HRLAnalysis: off-line processing of neural spike data.

Installation

At this time only Linux and Mac OS X systems have been tested. To make and install the HRLAnalysis suite you will need the following software installed on your system:

- cmake
- boost and boost python
- python 2.4 or greater
- cppunit

The directory structure will look like

```
HRLAnalysis
├── HrlAnalysis
│   ├── bin
│   ├── lib
│   ├── python
│   ├── results
│   └── test
├── HrlNeuralAnalysis
├── HrlNetworkAnalysis
└── test
    ├── AnalysisData
    ├── AnalysisTest
    ├── burstExample
    ├── data
    │   ├── burstAnalysis
    │   ├── CA1
    │   ├── NEO
    │   ├── Network
    │   ├── OLM_VC_5_sec
    │   ├── Sub
    │   ├── synchrony
    │   └── VOLT
    ├── lib
    ├── python
    ├── scripts
    └── SpikeAnalysisAccess
```

At the top level of the tree create the build directory and move into that

```
>> mkdir build
>> cd build
```

Use cmake to create the build scripts and then compile the suite

```
>> cmake ..
>> make
```

The libraries and binaries will be installed in the hrlAnalysis subdirectory.

Mac OS-X: Mac OS-X uses the *Mach-O* binary format, which is different from the traditional *Elf* Unix format. In the Mach-O format, there is a distinction between *shared libraries* and *loadable bundles*. To enable Python to load the HrlAnalysis libraries as modules, and additional step is required:

```
>> ln -s ../HrlAnalysis/lib/libHrlAnalysis.dylib ../HrlAnalysis/lib/libHrlAnalysis.so
>> ln -s ../HrlAnalysis/lib/libHrlAnalysisData.dylib ../HrlAnalysis/lib/libHrlAnalysisData.so
```

Max OS-X Troubleshooting Note: If you have installed Python using Homebrew, you may run into problems in which the build process has used the system version of Python to build the HrlAnalysis libraries, but when executing the Python tests (and your own scripts), the Homebrew version of Python is used. Either use the system's Python for executing analysis scripts, or tell CMake which Python to build the library with. In our experience, the latter approach does not work well because certain required Python libraries are not available for all builds of Python.

To ensure that the compilation was completed successfully execute the included tests

```
>> cd ../hrlAnalysis/test/
>> ./hrlAnalysisTest
>> python testHRLAnalysis.py
```

A sample analysis can be done by executing runAnalysis.py

```
>> cd ../python/
>> python runAnalysis.py 0 5000
```

This should create two simple plots named CA1.png and CA2.png in the *results* folder

```
>> cd ../results
```

Usage within an HRLSim experiment

When enabled, a configuration file is output based on the setup within the experiment. The following needs to be placed in the user experment .exp file To enable the output generator:

```
OUTPUT_FOR_ANALYSIS = True
```

To output the information on a particular population call the Analyze function on that population. For example

```
// Create the population.
E = build_net.NewPopulation(1400, NeuronKind().SetIzhikevich(a,b,c,d));
// Tell the simulator to create the analysis code for this population.
E->Analyze(std::string("CA1"));
```

Where “CA1” is the name that you would like used for that population. When the simulation is executed a python file will be placed in the Data directory. For this simple example given that Python file will be

```
#
# This file was automatically generated by the hrlAnalysis Software.
#

def getCellGroups():
    cellGroup = []
    cellGroup.append({"name":"CA1","startIdx":0,"endIdx":1399})
    cellGroup.append({"name":"CA2","startIdx":1400,"endIdx":2799})
    return cellGroup

if __name__ == "__main__":
    print getCellGroups()
```

This file can be used by the provided example analysis code or in within your own analysis code.

You also have the option of grouping together populations that are defined consecutively. This will create a single analysis group with cell indexes spanning all of the given model populations. At this time nonconsecutive populations are not supported. To use this feature you would give the same name in the calls to Analyze. For example

```
// Create the populations.
StriatumD1 = build_net.NewPopulation(NUM, NeuronKind().SetInhibitory().SetIzhikevich(a,b,c,d))
    ;
StriatumD2 = build_net.NewPopulation(NUM, NeuronKind().SetInhibitory().SetIzhikevich(a,b,c,d))
    ;
SNr = build_net.NewPopulation(NUM, NeuronKind().SetInhibitory().SetIzhikevich(a,b,c,d));
STN = build_net.NewPopulation(NUM, NeuronKind().SetIzhikevich(a,b,c,d));
GPe = build_net.NewPopulation(NUM, NeuronKind().SetInhibitory().SetIzhikevich(a,b,c,d));
// Tell the simulator to create the analysis code for the entire group.
StriatumD1->Analyze(std::string("Channel"));
StriatumD2->Analyze(std::string("Channel"));
SNr->Analyze(std::string("Channel"));
STN->Analyze(std::string("Channel"));
GPe->Analyze(std::string("Channel"));
```

The resulting configuration file would look like:

```
#
# This file was automatically generated by the hrlAnalysis Software.
#

def getCellGroups():
  cellGroup = []
  cellGroup.append({"name":"Channel","startIdx":320,"endIdx":639})
```

```
    return cellGroup

if __name__ == "__main__":
    print getCellGroups()
```

Using HRLAnalysis

Once the simulation has been completed and the configuration has been generated the analysis can be completed off line using the python interface. Note that the use of the configuration file and the support scripts provided here is completely optional. You only need to tell python where the HRLAnalysis libraries are located and where the simulation output files are. At that point how the analysis is completed and what is done with it are up to you. Provided below is an example of how the provided scripts use the configuration files and plot the results of the analysis.

Setup

Unless the HRLAnalysis library path has been placed in the PYTHONPATH variable you will need to inform Python where those are located. The libraries are installed to the *HRLAnalysis/hrlAnalysis/lib/* folder. In the runAnalysis.py script this path is a command line option. The path and library can be imported using:

```
# Add the library include path
sys.path.append(options.includePath)
# import the analysis library
import libHRLAnalysis
```

Similarly, if you are using the generated configuration file you will need to tell python were that is located and dynamically import it. In the runAnalysis.py script this is done using the full filename:

```
# Import the configuration module.
(dir,fileName) = os.path.split(options.configFileName)
sys.path.append(dir)
try:
    exec('from %s import *' % fileName)
except:
    print "Error: Cannot find the requested configuration file: ", fileName
    raise
```

The configuration file provides one function that will return a dictionary containing information on all of the requested cell populations. Once the file is imported it can be called using:

```
# Get the cell groups and the parameters
cellGroups = getCellGroups()
```

The list of binary files can be extracted directly from the given directory using the following filter and regular expression code:

```
# Search for binary files in the search path.
binFiles = os.listdir(options.searchPath)
filterTest = re.compile("^spikes",re.IGNORECASE)
binFiles = filter(filterTest.search,binFiles)

for i in xrange(len(binFiles)):
    binFiles[i] = os.path.join(options.searchPath,binFiles[i])

# Sort the list of files since some machines will return the files in reverse order
binFiles.sort()

fileList = libHrlAnalysis.vector_string()
    for f in binFiles:
        fileList.append(f)
```

Finally, the analysis can be done each of the populations provided by the configuration file.

```
for cells in cellGroups:
    analyzeData(cells,options,binFiles)
```

Running the Analysis

After collecting the cell group information and the full paths to the binary files, the analysis object for a particular cell group can be constructed:

```
analysis = libHrlAnalysis.HrlNeuralAnalysisHRLSim(0,4000,26200,28199,fileNames)
analysis.buildDataStructures()
```

Note that the call to *buildDataStructures()* is optional. It will be called internally if an analysis function has been called before the data structures have been constructed.

The resulting analysis object and be used to fill in the desired analysis data structures. The spike raster data will fill two arrays, one containing a spike time and the other containing a cell index. Keep in mind there will be redundant spike times. This was originally intended for use in plotting the spiking activity as a raster plot. This is collected using:

```
# Gather the spike timing information
spikes = analysis.getSpikeTimes()
# The resulting object contains two lists
spikes.time
spikes.spikes
```

The population level average spiking activity can be computed using either

```
rates = analysis.calcWindowRate(options.WindowSize, options.StepSize)
```

or

```
rates = analysis.calcGaussWindowRate(options.WindowSize, options.StepSize)
```

There are several functions for getting individual cell information either separately or in a single call. Here the run-Analysis.py script is gathering the average spike count rate over the given interval as well as using that information to create bins of cell counts with activity in spike rate ranges.

```
# Get the Window Rates and the Binned Rates
numBins = 100
rateBinInfo = analysis.calcRatesWithBins(numBins)
# The resulting object contains four lists
rateBinInfo.cells
rateBinInfo.rates
rateBinInfo.freqs
rateBinInfo.counts
```

The coefficient of variation can be returned using:

```
# Get the COV analysis.
covInfo = analysis.getCOV()
# The resulting object contains two lists
covInfo.cells
covInfo.cov
```

Plotting the Results

There are a couple examples of plotting the calculated information provided in the suite. The main two use either the Python Biggles library of the Matplotlib library. The Biggles plots are simpler and the library os more efficient then then the Matplotlib library. However, the Biggles library is less flexible as far as options and output formats are concerned. An example of calling the provided Biggles interface is given below.

```
import hrlAnalysisPlt_biggles
plotter = hrlAnalysisPlt_biggles.spikePlotterBiggles(cellGroup['name'],
                                                     options.startTime, options.endTime,
                                                     cellGroup['startIdx'],cellGroup['endIdx'])
```

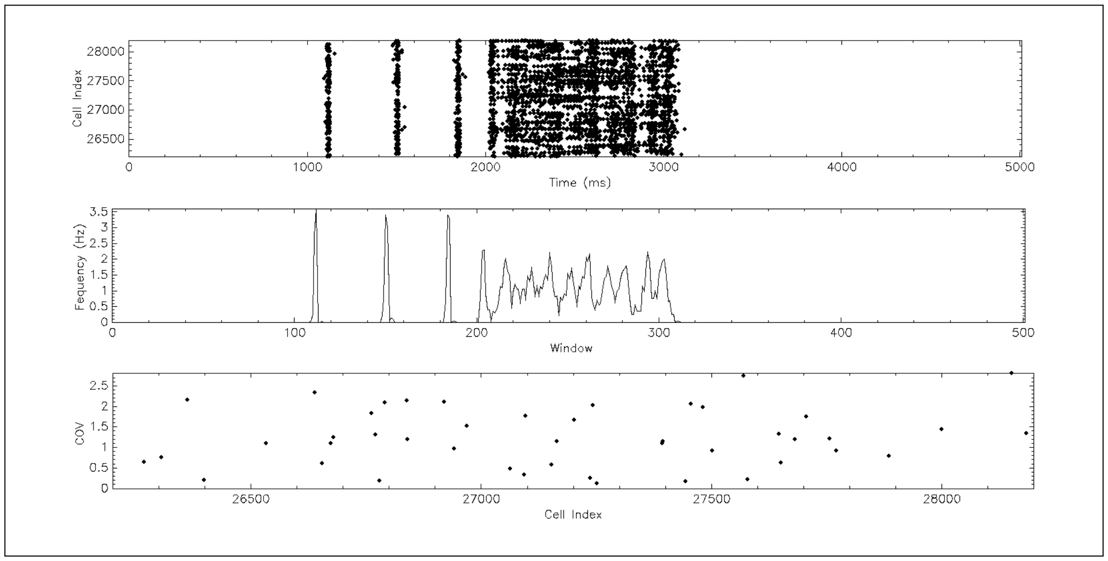

Figure 1: Simple plot using Biggles package

```
plotter.plotRaster(spikes.time,spikes.spikes)
plotter.plotWindowRate(rates.rates)

if len(covInfo.cov) > 0:
    plotter.plotCOV(covInfo.cells,covInfo.cov)

plotter.savePlot(os.path.join(options.outputPath,cellGroup['name']+'.png'))
```

The results of this command for the sample data provided in the code is shown in Figure 1.

The second plotting interface uses the Matplotlib library. This can be called using:

```
import hrlAnalysisPlt
plotter = hrlAnalysisPlt.spikePlotter(cellGroup['name'],
                                        options.startTime,options.endTime,
                                        cellGroup['startIdx'],cellGroup['endIdx'])

plotter.plotRaster(spikes.time,spikes.spikes)
plotter.plotWindowRate(rates.rates)

if len(covInfo.cov) > 0:
    plotter.plotCOV(covInfo.cells,covInfo.cov)

plotter.plotCellRates(rateBinInfo.cells,rateBinInfo.rates)
plotter.plotSpikeBins(rateBinInfo.freqs,rateBinInfo.counts)

plotter.savePlot(os.path.join(options.outputPath,cellGroup['name']+'.png'))
plotter.closePlot()
```

The resulting plot is presented in Figure 2.

Notice that each of these code listings use the group name to construct the output image file name. Currently the generator code will allow nonconsecutive groups to have the same name. If you would like to have unique names for each of the output files the function *uniqueFilenames* is proved in the runAnalysis.py script. This can be used to construct the file name that is passed to save plot.

```
def uniqueFilename(path,filename,extension):
```

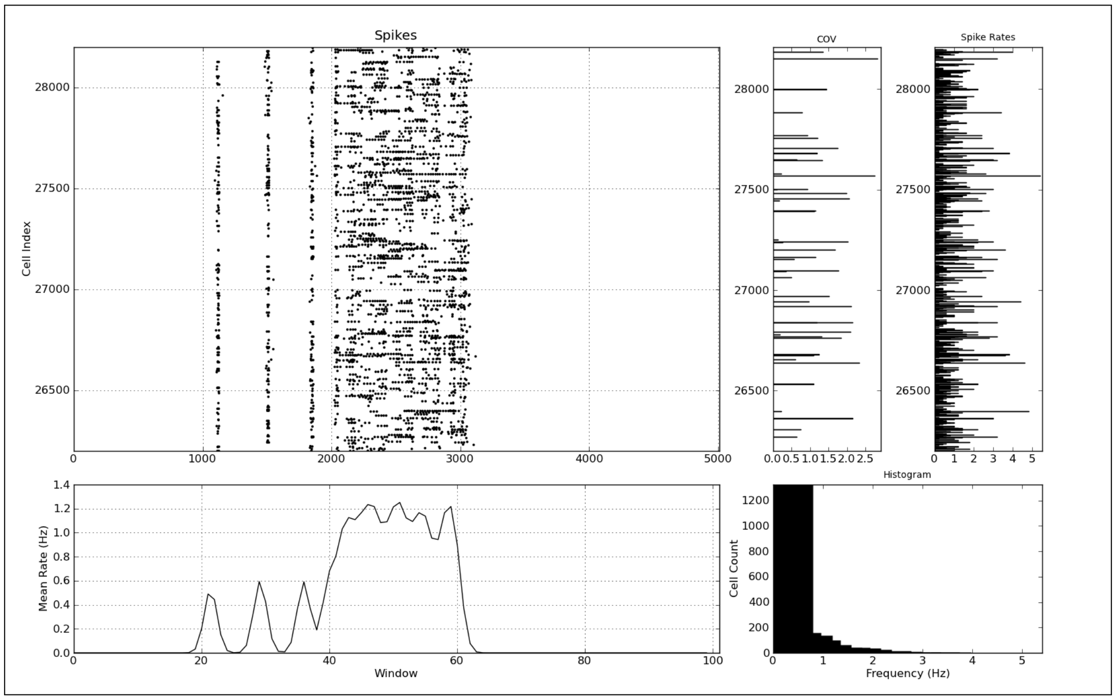

Figure 2: More detailed plot using Matplotlib package

```
    fileName = os.path.join(path, filename+extension)
    if( os.path.exists(fileName)):
        createFileName = lambda i: os.path.join(path,"%s_%d%s"% (filename,i,extension))
        for i in xrange(1, sys.maxint):
            fileName = createFileName(i)
            if not os.path.exists(fileName):
                return fileName
    else:
        return fileName

    return None
```

Disclaimer

The authors gratefully acknowledge the support for this work by Defense Advanced Research Projects Agency (DARPA) SyNAPSE grant HRL0011-09-C-001. The views, opinions, and/or findings contained in this source are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the DARPA or the Department of Defense. Approved for public release, distribution unlimited.