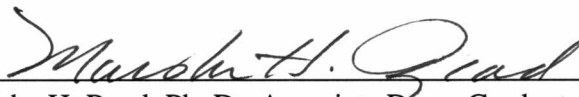
Frederick C. Harris, Jr., Ph. D., Advisor



Sushil Louis, Ph. D., Committee Member



Philip Goodman, M.D., M.S., Graduate School Representative



Marsha H. Read, Ph. D., Associate Dean, Graduate School

May, 2005

Abstract

The NCS (NeoCortical Simulator) system is a powerful batch processing spiking neural network simulator capable of efficiently working with networks of thousands of synapses at a level of biological realism extending to membrane dynamics and multiple ion channels. NCS is complex and can be difficult to use in several respects however, and its fullest potential is difficult to realize both for small projects and large projects. To address this problem, a variety of special purpose tools have been developed, but these tools lack generality, power, flexibility, and integration with each other. This thesis describes Brainlab, a set of tools designed to make working with NCS easier, more expressive, productive, and powerful. Brainlab is an integrated modeling and operating environment for NCS, based on a simple yet powerful standard scripting language (Python).

Acknowledgements

Thanks to my committee members for serving, and Dr. Harris for agreeing to chair. Thanks especially to Dr. Goodman for his vision, energy, and expertise building and sustaining the Brain Computation Laboratory at UNR.

Contents

Abstract	i
Acknowledgements	ii
Contents	iii
List of Figures	v
1 Motivation and background	1
1.1 NCS, the NeoCortical Simulator	1
1.2 NCS applications	3
1.3 The NCS input file format (the <code>.in</code> file)	4
1.4 Common NCS usage pattern	8
1.4.1 Manual <code>.in</code> file creation in an editor	8
1.4.2 Scripted Preprocessor	8
1.4.3 Templated Preprocessor	9
1.5 Why Brainlab?	10
2 Brainlab	13
2.1 Creating a network model	14
2.2 Component type libraries	14
2.2.1 Seeing what component libraries are available	15
2.2.2 Selecting component libraries	15
2.2.3 Seeing what component types are in a component library	15
2.2.4 Seeing the members of a component type	16
2.2.5 Modifying a member of a component type	17
2.2.6 Copying existing component types	18
2.2.7 Adding element instances to the model	19
2.3 Making connections in the model	28
2.4 Designated input/output layers and cell groups	30
2.5 Stimulus and report requests	31
2.6 Simulating a network model	32
2.7 Viewing a network model in three dimensions	34
2.8 Data loading utilities	38

2.9	Report plotting utilities	39
2.10	Experimental features	41
2.10.1	Areas	41
2.10.2	Enumerated Columns	41
2.11	Setting up Brainlab	42
2.11.1	Remote operation	42
2.11.2	Local operation	43
3	A detailed example	44
3.1	Constructing complex stimulus patterns	45
3.2	Genetic search of model parameters	47
3.3	Automated data analysis	50
4	Design and implementation issues	52
4.1	Python language features	52
4.2	Python scientific community	53
4.3	Python object orientation and Brainlab	54
4.4	Brainlab modules	55
4.5	Brainlab remote execution	56
5	Thoughts on a next generation NCS and future work	58
5.1	A next generation NCS	58
5.2	Future work	60
	Bibliography	63

List of Figures

1.1	The <code>BRAIN</code> section of a sample NCS <code>.in</code> file.	5
1.2	A <code>COLUMN</code> referenced from the <code>BRAIN</code> section in Figure 1.1.	6
2.1	An example of the <code>netplot.py</code> module of Brainlab, which allows interactive 3D examination of a network model. This screen capture shows the low-realism version of the display, which is most efficient for large network models or older display hardware with poor 3D hardware acceleration.	36
2.2	An example of the realism mode of Brainlab's <code>netplot.py</code> module. This allows 3D interactive exploration of models defined in Brainlab.	37
2.3	An example of three plot types, generated using Brainlab convenience functions, combined onto one master plot. The matplotlib library generates the underlying graphs.	40
3.1	A complex spike pattern set.	46
3.2	A spike pattern match plot. Larger circles comprise a pattern that is to be matched, and the overlaid smaller circles comprise the pattern that was found to be the closest match.	51

Chapter 1

Motivation and background

In the 1980s and 1990s, a growing realization of the limitations of artificial neural networks, and also a desire to investigate questions of biology in simulation, led to the creation of a number of biologically realistic spiking neural network simulation software packages. NEURON[8] and GENESIS[4] were and remain among the most popular of these systems. Though some improvements have been made in the ability of these programs to handle large numbers of neurons and synapses, they remain more at home modeling smaller numbers of very realistic neurons and synapses. There was a need for a simulation system with a reasonably high degree of biological realism that could also work efficiently with large numbers of cells (more than 10,000) and synapses (more than 1,000,000). This thesis concerns such a program, called NCS (the NeoCortical Simulator), and a toolkit designed to simplify its use, called Brainlab. The remainder of this chapter will describe the development of NCS and some aspects of its use that led to the creation of Brainlab.

1.1 NCS, the NeoCortical Simulator

NCS[19] began in 1998 as a serial implementation of a biologically realistic neural network simulator written by researcher Philip Goodman of the University of Nevada,

Reno, in the MATLAB[34] environment. Partial inspiration for the work came from Goodman's research fellowship at the Institute for Neuroinformatics at the University of Zurich/ETH, and also the recent design of new modeling approaches that promised considerable realism but also reasonable computational requirements[30, 31, 35]. The resulting neural simulator program was novel in that it combined the efficiency of templated AP spike waveforms (since the shape of those waveforms vary little within a given biological neural network), with realistic cell membrane and ion channel dynamics to determine spike onset. Even with the speed provided by templated AP waveforms, performance limitations of the MATLAB environment restricted this program to simulation of relatively small numbers of cells and synapses.

To increase execution performance (or to increase the potential network size that could be simulated in a given amount of time), Goodman collaborated with computer science professor Sushil Louis and student Ali Etezadi-Amoli in 1999 to create the first C version of the program. This was a fairly direct port of the MATLAB code and it was still a serial implementation. The speedup over the MATLAB version was approximately 10-fold. Further improvements to this code and the first parallel version of NCS were implemented by student Keith Wesolowski also under the direction of Louis.

In 2000, the program was largely rewritten by E. Courtenay Wilson[37], working under computer science professor Frederick C. Harris, Jr., using the C++ language and the MPI[7] standard parallel libraries. This was NCS version 3.

Starting in 2001, student James Frye, working under Harris, began another rewrite to optimize and reorganize the code[18]. He removed virtually all object-oriented features and the code was considerably tightened. Student Rich Drewes assisted with

this effort. Numerous errors and inefficiencies were eliminated from the code. This effort resulted in a new release of NCS version 3 in 2003, NCS version 4 in 2003-2004, and NCS version 5 later in 2004. The NCS code remains today nominally a C++ program, but with C coding style and no significant use of C++ object orientation, nor C++ standard template library, nor C++ I/O facilities.

Student James King took over maintenance of NCS in 2004 and also introduced a number of feature enhancements, including a network socket-based communication server for external programs to interface to NCS[22] (with early stage assistance from Drewes), improved support for multi-compartment neuron models, and several new model features including the recently characterized synaptic augmentation property of cortex.

NCS is described in several conference papers and technical reports[28, 29, 38, 39]. User documentation for NCS is available at[21].

1.2 NCS applications

A number of research projects have used NCS:

- In 2002-2003 James Maciokas and collaborators used NCS to investigate a hypothesis regarding entropy change through stages of bimodal (auditory and visual) neural processing[20, 27, 28].
- In 2002-2003, Waikul and collaborators created a world wide web interface to NCS to simplify design and submission of network models for simulation[36].
- In 2003, Juan Carlos Macera and collaborators created a first prototype demonstration of robot-to-NCS communication[25, 26].

- In 2004, Jake Blake and collaborators created and simulated a model of auditory neocortex[16].
- In 2004, Matt Ripplinger and collaborators created and simulated scalable “sheet” neuronal structures[33].
- In 2004, Brian Opitz and collaborators studied the effect on information transfer in a spiking neural network with selective knockout of channels in simulation[32].
- In 2004 Drewes and collaborators used a genetic algorithm to evolve a network model for evolutionary autonomous agents to perform a delayed-matching memory task using a neural network with layer-structured visual cortex[17].

All of these investigations were conducted within the Goodman Brain Computation Laboratory[5], affiliated with the Biomedical Engineering Program[13] of the University of Nevada, Reno[12], where the NCS software itself was developed.

1.3 The NCS input file format (the `.in` file)

NCS reads a description of a neural network model and other simulation parameters from a plain text file whose filename is supplied to NCS as a command line argument. For our purposes here it is not necessary to go into great detail about the format of this file, but we do wish to describe it generally in order to explain some of the shortcomings of working with it.

This input file, hereafter called a `.in` file after the convention of using `.in` as a filename extension for such files, contains a variable number of subsections. Each subsection starts with a line that contains the name of the subsection (which must be one of a limited number of keywords permitted by the system) and ends with a line

that contains `END_` with the section name appended. The first subsection in a `.in` file is the `BRAIN` section. In the `BRAIN` section of the file are defined global features that affect the entire simulation. For example, a line beginning with `JOB` defines a job name for the simulation. This portion of the `.in` file typically looks something like that shown in Figure 1.1.

Some of the lines in the `BRAIN` section in that figure are references to additional subsection blocks that must appear elsewhere in the `.in` file (the lines beginning with `COLUMN_TYPE`, `STIMULUS_INJECT`, and `REPORT` for this example). The name of the referenced object is the second item on the line, and that same name must be

```

BRAIN
  TYPE                AREA-BRAIN
  FSV                 10000.00
  JOB                 E0_332
  SEED                -333
  DURATION            80.0
  COLUMN_TYPE         col
  COLUMN_TYPE         datain0
  COLUMN_TYPE         keyin0
  COLUMN_TYPE         thal0
  CONNECT             datain0 datain0_1CELL ER SOMA
                    col layER ES SOMA E 1.000 10.000
  CONNECT             keyin0 keyin0_1CELL ER SOMA
                    col layL1 ES SOMA E 1.000 10.000
  CONNECT             thal0 thal0_1CELL ER SOMA
                    col layL1 ES SOMA E 0.500 10.000
  CONNECT             col layL1 ES SOMA thal0
                    thal0_1CELL ER SOMA E 0.500 10.000
  STIMULUS_INJECT    stimdatain0_0-inj-0
  REPORT              EMRep
  OUTPUT_CELLS       YES
  OUTPUT_CONNECT_MAP YES
  DISTANCE           YES
END_BRAIN

```

Figure 1.1: The `BRAIN` section of a sample NCS `.in` file.

specified as the second item on a `TYPE` line in another subsection that defines that referenced object. For example, the `.in` file with the `BRAIN` section in Figure 1.1 makes a reference to a column named `col` and therefore the `.in` file would also need something like that shown in Figure 1.2.

The `COLUMN` on the first of line of Figure 1.2 says that we are defining a cortical column structure, and the next line says that the name of this column is `col`. The block then identifies a number of layers that make up the column (the `LAYER_TYPE` lines) which must point to a `LAYER` block with the appropriate `TYPE` name later in the file. There are also some `CONNECT` blocks, each of which is one logical element broken out onto two lines in this listing. These `CONNECT`s are logically part of this `COLUMN` but also implicitly reference other subsections of the file, such as the type of synapse to use for the connection.

```

COLUMN
  TYPE                col
  COLUMN_SHELL        col_sh
  LAYER_TYPE          layL1
  LAYER_TYPE          layEM
  LAYER_TYPE          layER
  LAYER_TYPE          layES
  LAYER_TYPE          layI1
  CONNECT             layL1 ES SOMA layEM ES SOMA
                    E 0.600 10.000
  CONNECT             layER ES SOMA layEM ES SOMA
                    E 1.000 10.000
  CONNECT             layER ES SOMA layES ES SOMA
                    E 0.600 10.000
  CONNECT             layI1 I1 SOMA layES ES SOMA
                    I 0.600 10.000
END_COLUMN

```

Figure 1.2: A `COLUMN` referenced from the `BRAIN` section in Figure 1.1.

Some aspects of the `.in` file format should now be apparent:

1. Even relatively simple models will have very long input files.
2. Making a change in one place in the input file, for example changing the name of an object or changing the pattern of connections in the model, could require many coordinated changes throughout the `.in` file.
3. Because of the length of the `.in` file, it is very difficult to learn much about the physical structure of a model by looking at the `.in` file.
4. Constructing a `.in` file manually could be very time consuming and prone to error.
5. To run a model a number of times with a variation of some parameter (a common experimental requirement) would require a version of the `.in` file for each run, and the results would have to be collected separately.

We should also point out that though the `.in` file format has some notion of hierarchy, since objects can reference other objects, there is no support for creation of macros or any other sort of higher level abstractions in the `.in` file. Similarly, there are no looping constructs permitted in a `.in` file. Numerical calculations are also very limited. There are a few historical quirks of the `.in` file format that further complicate working with the file directly. For example, though a `LAYER` can be defined once and reused in multiple `COLUMNS`, a `COLUMN` can only be referenced once. To be reused, a `COLUMN` must be completely defined again with the same structure and a new name. The lack of a proper type-of abstraction for `COLUMNS` seems to be simply a case of mistaken implementation. There are indications that a proper type-of abstraction

for `COLUMNS` was designed, and that the `BRAIN` section of the `.in` file was intended to contain references to `COLUMN_SHELL` blocks instead of `COLUMN` blocks as is currently the case. Regardless, the current state of affairs prevents truly hierarchical models from being represented logically in a `.in` file.

1.4 Common NCS usage pattern

In our experience, users of NCS typically progress through several stages of a common pattern of usage.

1.4.1 Manual `.in` file creation in an editor

Initially a new user of NCS will typically experiment with creating a brain model (`.in` file) by hand in a text editor. Quite often the editing is done on a personal workstation separate from the actual compute cluster. Then the user will copy the file to the cluster, invoke the NCS simulator on the file, and copy the output report files back to the personal workstation for analysis. This is in the best case; in reality, hand edited `.in` files are prone to syntactical and typographical errors, and even getting one file to simulate properly often requires anywhere from a few to a few dozen edit-copy-run attempts.

After using NCS this way for a while, most users working with network models of any significant complexity create some sort of partially automated `.in` file generation tool: a preprocessor in a scripting language.

1.4.2 Scripted Preprocessor

The next step usually taken by serious users of NCS is to enlist a scripting language, for example MATLAB[34], as a tool to automate some of the tedious aspects

of manual `.in` file generation. Often these preprocessors contain an intermixture of literal text chunks and algorithmically generated sections (for example, loops over x and y dimensions to create arrays of columns and connections among them).

There are a number of problems with this approach. It is still prone to error; there is no guarantee that the output `.in` file is coherent. Lack of convenient abstractions, or even standard naming conventions, can lead to situations where the model cannot easily be extended to allow some new model feature, without extensive recoding. Time is spent developing the preprocessor that could better be spent on the design of the models themselves.

Most users of NCS actually seem to stop at this stage and live with the difficulties. The resulting preprocessor systems are generally suitable only for one problem, the one they were created for, and they are cumbersome to use and inflexible. Model changes typically do require extensive recoding. While one user can sometimes reuse chunks of code for a later project, we are not aware of NCS users making significant use of others' preprocessor code in their own projects. Each user reinvents the solution and rediscovers the pitfalls of the approach.

1.4.3 Templated Preprocessor

There have been limited efforts by others to create templated preprocessors that would accept some higher-level abstractions and map them to a `.in` file for use by NCS. One such project by King saw some limited use. This tool is a templated preprocessor written in C. It accepts a statically defined file, in a new format, that contains a templated description of some higher level abstractions (most significantly, a “sheet” abstraction that is basically an array of columns). The tool outputs another file with the higher-level abstractions reduced to component parts expressible in the

standard NCS `.in` format. Some portions of the new input file are simply transcribed literally to the generated `.in` file.

This tool was used in at least one project[33] but it is deficient in several important ways. The tool did not support the creation of new abstractions by a user, it simply provided a few new ones (like the sheet) for one particular project. Since it was coded in C, it was not easy for non-programmer users of NCS to extend it in any way.

1.5 Why Brainlab?

For small projects, NCS requires considerable setup overhead and complication to accomplish even the simplest tasks, and the batch processing work flow makes experimentation difficult and time consuming. For large scale projects, NCS lacks extensibility without resorting to C programming, lacks convenient model generation, and lacks support for some important high-level abstractions and no ready means to add them. The Brainlab toolkit offers an experimenter the following enhancements over working directly with NCS:

1. An interactive shell for simple experimentation, making NCS a more suitable educational tool for learning the behavior of spiking neural networks and also a more convenient platform for experienced users to explore the behavior of new cell or network elements.
2. A convenient platform for parameterized control of sets of experiments.
3. A convenient platform for scripted regression testing of NCS itself, with flexible output validation.
4. Scripted, algorithmic generation of neural network models rather than NCS's

native static file specification of networks.

5. Convenient, integrated, graphical on-line reporting and plotting of cell spiking activity.
6. Convenient, integrated, on-line three-dimensional plotting of neural network architecture for expository and diagnostic purposes.
7. Support for higher level abstractions than those provided natively in NCS (for example support for *areas*, composed of arrays of columns, and a variety of distinct area-to-area synaptic connection patterns), and an easy way to add new ones.
8. Support for lower level abstractions too unwieldy to reasonably manage in native NCS (for example, columns where all cells are enumerated and independently addressable).
9. A container for a standard and extensible library of network building blocks (for example channels, cell types, columns, spike templates), where all components are guaranteed to interoperate, utilize consistent naming conventions, and may be manipulated programmatically as variable objects rather than text chunks.
10. A more convenient, higher level, object-oriented representation of neural networks that hides many complexities and inconveniences inherent in NCS's native `.in` file format.
11. The ability to convert a neural network description into a chromosomal representation suitable for use with a genetic algorithm.

12. The ability to conveniently and transparently extend all of these capabilities without recourse to coding in NCS's native programming environment (the C/C++ language).

The remainder of this thesis will describe Brainlab in more detail. Chapter 2 begins a tutorial-style introduction of some of Brainlab's features, starting with simple model building, working with the standard component type library, loading NCS report data, creating simple graphs, and viewing three-dimensional representations of neural models. Chapter 3 provides a more detailed and concrete look at some of the programming techniques used to create a real-world experiment with Brainlab. In Chapter 4 we will briefly consider some of the design choices made during the creation of Brainlab. Chapter 5 will conclude with a proposal describing how NCS might be improved by taking advantage of a tighter integration with a Brainlab-like modeling system, and a description of some enhancements planned for future versions of Brainlab.

Chapter 2

Brainlab

For my first serious research using NCS, I started using a MATLAB-based preprocessor system to create the `.in` files for the simulations. (This MATLAB code was initially derived from earlier work in our lab by Maciokas.) As I began extending this code for my experiments I found it increasingly difficult to manage the preprocessor. When my experiment required investigating variable model architectures explored with a genetic algorithm, the preprocessor system became too unwieldy and complex to reasonably use. I began to think about a more flexible, powerful, and concise modeling system, and this effort resulted in the first version of Brainlab. The Brainlab environment went through several more significant revisions as I required even more flexibility for experiments testing a proposed information processing function for a cortical microcircuit, which required complex input stimulus protocols and analysis. Gradually Brainlab evolved from just a model development tool into an integrated modeling and experimental environment for NCS.

The remainder of this chapter is an introduction to using Brainlab for modeling and experimentation.

2.1 Creating a network model

In Brainlab, every brain model is an instance of a Python object class called `BRAIN`. Creating a brain object is via the usual Python means:

```
b=BRAIN()
```

The variable `b` then refers to the newly created, and initially empty, brain model. Later we will discuss how to add neuron instances (and their containers, cortical columns), synapses, and other object instances to the `BRAIN` object. But first, we will describe in the next section the standard library of *types* of objects that each `BRAIN` object contains.

2.2 Component type libraries

When a `BRAIN` object is created, it contains a default set of commonly used types of neural network modeling components. (There are initially no *instances* of these types in the brain model.) Component types include ion channels, synapse facilitation and depression profiles, synaptic Hebbian learning parameters, cell definitions, and more. These component types can be directly instantiated and then used for construction of network models, or they can be modified in place and then used in a model, or they can be copied to new types with different names and then the copies can be modified and instantiated for use in a model. (Copying existing types will be demonstrated in Section 2.2.6). The component types are contained in standard Python dictionaries, and the keys of the dictionary are simply the text names of the components.

2.2.1 Seeing what component libraries are available

Multiple libraries are permitted, and each has a name. The standard library of component types is called `standard` and it is selected by default when a new `BRAIN` object is created. The names of available libraries are the keys to a dictionary in `BRAIN` called `libs`. The following code fragment shows that a newly created brain has only one default library available, the standard one:

```
print b.libs.keys()
```

The output is a list containing just the name of the standard library:

```
['standard']
```

2.2.2 Selecting component libraries

A component type library can be selected as the default for use by supplying its name to the `BRAIN`'s `SelectLib()` method:

```
b.SelectLib('standard')
```

The standard library does not need to be explicitly selected as above, though it does no harm to do so. It is automatically selected when a `BRAIN` is created.

2.2.3 Seeing what component types are in a component library

A component type library contains a number of sub-dictionaries that contain neural modeling component types. The keys to the library define the categories of the available components. The following code fragment

```
print b.libs['standard'].keys()
```

shows the following available component types:

```
['comptypes', 'spks', 'chantypes', 'spsgs', 'cols',
 'celltypes', 'sls', 'syntypes', 'lays', 'sfds']
```

The keys to the library dictionary in this case are neuron compartment types (`comptypes`), spike profiles (`spks`), channel types (`chantypes`), post-synaptic gap waveforms (`spsgs`), column types (`cols`), cell types (`celltypes`), synaptic long-term Hebbian learning profiles (`sls`), synapse types (`syntypes`), layer types (`lays`), and short term facilitation/depression profiles (`sfds`).

2.2.4 Seeing the members of a component type

Members of each component type can be viewed by simply printing the keys of the appropriate dictionary, like this:

```
lib=b.libs['standard']
chantypes=lib['chantypes']
print chantypes.keys()
```

This generates the output list:

```
['a-1', 'm-1', 'ahp-2']
```

The contents of an individual member of the type can be printed by accessing the dictionary element with the key that is the name of the type from the above list:

```
print chantypes['ahp-2']
```

which shows the following:

```

CHANNEL Kahp
  TYPE                ahp-2
  REVERSAL_POTENTIAL -80.000000 0.000000
  M_INITIAL           0.300000 0.000000
  M_POWER             2.000000 0.000000
  CA_SCALE_FACTOR     0.000125 0.000000
  CA_EXP_FACTOR       2.000000 0.000000
  CA_HALF_MIN         2.500000 0.000000
  CA_TAU_SCALE_FACTOR 0.003000 0.000100
  UNITARY_G           0.054000 0.000000
  STRENGTH            0.200000 0.020000
END_CHANNEL

```

Note that the element is printed in exactly the format that this type of element appears in the NCS `.in` file. This is not an accident; printing any type of object results in output that is suitable for direct inclusion in a NCS `.in` file. In fact, as we shall see, printing the highest-level object, the `BRAIN` itself, results in a complete `.in` file suitable for simulation with NCS.

2.2.5 Modifying a member of a component type

The individual NCS parameters associated with a component type element are stored in a parameter dictionary called `parms`. The keys to this dictionary are (with a few exceptions described later) simply the names of normal NCS parameters. Modifying the contents of a parameter dictionary element makes a change to that element type, and this change will affect all instances of that element. For example, we could change the `STRENGTH` of the channel `ahp-2` with the following bit of code:

```

c=chantypes['ahp-2']
c.parms['STRENGTH']='0.300000 0.030000'
print chantypes['ahp-2']

```

which gives the following output:


```

CHANNEL Kahp
  TYPE                ahp-2
  REVERSAL_POTENTIAL -80.000000 0.000000
  M_INITIAL          0.300000 0.000000
  M_POWER            2.000000 0.000000
  CA_SCALE_FACTOR    0.000125 0.000000
  CA_EXP_FACTOR      2.000000 0.000000
  CA_HALF_MIN        2.500000 0.000000
  CA_TAU_SCALE_FACTOR 0.003000 0.000100
  UNITARY_G          0.054000 0.000000
  STRENGTH           0.300000 0.030000
END_CHANNEL

```

Notice that the last parameter line, the `STRENGTH`, differs from the `STRENGTH` shown in the output in Section 2.2.4.

2.2.6 Copying existing component types

The `BRAIN Copy()` method conveniently makes copies of component types. It accepts three parameters: the dictionary that contains the object to be used as the template for the new object, the string name of the key in that dictionary that will be used as the template, and the string name for the result. The result is placed in the same dictionary. For example, if instead of modifying the existing `ahp-2` channel we wished to create a new channel type named `testchan` based on `ahp-2` we could do this:

```

lib=b.libs['standard']
cnew=b.Copy(lib['chantypes'], 'ahp-2', 'testchan')
cnew.parms['REVERSAL_POTENTIAL']='-85.00000 0.000000'
print 'The available channel types:'
print lib['chantypes'].keys()
print 'The new channel testchan just created:'
print lib['chantypes']['testchan']

```

which prints:

```

The available channel types:
['a-1', 'testchan', 'm-1', 'ahp-2']
The new channel testchan just created:
CHANNEL Kahp
      TYPE                testchan
REVERSAL_POTENTIAL      -85.00000 0.000000
M_INITIAL                0.300000 0.000000
M_POWER                  2.000000 0.000000
CA_SCALE_FACTOR          0.000125 0.000000
CA_EXP_FACTOR            2.000000 0.000000
CA_HALF_MIN              2.500000 0.000000
CA_TAU_SCALE_FACTOR      0.003000 0.000100
UNITARY_G                 0.054000 0.000000
STRENGTH                 0.300000 0.030000
END_CHANNEL

```

Note that the `REVERSAL_POTENTIAL` differs from the example in Section 2.2.5.

In general, when creating a model in Brainlab, the modeler has two choices: to create an element (`CELL`, `LAYER`, and so on) from scratch using the object creator function for that class (`c=brain.CELL()`, `l=brain.LAYER()`, and so on), or instead, use `Copy()` to clone an existing element from the library or from an object created earlier in the script. Either approach is perfectly valid, and the designer should use the one that is most efficient for the job.

2.2.7 Adding element instances to the model

So far we have been modifying component types and examining their containers, the component type libraries. A brain model will also contain actual instances of these types. In this section we will describe how to add those instances.

In NCS, cells cannot exist on their own but rather only as part of a higher level structure called a column. A column is composed of one or more layers, which in turn is composed of one or more groups of cells. Brainlab has `COLUMN`, `LAYER`, and `CELL` objects that correspond to these structures. The general procedure for creating a

brain model is to find the lowest level object in the standard component type library that corresponds fairly closely to what you want to model and build up from there. By “fairly closely” we mean that the effort required to modify the standard component is less than the effort required to build it from scratch. Remember also that even if you only wish to simulate a single cell or pair of cells, you must place them into columns (there are also convenience `BRAIN` methods that simplify this).

For example, let us consider construction of a model of a four layer column. Assume we are satisfied with the existing cell types `E` and `I` (excitatory and inhibitory) in the standard component library. We start by assigning a variable to each of these cell types:

```
# Create a new empty brain model in variable b:
b=BRAIN()

# Get the standard component library from the brain:
lib=b.libs['standard']
celltypes=lib['celltypes']

# Variable ecell will hold the standard excitatory cell we will use:
ecell=celltypes['E']

# Variable icell will hold the standard inhibitory cell we will use:
icell=celltypes['I']
```

Next we will create an empty `COLUMN` container and add it to our brain:

```
# Put an empty column into the new brain:
c=b.COLUMN()
b.AddColumn(c)
print c
```

which shows the following output:

```
COLUMN
  TYPE          col1
  COLUMN_SHELL  col1_sh
END_COLUMN
```

Notice that the column was automatically assigned the name (TYPE) `col1` when it was added to the BRAIN. Unless we specify a name (TYPE) explicitly, one will be provided in `AddColumn()` since NCS requires that all such entities have a name. If we didn't care about the name, we could let Brainlab assign it for us while we continue to reference the variable returned from the object instantiation any time we needed to reference the column. Brainlab would simply handle all the text name references behind the scenes, and we could ignore them. This is often quite a convenience. If we did want to provide our own name for the COLUMN, we could pass it in the `parms` dictionary as an argument when we construct the column, like this:

```
# Put an empty column named testcol into the new brain:
c=b.COLUMN({'TYPE': 'testcol'})
b.AddColumn(c)
print c
```

to produce this:

```
COLUMN
      TYPE                testcol
      COLUMN_SHELL        testcol_sh
END_COLUMN
```

Any other parameters permitted by the NCS `.in` file for that item type could also be passed in that argument¹.

Now we will construct the layers inside the column. We will have four layers of excitatory cells and one of inhibitory cells. (Note that it would be more biologically accurate to have the inhibitory cells spread out among the other layers, and that could easily be accomplished by adding a second cell group to each layer. For this example we will put all inhibitory cells in a separate layer to simplify exposition.)

Here is the code fragment:

¹In Python, braces `{}` specify a dictionary. The colon between the two strings separates a name/-value pair. If there were more than one name/value pair, the pairs would be separated by commas.

```

l1=b.LAYER({'TYPE':'layL1'})
c.AddLayerType(l1)
l1.AddCellType(ecell, 10)
em=b.LAYER({'TYPE':'layEM'})
c.AddLayerType(em)
em.AddCellType(ecell, 10)
er=b.LAYER({'TYPE':'layER'})
c.AddLayerType(er)
er.AddCellType(ecell, 10)
es=b.LAYER({'TYPE':'layES'})
c.AddLayerType(es)
es.AddCellType(ecell, 10)
# Common inhibitory cells for entire column.
i1=b.LAYER({'TYPE':'layI'})
c.AddLayerType(i1)
i1.AddCellType(icell, 10)

print "Here is the column, with four layers:"
print c
print "Here is layer es, with 10 cells named 'E':"
print es

```

and the output:

```

Here is the column, with four layers:
COLUMN
      TYPE                testcol
COLUMN_SHELL            testcol_sh
LAYER_TYPE              layL1
LAYER_TYPE              layEM
LAYER_TYPE              layER
LAYER_TYPE              layES
LAYER_TYPE              layI
END_COLUMN

Here is layer es, with 10 cells named 'E':
LAYER
      TYPE                layES
LAYER_SHELL            layES_sh
CELL_TYPE              E 10
END_LAYER

```

Printing the column does not recursively print all the referenced sub-structures. Also,

notice that only the highest level structure instance, the column, must be added to the brain. The layers are added to the COLUMN object that directly contains them and not to the BRAIN object. Likewise, the CELL objects are added to the LAYER object. Printing the entire brain, however, recursively descends through all the objects that have been added to the BRAIN. Each object is printed in its proper sequence so that the combined resulting output has the proper format to be used as an input file for NCS. The command

```
print b
```

results in the output:

```
BRAIN
  TYPE          testbrain
  FSV           10000.00
  JOB           testbrainjob
  SEED          -99
  DURATION      1.0
  COLUMN_TYPE   col1
  COLUMN_TYPE   testcol
  OUTPUT_CELLS  YES
  OUTPUT_CONNECT_MAP YES
  DISTANCE      YES
END_BRAIN

# SECTION fill columns
COLUMN_SHELL
  TYPE    col1_sh
  WIDTH   300.0000
  HEIGHT  150.0000
  LOCATION 10.0000 20.0000
END_COLUMN_SHELL

COLUMN
  TYPE          col1
  COLUMN_SHELL  col1_sh
END_COLUMN
```

```

COLUMN_SHELL
  TYPE      testcol_sh
  WIDTH     300.0000
  HEIGHT    150.0000
  LOCATION  10.0000 20.0000
END_COLUMN_SHELL

```

```

COLUMN
  TYPE      testcol
  COLUMN_SHELL testcol_sh
  LAYER_TYPE layL1
  LAYER_TYPE layEM
  LAYER_TYPE layER
  LAYER_TYPE layES
  LAYER_TYPE layI
END_COLUMN

```

```

# SECTION fill layers
LAYER_SHELL
  TYPE      layL1_sh
  UPPER     80
  LOWER     20
END_LAYER_SHELL

```

```

LAYER
  TYPE      layL1
  LAYER_SHELL layL1_sh
  CELL_TYPE E 10
END_LAYER

```

```

LAYER_SHELL
  TYPE      layEM_sh
  UPPER     80
  LOWER     20
END_LAYER_SHELL

```

```

LAYER
  TYPE      layEM
  LAYER_SHELL layEM_sh
  CELL_TYPE E 10
END_LAYER

```

```

LAYER_SHELL
  TYPE      layER_sh
  UPPER     80

```

```

    LOWER    20
END_LAYER_SHELL

```

```

LAYER
    TYPE                layER
    LAYER_SHELL        layER_sh
    CELL_TYPE          E 10
END_LAYER

```

```

LAYER_SHELL
    TYPE    layES_sh
    UPPER   80
    LOWER   20
END_LAYER_SHELL

```

```

LAYER
    TYPE                layES
    LAYER_SHELL        layES_sh
    CELL_TYPE          E 10
END_LAYER

```

```

LAYER_SHELL
    TYPE    layI_sh
    UPPER   80
    LOWER   20
END_LAYER_SHELL

```

```

LAYER
    TYPE                layI
    LAYER_SHELL        layI_sh
    CELL_TYPE          I 10
END_LAYER

```

```

# SECTION cells
CELL

```

```

    TYPE                E
    COMPARTMENT        SOMA1 SOMA1_name 0.00000 0.00000 0.00000
END_CELL

```

```

CELL

```

```

    TYPE                I
    COMPARTMENT        SOMA1 SOMA1_name 0.00000 0.00000 0.00000
END_CELL

```

```

# SECTION stimulus
# SECTION stiminject

```



```
# SECTION report
# SECTION compartments
COMPARTMENT
```

```

TYPE                SOMA1
SPIKESHape          AP_Hoffman
SPIKE_HALFWIDTH     10.000000 0.000000
TAU_MEMBRANE        0.015000 0.000500
R_MEMBRANE          200.000000 3.000000
THRESHOLD           -40.000000 1.000000
LEAK_REVERSAL       0.000000 0.000000
LEAK_CONDUCTANCE    0.000000 0.000000
VMREST              -60.000000 1.000000
CA_INTERNAL          100.000000 0.000000
CA_EXTERNAL          0.000000 0.000000
CA_SPIKE_INCREMENT  300.000000 20.000000
CA_TAU              0.070000 0.001000
CHANNEL              ahp-2
CHANNEL              m-1
CHANNEL              a-1
END_COMPARTMENT
```

```
# SECTION channels
CHANNEL Kahp
```

```

TYPE                ahp-2
REVERSAL_POTENTIAL -80.000000 0.000000
M_INITIAL           0.300000 0.000000
M_POWER             2.000000 0.000000
CA_SCALE_FACTOR     0.000125 0.000000
CA_EXP_FACTOR       2.000000 0.000000
CA_HALF_MIN         2.500000 0.000000
CA_TAU_SCALE_FACTOR 0.003000 0.000100
UNITARY_G           0.054000 0.000000
STRENGTH            0.200000 0.020000
END_CHANNEL
```

```
CHANNEL Km
```

```

TYPE                m-1
REVERSAL_POTENTIAL -80.000000 0.000000
M_INITIAL           0.300000 0.010000
M_POWER             1.000000 0.000000
E_HALF_MIN_M        -44.000000 0.200000
SLOPE_FACTOR_M      40.000000 20.000000 8.800000
SLOPE_FACTOR_M_STDEV 0.000000
TAU_SCALE_FACTOR_M  0.303000 0.000000
UNITARY_G           0.084000 0.000000
STRENGTH            0.060000 0.002000
```

END_CHANNEL

CHANNEL Ka

TYPE	a-1
REVERSAL_POTENTIAL	-80.000000 0.000000
M_INITIAL	0.300000 0.010000
M_POWER	1.000000 0.000000
E_HALF_MIN_M	-21.300000 0.200000
SLOPE_FACTOR_M	35.000000
SLOPE_FACTOR_M_STDEV	0.500000
V_TAU_VOLTAGE_M	100.000000
V_TAU_VOLTAGE_M_STDEV	0.000000
V_TAU_VALUE_M	0.000200 9999.000000
V_TAU_VALUE_M_STDEV	0.000000
H_INITIAL	0.600000 0.005000
H_POWER	1.000000 0.000000
E_HALF_MIN_H	-58.000000 0.210000
SLOPE_FACTOR_H	8.200000
SLOPE_FACTOR_H_STDEV	0.500000
V_TAU_VOLTAGE_H	-21.000000 -1.000000 10.000000 21.000000
V_TAU_VOLTAGE_H_STDEV	0.000000
V_TAU_VALUE_H	0.005000 0.001000 0.015000 0.020000 0.2500
V_TAU_VALUE_H_STDEV	0.000000
UNITARY_G	0.120000 0.000000
STRENGTH	0.100000 0.002000

END_CHANNEL

SECTION spikeshape

SPIKESHape

TYPE	AP_Hoffman
VOLTAGES	-38.000000 -35.000000 -30.000000 -20.000000
	-7.000000 15.000000 30.000000 20.000000
	7.000000 -8.000000 -16.000000 -22.000000
	-28.000000 -33.000000 -37.000000 -40.000000
	-43.000000 -45.000000 -47.000000 -49.000000
	-50.000000

END_SPIKESHape

SECTION syn_psg

SECTION syn_facil_depress

SECTION syn_learning

SECTION synapse

Notice here that:

- Both the original column named by default `col1` and also our explicitly named column `testcol` are present, since they were both added to the `BRAIN`.
- The column named `col1` has no layers and therefore no cells either. It will have no functional role in any simulation.
- Since no synaptic connections have been specified, no synapse items were found during the recursive descent of the objects added to the `BRAIN` object. Therefore several sections at the end of the file (such as `SECTION syn_psg`) are empty.

2.3 Making connections in the model

In NCS, synaptic connections can be specified at three levels: from one cell group to another cell group within a layer, from one one cell group in one layer to another cell group in another layer in that same column, and from one cell group in one layer to another cell group in another layer in a different column. Conventionally, the first category of connections appears in a `LAYER...END_LAYER` block of the `.in` file, the second category appears in a `COLUMN...END_COLUMN` block, and the last category is made in the `BRAIN...END_BRAIN` block. (Note that in principle all connections could be defined at the `BRAIN` level; all connections, once made, are functionally identical. The only difference is the amount of information that is specified in the connection request. Connections made at the lower levels make the obvious assumptions about connection endpoints based on the context of the block where they are defined.)

In Brainlab, connections are made with the `AddConnect(from, to)` method. This method exists in the `BRAIN` object, the `COLUMN` object, and the `LAYER` object. The

arguments `from` and `to` are tuples that specify a (column, layer, cell group, compartment) in the case of a `BRAIN.AddConnect()`, a (layer, cell group, compartment) in the case of a `COLUMN.AddConnect()`, or a (cell group, compartment) in the case of a `LAYER.AddConnect()`. Either the text `TYPE` name or a variable can generally be supplied to `AddConnect()` to specify a column, layer, or cell group connection point. In the case of a `COLUMN` or `LAYER` connect, partial information can be supplied in the `AddConnect()` request and Brainlab will guess the connection point. The guesses are made as follows:

1. If there is a designated output or input layer and cell group (see Section 2.4), that will be chosen as the source or destination of a connection.
2. If there is only one candidate layer or cell group, then that will be used.

In the following example, we make a connection between two cell groups of a layer, between two cell groups in different layers, and then a connection between two cell groups in two different columns. We use a variety of different addressing formats, sometimes a tuple giving several parts of an address, and sometimes only giving one part of an address if the target is not ambiguous. The only restriction in tuple addressing is that the items in the tuple be in descending order of specificity (starting with column, then layer, then cellgroup, then compartment) and starting at the lowest level that is not implied by the choice of which object's method is being used (so for a layer to layer connect in a column, the address tuple would never contain the column name since the `AddConnect()` method being used is the column's).

```

syms=lib['syntypes']          # get the synapses from the library
esyn=syms['E']                # a standard excitatory synapse
# Connect this cell group to itself (a within-layer connection).
# We do this using the layer's AddConnect() method
l1.AddConnect(ecell, ecell, esyn, prob=.1)
# Connect this cell group to a cell group in another layer.
# We do this using the column's AddConnect() method
c.AddConnect((l1, ecell), i1, esyn, prob=.1)

# create another cloned column named 'c2' to connect to:
cols=lib['cols']
c2=b.Copy(cols, c, 'c2')

# Connections between columns use the brain's AddConnect() method:
b.AddConnect((c, l1, ecell), (c2, l1, ecell), prob=.3)

```

2.4 Designated input/output layers and cell groups

If a LAYER object has the key `_INPUT_LAYER` set in its parameter dictionary, then it will be given preference as a point of input into a COLUMN that contains that LAYER if the connection point is not fully specified. This allows the designer to specify minimum information in CONNECT requests to Brainlab. Similarly, the presence of an `_INPUT_CELLGROUP` parameter in a cell group gives it preference as a point of input into a LAYER. The `_OUTPUT_LAYER` and `_OUTPUT_CELLGROUP` work analogously for output. (Note that these parameters are stored in the same parameter dictionary as the NCS keyword parameters, but the presence of the leading underscore indicates they are for internal use and they will be ignored during the `.in` file creation process, rather than causing an NCS keyword line to be emitted as is the case for normal parameters.)

2.5 Stimulus and report requests

To define a stimulus input for a network using Brainlab, one approach is to create a `STIMULUS` class and set the NCS keyword parameters as desired. Then create a `STIMULUS_INJECT` object that references the `STIMULUS` object just created. As always, the parameters desired for an object can be passed using the `parms={}` keyword argument during object creation, or items in the `parms` dictionary can be set after the object is created. Both techniques are shown in this example:

```
s=b.STIMULUS(parms={'MODE': 'VOLTAGE'})
# add more parameters here to construct the report as desired

si=b.STIMULUS_INJECT()
si.parms['TYPE']='demo-stiminj'
si.parms['STIM_TYPE']=s          # reference to the stim created above
# only the STIMULUS_INJECT is then added to the brain:
b.AddStimInject(si)
```

If the `TYPE` parameter is not set for either a `STIMULUS` or `STIMULUS_INJECT` object, an automatic name with a sequential number appended will be assigned, since NCS requires a `TYPE` to accomplish the reference from `STIMULUS_INJECT` to `STIMULUS`. In a Brainlab script, however, it is common practice (and quite convenient) to leave the `TYPE`s undefined and do the reference from `STIMULUS_INJECT` to `STIMULUS` by using the variable, rather than the text `TYPE` name. Brainlab will handle creating the underlying text reference when the `.in` file is created.

Since many `STIMULUS/STIMULUS_INJECT`s follow a common usage pattern, Brainlab provides several convenience methods in the `BRAIN` class to speed up the process. `AddSimpleStim()` automates several aspects including the creation of linked `STIMULUS/STIMULUS_INJECT` objects:

```

# c is a column object previously defined in the earlier example.
# Since c has more than one layer, and no default _INPUT_LAYER
# was specified when the column was created, we pass in a (col, lay)
# tuple for the target address of the stimulus.
# 'v' means voltage stimulus
b.AddSimpleStim('demo', (c, es), 'v', ampstart=.02, dur=(.1, .5))

```

Another common need is to apply a large number of individual voltage or current spike stimuli at certain precise times. This is readily accomplished using Brainlab's `AddSpikeStim()` convenience method. In Section 3.1 we consider a real-world example that makes use of that method, where the experiment required constructing many distinct input patterns, each comprising dozens of spikes across a group of about ten cells for a duration of about 300 milliseconds.

Report requests are treated in an analogous manner: create a `REPORT` object manually, set the NCS keyword parameters as desired, and add the object to the `BRAIN`. Or, use one of the convenience `BRAIN` class methods to speed up the process. Refer to the `brainlab.py` online documentation[3] for complete options.

2.6 Simulating a network model

Since printing a `BRAIN` object outputs the NCS `.in` file for that model, one way of running a simulation is to save that output to a file and then manually invoke NCS using that file as input. Saving the Brainlab output to a file can be done in the script itself, like this:

```

f=open('simplebrain.in', 'w')
f.write('b')
f.close()

```

(The backquotes around the `BRAIN` object `b` are the Python idiom for “convert to string” and they invoke the `__repr()` method for the object.) Or, if a Brainlab script prints out the brain, simple shell redirection can be used to divert the output

into a file for use by NCS. When using this technique, make sure that the Brainlab script in question does not also print out any extraneous information in addition to the model itself, because that will cause NCS to give a parsing error trying to load the file.

A more convenient technique is to use the `Run()` method of the `BRAIN`. This allows the model to be simulated within the context of the script's execution flow. When the simulation is complete, control returns to the script. The script can then examine the results of the run (see Section 2.8) and make some decision based on those results, or the script could simply adjust some model parameter(s) and simulate again. The `Run()` method takes a number of optional arguments. Passing `verbose=True` causes extra information about the remote job invocation to be printed. Passing `showprogress=True` causes incremental progress messages to be printed. Passing `nprocs=<n>` invokes the job on the indicated number of compute processor nodes. Passing `procnum=<n>` causes the first processor in the set of processors for the job to be the indicated processor number. (Note that in some installations, job queueing system may prevent some of these features from working as described. The `nprocs` parameter should work in almost all installations, however.) See the online documentation[3] for all options.

`Run()` will return `True` if the job run was successful, or `False` or a Python exception if it was not. It is a good idea to invoke `Run()` within a `try...except` block if the invoking script hopes to catch errors and retry. Status for a run is saved into a file named `<brainname>-nlog.txt`. Examining the file will often give clues as to why a job failed.

Note that even if the `Run()` method is used, it is often very helpful to save the `.in`

file for later examination. Certainly this should be done at least a few times when working with a new model, so that spot checks can confirm that the model is built as you think it should be.

2.7 Viewing a network model in three dimensions

Though most network models are too complex to readily understand visually, it is occasionally useful to try. Sometimes viewing a three dimensional rendering of a network model will expose some obvious error, such as a completely disconnected column or group of columns. Brainlab provides a module called `netplot.py` that allows interactive three dimensional rendering of a network model.

In NCS, cell locations within a column and connections from cell group to cell group are defined probabilistically in the `.in` file. Actual cell spatial locations and synaptic connections are not made until the model is read in by NCS for simulation. Special options in the `BRAIN` section of the `.in` file (`OUTPUT_CELLS` and `OUTPUT_CONNECTMAP`) instruct NCS to save the cell location and synaptic connection information to text files once it has been defined. This information is used by Brainlab's `netplot.py` module to construct a three dimensional model for display.

Location information should be provided during creation of the model with Brainlab, otherwise the resulting display will probably not be helpful. Brainlab provides default location information if none is supplied, but the resulting layout is usually poor. Location information is stored in parameters with a leading underscore character in front of the normal NCS location keyword. The rationale for the leading underscore is simply that in NCS, the location data is stored in `SHELL` structures rather than the `COLUMN` or `LAYER` blocks themselves. Brainlab does not require the

user to keep track of the `SHELL` blocks, since they map directly to the underlying structures, one for one, and Brainlab generates them only when the model is converted to a `.in` file. So a sample usage would look like this:

```
x, y, w, h=(100, 200, 50, 80)
c=b.COLUMN({"_WIDTH":w, "_HEIGHT":h, "_XLOC":x, "_YLOC":y})
b.AddColumn(c)

l1=b.LAYER({"TYPE":"lay9", "_UPPER":90, "_LOWER":10})
l1.AddCellType(ecell, 10)
c.AddLayerType(l1)
```

The `netplot.py` model viewer can be invoked on saved `<modelname>.cells.dat` and `<modelname>.synapse.dat` files from a prior run. Or, `netplot.py` features can be invoked in a Brainlab script using the BRAIN's `ThreeDPlot()` function. Remember that in any case, the `netplot.py` command must be executed after the model has been run through NCS, either on the command line or with the `BRAIN.Run()` method. This is necessary to create the `.dat` files that contain the connection information. Just as with the report plotting utilities, local `.dat` data files with the appropriate names will be used if they exist, otherwise Brainlab will attempt to fetch them from the remote execution directory. If they do not exist in the remote execution directory, the attempt will fail.

The `netplot.py` supports two plotting modes: a minimally decorated version that uses Bezier curves for synaptic connections and pyramids for cells (see Figure 2.1), and a nicer looking version that uses shading, `PolyCylinders` from the `GL Extrusion` library for axons and dendrites, more interesting shapes for the neurons, and several other enhancements (see Figure 2.2).

Refer to the `netplot.py` online documentation[3] for complete options.

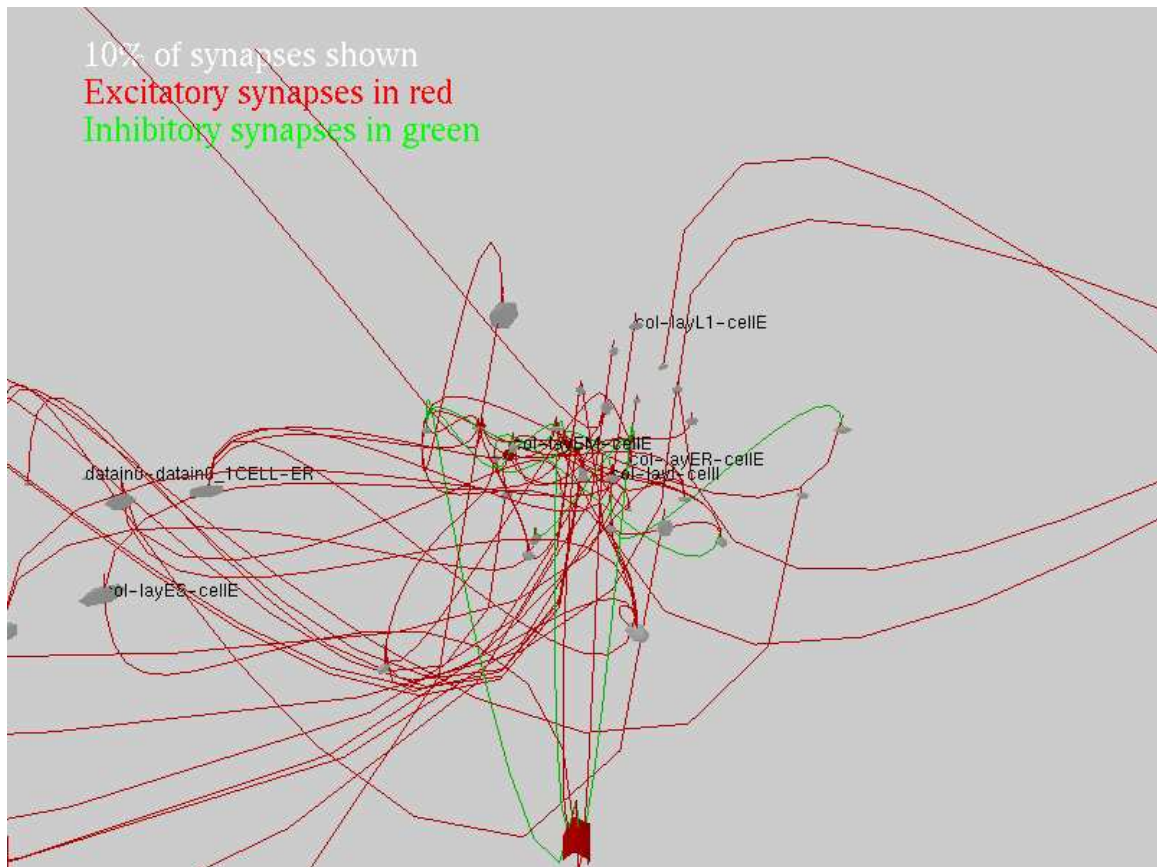


Figure 2.1: An example of the `netplot.py` module of Brainlab, which allows interactive 3D examination of a network model. This screen capture shows the low-realism version of the display, which is most efficient for large network models or older display hardware with poor 3D hardware acceleration.

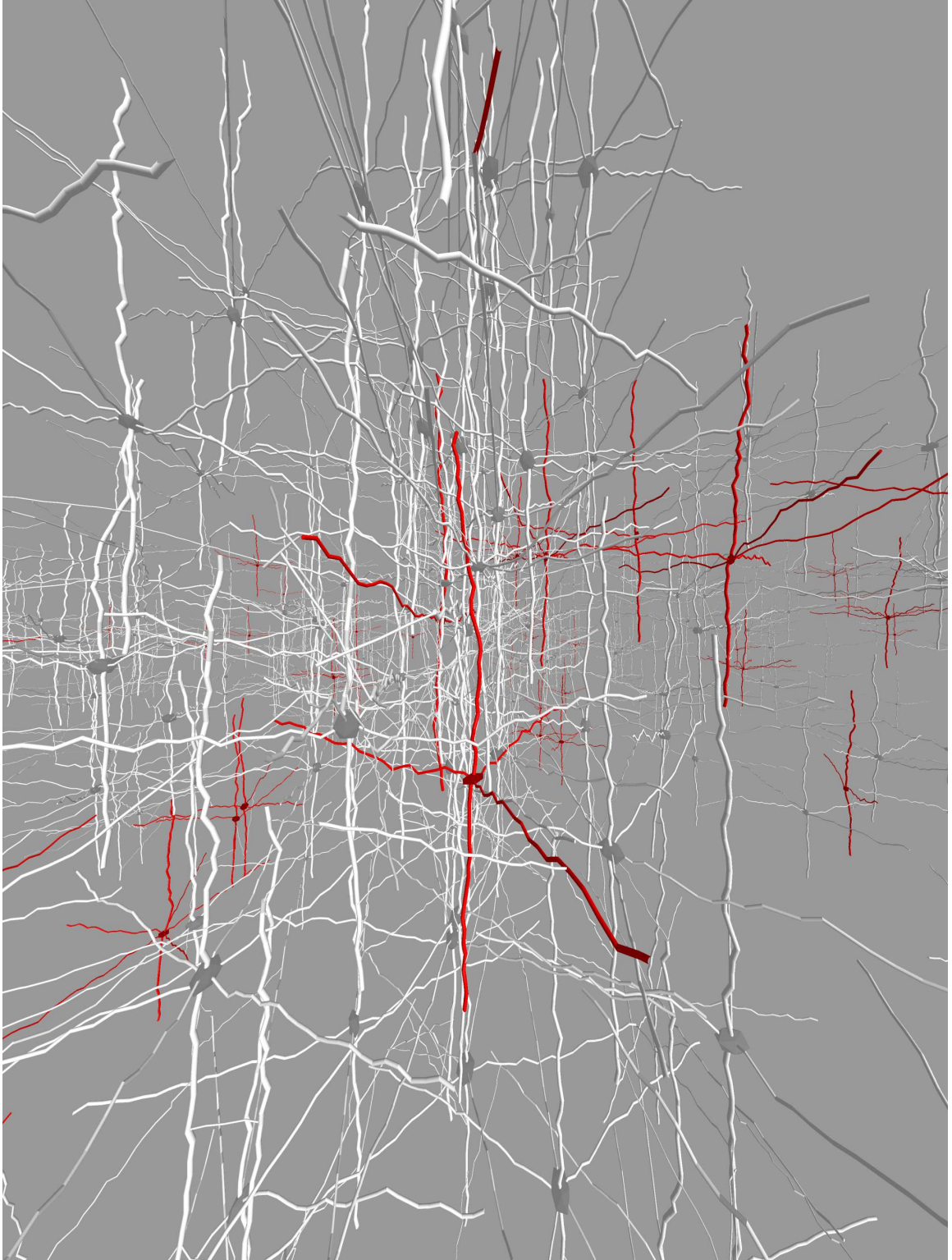


Figure 2.2: An example of the realism mode of Brainlab's `netplot.py` module. This allows 3D interactive exploration of models defined in Brainlab.

2.8 Data loading utilities

Often a Brainlab script will examine the report output of an NCS run. (This may be in place of, or in addition to, human examination of reports or graphs of data.) Brainlab provides a number of utility functions to make loading NCS report data into scripts easier and more efficient. Note that these are functions within the `brainlab.py` module, and not methods within the `BRAIN` class itself.

The `LoadSpikeData(brainname, reportname)` function takes two mandatory arguments, the name of the brain and the name of an NCS *voltage* report, and returns a Python list containing a sublist that holds the spike *times*, given as a floating point number in seconds, for each cell in the report. Several optional arguments allow selection of start and end time ranges, setting the threshold voltage value that defines a spike, and more.

The `LoadReport(brainname, reportname)` function takes two mandatory arguments, the name of the brain and the name of any type of text NCS report. The data is returned in a two dimensional array. When working remotely, files will be automatically copied across the network on demand from the remote work directory to the local working directory, so that further accesses will be local and faster. If only spiking data is required, `LoadSpikeData()` is often considerably faster than `LoadReport()`, since `LoadSpikeData()` transfers only the spike times and not all the voltage information across the network.

For other data loading utilities, refer to the online documentation[3].

2.9 Report plotting utilities

Brainlab provides a number of convenience functions for report plotting, built on the data loading utilities of Section 2.8 and the matplotlib[6] library. Refer to the online documentation[3] for descriptions of all the plot functions and their arguments.

Plots are often highly customized, and frequently several subplots are combined into one larger plot. Brainlab plotting functions are designed to simplify creation of the most commonly desired plots and allow them to be combined together as subplots of a larger plot. For example, the three-part plot in Figure 2.3 was generated with the following code:

```
f=nextfigure()
np=3; pn=1

ts=150000      # start at 15 sec
dd=100000     # go for 10 more sec

subplot(np, 1, pn); pn+=1
ReportPlot("E0442", "EMUSERep", newfigure=False, cols=[75, 80, 85],
           linelab=['synapse 75', 'synapse 80', 'synapse 85'], xlab="",
           xrange=(ts, ts+dd), legendloc='center right', ylab='USE')

subplot(np, 1, pn); pn+=1
SpikePatternPlot("E0442", ["EMRep"], newfigure=False,
                 xlab='', ylab='EM cell number', xrange=(15, 25))
locs, labels=xticks(); xticks(locs, [""]*len(locs))
legend(('Each dot is a spike', ), loc='center right')

subplot(np, 1, pn); pn+=1
ReportPlot('E0442', ['EMRep'], newfigure=False, cols=[5],
           linelab=['EM cell 5'], xrange=(ts, ts+dd),
           legendloc='center right')

savefig("demoplot.ps")
```

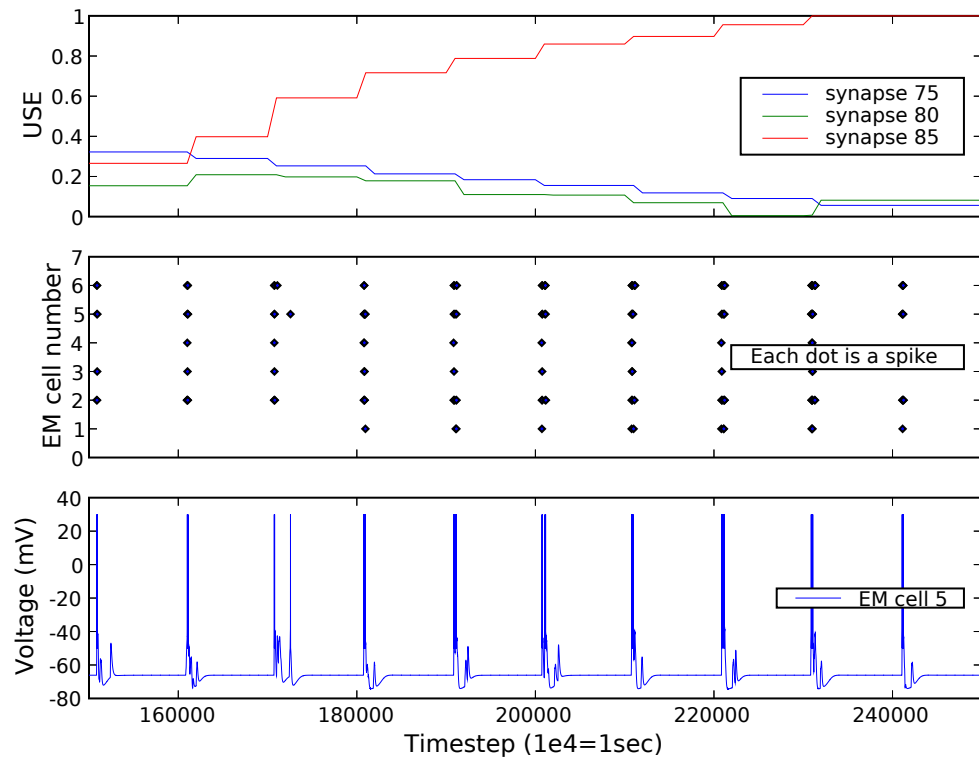


Figure 2.3: An example of three plot types, generated using Brainlab convenience functions, combined onto one master plot. The matplotlib library generates the underlying graphs.

2.10 Experimental features

A few Brainlab features are considered experimental and have not been extensively tested in the current Brainlab version. We mention them here so that if a Brainlab user encounters a need for similar functionality, this existing work can serve as a reference or model.

2.10.1 Areas

Some brain modelers may have a need for a convenient building block at a level higher than column. The **AREA** abstraction provides such a building block. In earlier version of Brainlab, an **AREA** was defined as a two dimensional array of hypercolumns, where each hypercolumn was a two dimensional array of columns. **AREAs** can be interconnected easily with a number of pre-defined connection rules. For example, one connection rule states that columns in one area are connected to the columns that surround the spatially corresponding column in the target area. An earlier version of the Brainlab **AREA** abstraction was used with some success in[17].

2.10.2 Enumerated Columns

The normal way of building models for use with NCS is to define one or more groups inside a layer, each cell group containing some tens of cells, and specifying a probability of connection between those cell groups. NCS then handles the details of creating synapses between individual cells according to the supplied probability. The actual connections are not made until NCS begins the simulation, and the user has no say in which connections are actually made. Some experiments require a higher degree of control, for example the ability to specify synaptic properties (like conductance or USE) for individual synapses. One way of doing that would be to define each cell in

its own column. Then each cell could be addressed by its own column name. However, it may also be desirable to preserve the convenient normal notion of column, which the single-cell columns approach would subvert.

An earlier version of Brainlab supported an “enumerated column” which for most purposes was treated the same as a conventional column in the Brainlab script. However, when the model was converted to a `.in` file, every cell in each layer was assigned a unique cell group name. This permitted each cell to be uniquely addressed in `CONNECT` and `REPORT` and `STIMULUS` blocks. Every synapse could be specified explicitly when needed, and normal column usage preserved.

2.11 Setting up Brainlab

Configuration options for Brainlab are stored in a `.brainlabrc` configuration file in the working directory. Brainlab can be set up for remote operation (where jobs are sent over a network to a computer or cluster) or local operation (where jobs are executed on a local machine, or on a cluster that is managed by the local machine, such as the headnode of a cluster). Future version of Brainlab may support an automatic configuration and testing program to ease the setup process. Future versions may also support different queueing models in addition to the current simple one where jobs are immediately invoked using MPI.

2.11.1 Remote operation

For remote operation, Brainlab makes the following assumptions:

- The local machine has secure shell (ssh) access to the remote machine with no password required (typically done via inserting the local user’s public key into the appropriate `authorized_hosts` file on the remote machine).

- The username on the local machine is the same as the username on the remote machine. (This restriction will probably be eliminated soon.)
- There is a directory on the remote machine that contains the NCS executable, a machine file named `mach` listing the usable cluster machines, and the Brainlab libraries.
- The path of this remote directory is defined with the `remdir=` option in the config file.
- The hostname of the remote machine is given in the configuration file using the `remotemachine=` option.
- The option `remoteexec=True` appears in the config file.
- The local working directory also contains the Brainlab libraries.

2.11.2 Local operation

For local operation, Brainlab makes the following assumptions:

- The local working directory contains the Brainlab programs as well as the NCS executable, and a machine file named `mach` listing the usable machines (which may in some cases only be the local machine).
- The option `remoteexec=False` appears in the config file.
- The option `remdir=` appears in the config file and contains the name of the local working directory.

Chapter 3

A detailed example

The real strength of Brainlab is that it allows an experimenter to conduct more complex experiments more conveniently. Real neuroscience modeling experiments often involve multiple complex neural network model variants, complex input stimuli and input protocols, and complex data analysis. Brainlab provides tools to make all of these tasks easier, and equally important, it provides these tools integrated together in one package. This last feature allows convenient automation and encapsulation: the ability to keep all aspects of one experiment together, in one coherent, comprehensible, and experimentally reproducible package: a Brainlab script.

The remainder of this chapter will consider some aspects of a more complex experiment that is the subject of my doctoral research. In that research, I am testing a proposed information processing feature of a cortical microcircuit. The unit of information transfer in that experiment is a spatio-temporal spike train pattern, which is a set of spikes occurring over a set of cells over a period of time. The hypothesis is that the layered cortical microcircuit is remembering correlations of these patterns. Though much of the sample code in the following sections is generic Python, and not Brainlab-specific, the entire example serves to make Brainlab usage more concrete in the context of a real experiment. Only portions of the entire experiment are shown.

3.1 Constructing complex stimulus patterns

The first step in the experiment is to construct the stimulus patterns that will be applied, in a particular sequence, to the cortical microcircuit. We start with a function that creates a simple list of spike times:

```
def RandSpikeList(dur, interspike, starttime=0, type=None, pmean=25):
    s=[]
    if type=='poisson':
        pn=dur*pmean*2
        time=starttime
        time+=float(poisson(pmean)/1000.0)
        while time < (starttime+dur):
            s.append(time)
            time+=float(poisson(pmean)/1000.0)
        return s

    time=starttime
    time+=int((random()*interspike)+10.0)/1000.0
    while time < (starttime+dur):
        s.append(time)
        time+=int((random()*interspike)+10.0)/1000.0
    return s

print "uniform:", RandSpikeList(.100, 25)
print "poisson:", RandSpikeList(.100, 20, type='poisson')
```

This gives us lists like this:

```
uniform: [0.029999999999999999, 0.047, 0.070000000000000007]
poisson: [0.014999999999999999, 0.048000000000000001, 0.078]
```

That defines an input for an individual cell for a time interval. We combine these spike train lists to make a single input *pattern*, which specifies the inputs for a group of cells. Then we will group a number of these patterns into *sets*. In the experiment we are considering, we have two sets, one called “key” and another called “data”. This example code shows the creation of just the “key” set:

```

numkstim=max(keystimpat)+1
#print "numkstim", numkstim
keyinstim=[]      # will contain a stim list for each cell
for i in range(0, len(keyin)):
    tc=[]
    keyinstim.append(tc)
    for j in range(0, numkstim): # for each unique pattern . . .
        sn=RandSpikeList(dur, interspike, starttime=0, type=type)
        tc.append(sn)

```

This results in a set of spike patterns similar to that depicted in Figure 3.1. These patterns are then applied as stimuli according to a stimulus protocol defined in the list `keystimpat`, as shown in the following code fragment.

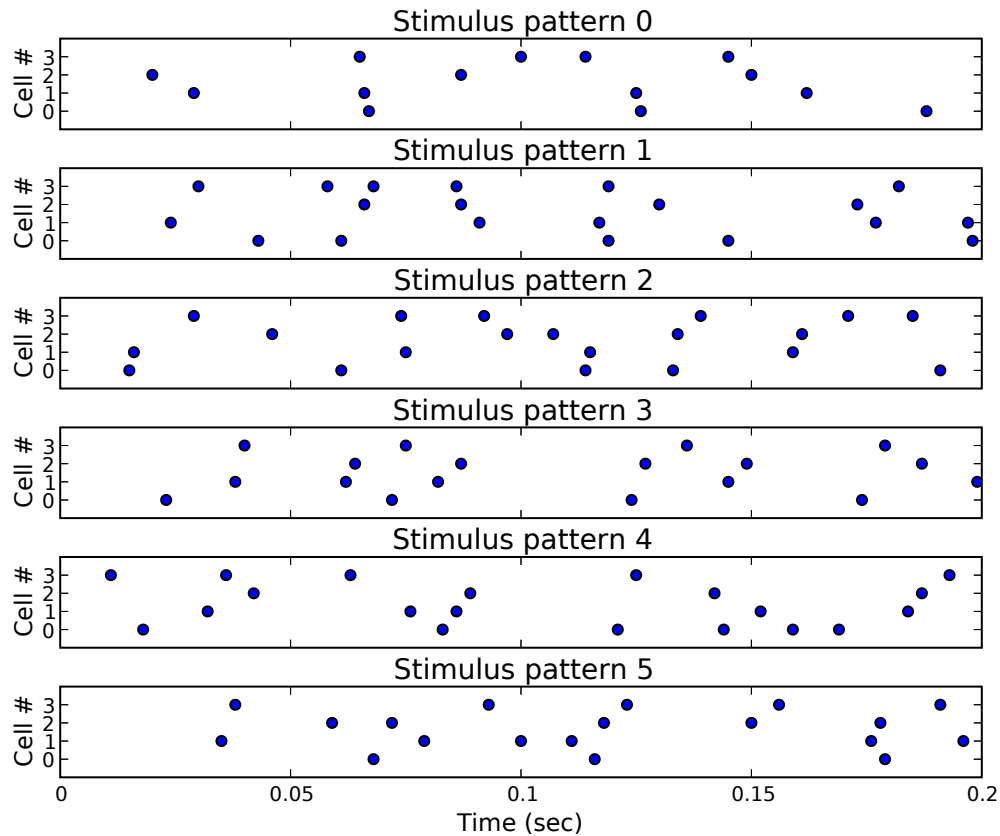


Figure 3.1: A complex spike pattern set.

```

stimstart=0.0
pn=0
for p in keystimpat:      # for each repetition of stim
    # p is the pattern number to apply starting at stimstart
    if p >= 0:           # pattern of -1 means don't apply stim
        for i in range(0, len(keyin)):      # cell in input area
            n=keyin[i]          # n is cell to apply to
            s=keyinstim[i][p]    # s is list of spikes for cell
            ns=[v+stimstart for v in s] # shift spike by stimstart
            b.AddSpikeStim(n, ns, stimname="skey%d-%d" % (i,pn))
            pn+=1
        stimstart+=stiminterval

```

The `AddSpikeStim()` takes a list of spike *times*, in seconds, and ensures that appropriate `STIMULUS` blocks to cause a spike at the desired times. (There are actually two underlying mechanisms Brainlab can use to accomplish this: creating a separate `STIMULUS` block with a pulse applied at each desired time, or creating a single `STIMULUS` block that references a file with appropriate data. Which of the two underlying mechanisms is used should be transparent to the user and they can be interchanged readily, but providing that the number of spikes to be applied does not number more than about 100,000, the first method is preferred since no external files are required.)

3.2 Genetic search of model parameters

Models are based on real-world parameters, but there is invariably some imprecision in the value of these parameters. When imprecision is present in each of a set of a dozen model parameters the result can be a nonfunctional or unbalanced model. Consequently, sometimes an experimenter must conduct a search for a model with certain properties by varying a number of the parameters and trying out the model, and then trying again. Brainlab provides an easy platform for such a search, and also

a convenient way to use a particularly powerful kind of search: genetic search. In the following example, we create a chromosome that contains a set of brain parameters and then launch a genetic search using the SciPy[11] package's genetic algorithm module. First we set up the chromosome and then start the genetic search:

```

genelist=[]
genelist.append((ga.gene.list_gene([2,3,4,5,6,7,8]), 'ntapecells'))
genelist.append((ga.gene.float_gene((.005, .05)), 'allmaxcond'))
genelist.append((ga.gene.float_gene((.005,.3)), 'allF'))
genelist.append((ga.gene.float_gene((.005,.3)), 'allD'))
genelist.append((ga.gene.float_gene((.005,.3)), 'poshebbdeltause'))
genelist.append((ga.gene.float_gene((.005,.3)), 'neghebbdeltause'))
genelist.append((ga.gene.list_gene([(float(x)/1000.0) \
                                for x in range(200, 801, 100)]), 'stimdur'))
genelist.append((ga.gene.list_gene(range(20,120,10)), 'interspike'))
genelist.append((ga.gene.float_gene((.05, 1.0)), 'eresprob'))
genelist.append((ga.gene.float_gene((.05, 1.0)), 'eremprob'))
genelist.append((ga.gene.float_gene((.05, 1.0)), 'l1emprob'))
genelist.append((ga.gene.float_gene((.05, 1.0)), 'esthalprob'))
genelist.append((ga.gene.float_gene((0.0, 1.0)), 'L1L5prob'))
genelist.append((ga.gene.float_gene((0.0, 1.0)), 'L6L4prob'))

all_genes=[]
for (g, parmname) in genelist:
    all_genes+=g.replicate(1)
    mychrom.append(parmname)

this_genome.performance=EOfitness
gnm=this_genome(all_genes)
pop=ga.population.population(gnm)
pop.evaluator=my_pop_evaluator()
galg=ga.algorithm.galg(pop)
settings={'pop_size':16,'p_replace':.8,'p_cross':.8, \
          'p_mutate':'gene', 'p_deviation': 0.,'gens':64,\
          'rand_seed':0,'rand_alg':'CMRG'}
galg.settings.update(settings)
galg.evolve()
print "best:", galg.pop.best()

```

Note that in this example, some items in the chromosome are lists and some are values within floating point ranges. The fitness evaluation function here is called

EOfitness(). It is given to the genetic algorithm as the performance method. The fitness evaluation function looks like this:

```
def EOfitness(self):
    score=0.0
    # get the chromosome values for this individual
    actual=self.get_values()
    allbd=0; allad=0; allbk=0; allak=0; anumposs=0
    # to get a better picture of how good a parameter set is,
    # run the model this many separate times to determine fitness
    numruns=2

    for rn in range(0, numruns):
        brainname="E0_"+'en'
        # get actual settings for this individual for all elements
        # in chrom and stuff them into the brain parms, so brain
        # we instantiate will use our evolved values
        parms={}
        pn=0
        global mychrom
        for parmname in mychrom:
            parms[parmname]=actual[pn]
            pn+=1
        success=False
        while not success:
            try:
                (bd,ad,bk,ak,numposs)=E0(parms=parms)
                success=True
            except:
                print "got exception, will retry simulation"

        print "results for this trial:  %d %d %d %d %d" \
              %(tn, rn, bd, ad, bk, ak, numposs)
        allbd+=bd; allad+=ad; allbk+=bk; allak+=ak
        anumposs+=numposs

    print "overall results:  allbd, allad, allbk, allak", \
          allbd, allad, allbk, allak, anumposs
    # create the overall fitness by combining results from all runs
    # of this individual:
    anumposs=float(anumposs)
    fitness=float(allad + allak)/anumposs
    return fitness
```


This function in turn invokes `E0()`, which handles the stimulus preparation (as outlined in Section 3.1) and invokes the `BRAIN`'s `Run()` method to perform the simulation. Our experiment involves matching output patterns, and that must be done to determine the model's fitness. Function `E0()` handles this work as well, and that is the subject of Section 3.3.

3.3 Automated data analysis

In the experiment, we gather output reports from the NCS run and then compare sets of spikes over groups of cells over a predefined period. The result is a metric for how closely spatio-temporal spike patterns match. A depiction of the result of such a comparison is in Figure 3.2. This plot was produced using conventional Python list processing techniques and matplotlib scatter plots. Based on the results of the matching, the genetic algorithm selects a new parameter set for each member of the population and Brainlab makes adjustments to the model parameters for the next round of simulations.

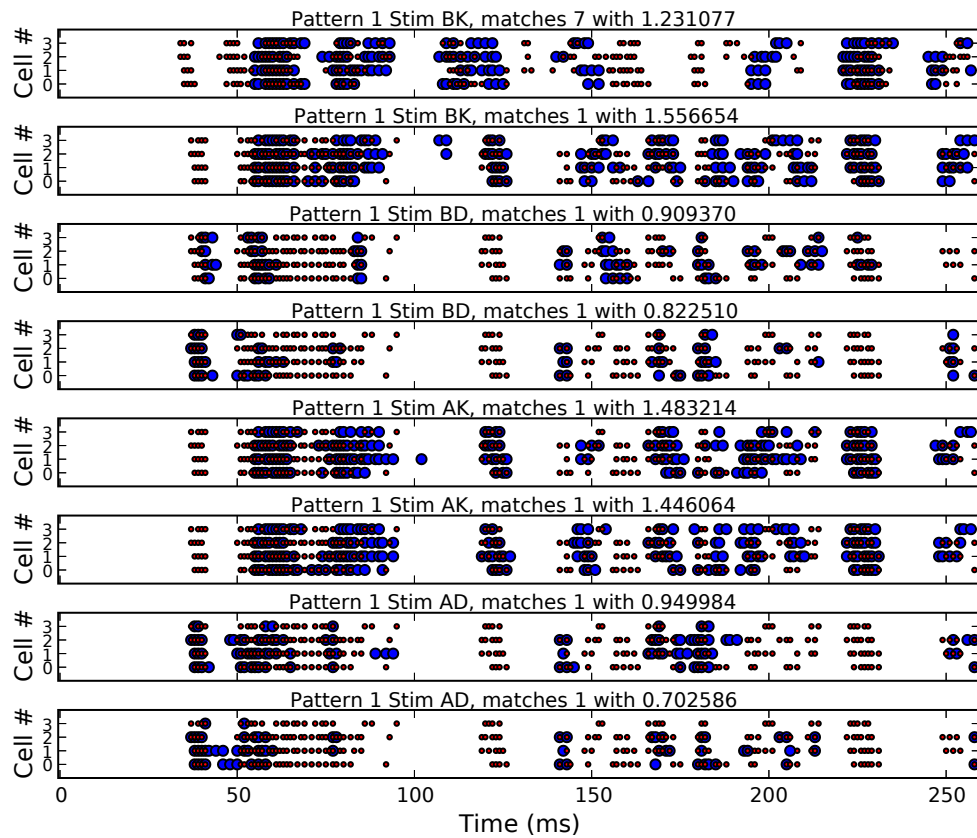


Figure 3.2: A spike pattern match plot. Larger circles comprise a pattern that is to be matched, and the overlaid smaller circles comprise the pattern that was found to be the closest match.

Chapter 4

Design and implementation issues

Several of Brainlab's design choices deserve specific comment. These include the rationale for the selection of the Python programming language, how Python language features mesh with Brainlab's object oriented design, Brainlab's modular design, and Brainlab's remote execution model.

4.1 Python language features

Python is an open source, cross platform programming language. It is free of charge and can even be used for commercial projects without complications. Supported platforms include Microsoft Windows, MacOS, and other Unix-like operating systems including Linux, FreeBSD, and Solaris. The base Python language is constantly being extended and made more powerful by hundreds of developers working together across the world. In addition to the base language, there are dozens of external packages in various states of development, from polished to prototype. These packages gradually move into the base distribution as they mature and if they are of sufficiently wide interest.

Python is ordinarily compiled into bytecode and the bytecode is then interpreted by a runtime engine. This is the same approach used by Java. It results in code

execution that is generally faster than interpreted code. There is also an excellent just-in-time compiler called Psyco[2] that converts a Python program's bytecode into directly executable machine language, on the fly, during execution.

Python is dynamically typed rather than statically typed, like C++ or Java. Dynamic typing is extremely convenient for the programmer, though it may come at some small performance cost.

Python is well known as an extremely clean and easy to read and understand language. The most controversial aspect of Python is that indentation is used to determine logical block level, rather than, for example, special tokens like the curly braces of C, Perl, and Java or the `begin...end` blocks of MATLAB. The practical consequence of Python's choice is simply that programs cannot be improperly indented. Anyone who has spent time trying to understand a MATLAB program where the logical blocks do not correspond to indent levels (which seems to be a very common state of affairs in MATLAB code, for whatever reason) will appreciate Python's choice. There may be some mild inconvenience in moving a block from one indentation level to another in a Python program, but that is seldom an issue with modern editors.

4.2 Python scientific community

Python has a large number of support library packages to make scientific computation more efficient and powerful. Some of these packages are used in Brainlab, including:

- matplotlib[6], a MATLAB-like plotting package
- PyOpenGL[10], OpenGL bindings for Python
- numarray[9], MATLAB-style array processing

- SciPy[11], a set of scientific tools for Python including a genetic algorithm module

One factor in selecting a language for a project is how many other people are already using it for related purposes, since that affects the availability of tools and add-on packages and also the likely interest among potential users. Based on Internet activity, the Tiobe Programming Community Index[1] estimates that the overall Python programming community is 5 to 15 times as large as MATLAB's. However, only a fraction of Python's programmers, perhaps 5 or 10%, are using Python as a tool for scientific research. The scientific community seems to be recognizing that the ease of use and scalability offered by Python and interest appears to be growing.

4.3 Python object orientation and Brainlab

Unlike many other languages where object orientation was added as an afterthought (MATLAB, C, and Perl for example), Python was conceived from the outset as an object oriented language. This is an important factor for managing large experiments. (Python can also be used in a conventional, non-object-oriented fashion if desired, which is convenient for small projects.) Python's object orientation features are used extensively in Brainlab.

The BRAIN, COLUMN, LAYER, and other objects are all implemented as Python object classes. The `__repr__()` method for each object is overridden so that printing an object results in text for that object in a format suitable for inclusion in the NCS `.in` file. In the case of a lower-level object, this method just prints out the object itself, but does not print any other objects that are referenced by the object being printed. The BRAIN object's `__repr__()` method, however, first recursively traverses

the entire tree of objects referenced from the BRAIN object and a list is composed for each type of referenced object. Once all referenced objects have been collected together, the entire NCS `.in` file is printed, starting with the BRAIN section, and proceeding to all of the other sections of the `.in` file in the conventional order.

The lower-level classes are implemented as nested classes within the BRAIN class. Note that they are not derived subclasses, but rather nested classes. Derived subclasses are appropriate where the subclass has most of the aspect of the superclass but some additional features. In Brainlab the nested classes are not logically subclasses of the BRAIN since they do not share the same characteristics as the super-object but are merely contained by it. However, the lower-level classes do need access to the component type libraries that are stored with the BRAIN class. If the lower-level objects were entirely separate classes, they would not have convenient access to the component type libraries. By making the lower-level classes nested within the BRAIN class, they do have that access.

4.4 Brainlab modules

Brainlab consists of three functional modules (files):

- `brainlab.py`: This module is the only module that needs to be directly imported by user programs. The `brainlab.py` module imports all other modules necessary for using Brainlab. The `brainlab.py` module itself consists of convenience functions for loading NCS report data, plotting common types of graphs of NCS data using the matplotlib library, running simulations on a remote cluster computer, and a few other functions. Refer to the online documentation[3] for a complete list. Knowledge of the execution environment (host name of the

compute cluster for NCS jobs, username for invocation of NCS jobs, and so on) is restricted to this file.

- `brain.py`: This module defines the `BRAIN` class and the lower-level, nested, brain modeling classes including `COLUMN`, `LAYER`, `CELL` and so on. The responsibility of this module is the creation of an object oriented brain model and converting it to a NCS `.in` format file for simulation.
- `netplot.py`: This module handles the three-dimensional graphical display of neural networks.

4.5 Brainlab remote execution

Brainlab is designed primarily to run on the user's workstation, and send jobs across a network to be simulated on a different computer (or cluster). There are several reasons for this focus. The user has more control over the software installed on a personal workstation than on a typical group or departmental compute server or Beowulf cluster, where it may be difficult to get installed the libraries necessary to run Brainlab. Often data will be analyzed repeatedly, displayed and analyzed in a variety of ways, and that is best done on a personal workstation so that specialized tools are guaranteed to be available and also so that other users of the simulation environment will not be affected. Also typically a personal workstation will have high-performance display hardware that will work more efficiently with extensive graphing, perhaps in three dimensions.

Since NCS reports are gathered on the compute cluster, Brainlab employs a smart caching strategy to move those files, as necessary, to the workstation. When a user requests a plot using NCS report data, the necessary files are copied on demand from

the compute cluster. Further references to those files are made locally. If the user is only interested in spike information rather than the full voltage information in a report, that spike information is extracted from the full reports on the cluster before the distilled file is copied back to the workstation automatically by Brainlab. This results in a significant speedup.

Brainlab can also run directly on the machine where NCS does the simulation.

Chapter 5

Thoughts on a next generation NCS and future work

At this point, Brainlab is functional and has been used in my own experiments quite extensively. Another member of our lab has made limited use of earlier versions of the toolkit. A new experiment by another lab member has begun using Brainlab, and members of an external laboratory have also expressed interest in using it. If NCS becomes more widely used in other laboratories, we expect interest in Brainlab will also increase. This user interest will largely drive the further evolution of Brainlab. However, we have some thoughts on where future development might take Brainlab.

5.1 A next generation NCS

NCS is a complicated program, and even though the transition from version 3 to version 5 resulted initially in a net simplification for a time, making changes to NCS gets increasingly more complicated as time goes on and features are added. A surprising amount of this complexity seems to arise from the simulation engine attempting to manage more information about the model than it strictly needs to simulate the network. Right now, the simulation engine has a fairly complicated file parsing module that reads the `.in` input file and builds a *flat* representation of this

model in memory based on that (this is called the `GCList` representation in NCS). The `.in` input file has some hierarchical structure including notions of columns and layers and cell groups, as we have seen. In practice though, users often end up subverting the intent of these structures, by, for example, compressing all cell groups into a single layer of a column, for simplicity in creating the model. However, the final memory representation of the network that is used for the simulation in NCS is flat.

This leads to a thought: what if NCS were stripped down so that it only contained the portions necessary to simulate the simpler, flatter, memory representation of the model? Brainlab, or something like it, could handle the model building and present a flat network, in memory, to NCS. Some benefits of this would be:

- The NCS code would be simpler with higher level biological structures removed. The cell and channel and synapse logic would have to remain, but information about layers and columns could be eliminated.
- The NCS code would be simpler with `.in` file parsing logic removed. As we have argued in this thesis, that complexity is not worth much anyway since the `.in` file is not a directly usable representation for complex modeling anyway.
- NCS would be more easily used for simulating general spiking neural networks that do not conform to column/layer structure. Much synfire chain research[14, 15] and liquid state machine research[23, 24] utilizes homogenous spiking networks without explicit biological structure above the level of neuron and synapse. Such networks can be simulated with the current version of NCS, but only by expending some effort to step around the enforced column/layer paradigm of the NCS `.in` file.

- Many new biological features could be implemented without touching the NCS C code. That means they could probably be done more easily and by less technical users. For example, recent work in our laboratory involved implementing reciprocal connections in the NCS C code. This required a fairly significant investment in coding and testing time. It is likely that in the proposed arrangement, this work could be done entirely in a Brainlab-like environment in a high-level scripting language where it could be done more quickly and flexibly.

A counterargument might point out that the simulation engine benefits from hints derived from the column/layer structure of the `.in` file, so that highly interconnected portions of the network can be placed onto individual nodes to increase simulation efficiency by minimizing network traffic. However, since the simulation engine has the full map of connections, it could perhaps make better decisions about allocating portions of a network across the compute resources based on that rather than based on the structure that the `.in` file would provide. Or if necessary, hint information could be passed from the modeling portion (Brainlab), based on the known biological structure of the model, to the simulation portion (NCS) along with the flat model network map.

5.2 Future work

The next minor version of Brainlab will incorporate a number of enhancements. These enhancements will minimally affect the current API so that current experiment scripts will continue to run with few or no changes. Some of the planned changes include:

- A few `BRAIN` class methods will become `brainlab.py` functions. In particular,

`BRAIN.Run()` will likely become deprecated, in place of `brainlab.Run(BRAIN)`. The motivation for this is that logically, the brain modeling portion of Brainlab (`brain.py`) should not have any dependencies on the broader simulation and data analysis tools (provided in `brainlab.py`.)

- Data loading and report plotting functions will be cleaned up and standardized to accept a new, general, cell addressing scheme that is currently implemented in only some functions.
- Simulation jobs may be executed in subdirectories automatically, or at least as an option. This should make it easier for an experimenter to maintain large sets of files for an experiment and reduce directory clutter in the working directory.
- Support for different queueing systems in remote execution environments will probably be enhanced. The current system makes rather broad assumptions about job execution capabilities on the remote compute cluster. This change to Brainlab will probably coincide with the onset of use of a queueing system on our own compute cluster.
- AREA and enumerated column support will be retested to ensure that they work with the current Brainlab code base.
- Some convenience functions for executing multiple parallel jobs may be added. This is possible in the current environment but requires some knowledge of Python to implement; the basic procedure is to use Python threading to invoke each `BRAIN.Run()` in a separate thread context. A Brainlab convenience function could make this commonly-desired feature accessible to a larger class of users.

We are hopeful that the use of NCS will continue to grow beyond our own lab, and that Brainlab can grow with it. We fully expect and hope that many future features will be demanded by an eager group of users conducting experiments of their own.

Bibliography

- [1] Tiobe Programming Community Index, a ranking of language popularity based on web search. http://www.tiobe.com/tiobe_index/index.htm.
- [2] Psyco, a just-in-time compiler for Python. <http://psyco.sourceforge.net/>, 22 April 2004.
- [3] Brainlab documentation. <http://www.interstice.com/drewes/brain/brainlabdoc/>, 22 April 2005.
- [4] GENESIS, neural network simulator. <http://www.genesis-sim.org/GENESIS/>, 22 April 2005.
- [5] Goodman Brain Computation Lab website. <http://brain.cse.unr.edu>, 22 April 2005.
- [6] matplotlib, MATLAB-like plotting extensions for Python. <http://matplotlib.sourceforge.net/>, 22 April 2005.
- [7] MPI, parallel programming library. <http://www-unix.mcs.anl.gov/mpi/>, 22 April 2005.
- [8] NEURON, neural network simulator. <http://www.neuron.yale.edu/neuron/>, 22 April 2005.
- [9] numarray, MATLAB-like array processing library for Python. http://www.stsci.edu/resources/software_hardware/numarray, 22 April 2005.
- [10] PyOpenGL, Python OpenGL binding. <http://pyopengl.sourceforge.net>, 22 April 2005.
- [11] SciPy, scientific tools for Python. <http://www.scipy.org/>, 22 April 2005.
- [12] University of Nevada, Reno website. <http://www.unr.edu>, 22 April 2005.
- [13] UNR Biomedical Engineering Program website. <http://www.unr.edu/bme/>, 22 April 2005.
- [14] M. Abeles. *Corticonics*. Cambridge University Press, Cambridge, UK, 1991.

- [15] M. Abeles, G. Hayon, and D. Lehmann. Modeling compositionality by dynamic binding of synfire chains. *J Comput Neurosci*, 17(2):179–201, 2004.
- [16] J.L. Blake and P.H. Goodman. Speech perception simulated in a biologically realistic model of auditory neocortex. *Journal of Investigative Medicine*, 2004.
- [17] Rich Drewes, James Maciokas, Sushil J. Louis, and Philip Goodman. An evolutionary autonomous agent with visual cortex and recurrent spiking columnar neural network. In *Proceedings of the 2004 Genetic and Evolutionary Computing Conference (GECCO 2004)*, volume 3, pages 257–258, 2004.
- [18] James Frye. Parallel optimization of a neocortical simulation program. Master’s thesis, University of Nevada, Reno, December 2003.
- [19] P. H. Goodman. NCS (NeoCortical Simulator project) homepage. <http://brain.cse.unr.edu>, 22 April 2005.
- [20] P.H. Goodman, E.C. Wilson, J.B. Maciokas, F. C. Harris, Jr., S.J. Louis, A. Gupta, and H.J. Markram. Large-scale parallel simulation of physiologically realistic multicolumn sensory cortex. Tech Report 01-01, <http://brain.unr.edu/publications/goodmanNIPS01final.pdf>, 2001.
- [21] James King, Philip Goodman, Frederick C. Harris, Jr., James Maciokas, Rich Drewes, James Frye, Christine Wilson, and Jake Kallman. NCS User Documentation. <http://brain.cse.unr.edu/ncsDocs/ncsCode5/index.html>, 22 April 2005.
- [22] James G. King. Brain Communication Server: A Dynamic Data Transferal System for A Parallel Brain Simulator. Master’s thesis, University of Nevada, Reno, May 2005.
- [23] W. Maass, T. Natschläger, and H. Markram. A model for real-time computation in generic neural microcircuits. In *Proceedings of NIPS 2002*. MIT Press, 2002.
- [24] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.
- [25] J. C. Macera, P. H. Goodman, F. C. Harris, Jr., R. Drewes, and J. Maciokas. Remote-neocortex control of robotic search and threat identification. *Robotics and Autonomous Systems*, 46(2):97–110, February 2004.
- [26] Juan Carlos Macera. Design and Implementation of a Hierarchical Robotic System: A Platform for Artificial Intelligence Investigation. Master’s thesis, University of Nevada, Reno, December 2003.
- [27] James B. Maciokas. *Towards an Understanding of the Synergistic Properties of Cortical Processing: A Neuronal Computational Modeling Approach*. PhD thesis, University of Nevada, Reno, August 2003.

- [28] J.B. Maciokas and P.H. Goodman. Spike-timing-dependent-plasticity model of bimodal (audio/visual) processing. Tech Report 02-04, http://brain.unr.edu/publications/mg_bimodal_02.pdf, 2004.
- [29] J.B. Maciokas, P.H. Goodman, and J.L. Kenyon. Accurate dynamical model of interneuronal gabaergic channel physiologies. Tech Report 02-05, http://brain.unr.edu/publications/mgk_channels_02.pdf, 2005.
- [30] H. Markram, J. Lubke, M. Frotscher, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275(5297):213–5, 1997.
- [31] H. Markram and M. Tsodyks. Redistribution of synaptic efficacy between neocortical pyramidal neurons. *Nature*, 382(6594):807–10, 1996.
- [32] B. Opitz and P. Goodman. *Journal of Investigative Medicine*, (52):01–S1, 2005.
- [33] M.C. Ripplinger, C.J. Wilson, J.G. King, J. Frye, R. Drewes, F.C. Harris, and P.H. Goodman. Computational model of interacting brain networks. *Journal of Investigative Medicine*, 52(1):S155 – S155, January 2004.
- [34] The MathWorks Inc. MATLAB, scientific computation environment. <http://www.mathworks.com>, 22 April 2005.
- [35] M.V. Tsodyks and H. Markram. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proceedings of the National Academy of Science, USA*, 94(2):719–23, 1997.
- [36] K.K. Waikul, L. Jiang, F. C. Harris, Jr., and P.H. Goodman. Implementation of a web portal for a neocortical simulator. In *Proceedings of CATA 2002*, 2002.
- [37] E. Courtenay Wilson. Parallel implementation of a large scale biologically realistic neocortical neural network simulator. Master’s thesis, University of Nevada, Reno, August 2001.
- [38] E. Courtenay Wilson, Phillip H. Goodman, and Frederick C. Harris, Jr. Implementation of a biologically realistic parallel neocortical-neural network simulator. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computation*, March 2001.
- [39] E. Courtenay Wilson, Frederick C. Harris, Jr., and Phillip H. Goodman. A large-scale biologically realistic cortical simulator. In *Proceedings of Supercomputing 2001*, November 2001.