Analyzing large-scale spiking neural data with HRLAnalysis™

Corey M. Thibeault*, Michael J. O’Brien and Narayan Srinivasa

Center for Neural and Emergent Systems, Information and Systems Sciences Laboratory, HRL Laboratories LLC., Malibu, CA, USA

1. INTRODUCTION

Large-scale neural simulations have become an increasingly important tool in computational neuroscience. Although the methods behind these simulations may be different they all result in an extraordinary amount of simulated data. Whether it is the inclusion of highly detailed biophysical models (Markram, 2006), huge numbers of point neurons (Izhikevich and Edelman, 2008; Ananthanarayanan et al., 2009), or nervous system spanning functional anatomy (Eliasmith et al., 2012), the data deluge is a concern. In addition, the relatively low cost of high-performance computing systems and the recent popularity of neuromorphic hardware promises to continue this trend toward larger, more detailed models. And this problem is not exclusive to neural simulations. One of the underlying goals of the recently announced BRAIN initiative1 is to develop tools capable of capturing the activity of at least one million neurons—an effort that will greatly improve the state of the art but also result in tremendous amounts of data.

From a software engineering perspective, this data deluge can be approached in different ways. Using strongly-typed compiled languages, such as C, offers high-performance, but can sacrifice flexibility and extensibility. Languages popular in neuroscience, such as MATLAB or Python provide a relatively simple interface but can result in poor performance for many large-scale problems. In this paper we present, HRLAnalysis™, a software suite that aims to alleviate these concerns when processing spiking data.

Developed as part of the DARPA SyNAPSE program (Cruz-Albrecht et al., 2012; DARPA, 2012; Srinivasa and Cruz-Albrecht, 2012), the HRLAnalysis™ package was initially created as an implementation of off-line analysis and visualization of spiking and network data for use with HRLSim™ (Minkovich et al., 2013)—although as described below, the design does not unnecessarily restrict its use to only those data formats. One of the primary results of a neural simulation are recordings of the spiking activity—action potential event times—of each of the neurons in the network. Spike-train analysis is useful for providing insight into the structure and function of a neural network or region of the nervous system. For instance the information encoded about a stimulus can be extracted using a number of different spike and rate based codes derived from spike trains (Dayan et al., 2001; Ince et al., 2009; Quiroga and Panzeri, 2009; Crumiller et al., 2011). In addition, the synchronization in the spiking of neurons may indicate a pathological condition or this can used as a measure of neuronal coding (Kreuz et al., 2013). Similarly, correlated firings can reveal information about motor behavior, attention, and external stimulus (de la Rocha et al., 2007).

HRLAnalysis™ provides a way to efficiently calculate many of these useful metrics on a large number of spike trains. As a general tool, the scope of HRLAnalysis™ is limited to spike-train analysis. However, as we demonstrate, its design does allow for integration with existing tools that can complement the functions provided by HRLAnalysis™.

To balance efficiency with usability, the extraction—reading spike data from disk or converting other formats into spiking data—and analysis of the data is performed in C++ and the plotting and any further manipulation required by the user is handled in Python through the Boost package2. With this, users get the...
performance benefits of C++ but also the feature rich and syntactically simple interface of Python. In addition, users can take advantage of the extensive libraries available to Python for further manipulation and visualization of the results.

2. METHODS

2.1. INTRODUCTORY EXAMPLE

Before exploring the design of HRLAnalysis™ in detail consider the example in Figure 1. This gives a qualitative illustration of the results for a simulated two-layer network using a class that reads in HRLSim™ voltage files (included with the codebase). The plot was created using the Matplotlib library3. Listing 1 is the relevant code required to create this plot. Note that this is incomplete and assumes that the subplots have been initialized on Line 4.

Each of the different analysis functions—Lines 18, 24, 34, and 40—return pointers to specific structures. In python the different data members of these structures are seen as lists and can be seamlessly plotted with Matplotlib—Lines 19, 25, 30, 35, 41, and 43. This example highlights the simplicity of using HRLAnalysis™ but hides the complexity of the design; in particular that these objects are implemented and instantiated in C++.

2.2. DESIGN

The hybrid language design of HRLAnalysis™ is facilitated by the Boost Python package which provides the logic required to interface C++ with Python. Boost was chosen over SWIG4 for its ease of use and additional features. These include namespace support, automatic support for return by pointer and base-class reference, templated type-conversions, and support for multiple source files.

3http://www.matplotlib.org/
4http://www.swig.org/

A data-centric object-oriented design (OOD) pattern was selected for its balance of performance and extensibility, as well as for its amenability to interfacing with Python. With this, structures that organize the spike information in ways that are optimal to the analysis can be developed by combining the data with the methods. This allows for algorithmic and compiler optimizations—since the form of the data structures are known and directly accessible—as opposed to relying on decoupled data and access functions where the formats may not be consistent. Beyond the performance of data encapsulation, is the extensibility OOD provides through object inheritance. Users can extend the main library, described below, by adding or modifying functionality to fit their particular needs in derived class objects.

The rationale for employing a Python interface rather than MATLAB is similar. The object oriented aspects of Python make it far easier to reuse than MATLAB (Spacek et al., 2009). In addition, this provides flexibility that simplifies the management of large-scale projects. Conversely, MATLAB functions are written in individual files making it difficult to make major modifications to an existing codebase (Ince et al., 2009). Finally, since Python is open source there is potentially a larger available user base—all that is required is a compatible operating system as opposed to a commercial license.

2.2.1. Core classes

The organization of HRLAnalysis™ is illustrated in the context diagram of Figure 2. The abstract base class HrlNeuralAnalysis, written in C++, is the core of the analysis context. The data structures—described in section 2.2.2—and core analysis functions—described in section 2.2.3—are contained within this class.

---

**FIGURE 1 | Example analysis of simulated data using a derived voltage input class.** For this example the rectangular bars highlight regions where different signals are injected into the network. (A) Raster plot of 100 cells selected from the input layer. (B) Voltage traces of the target layer neurons. (C) Raster plot of the target layer. (D) Fire rate of the target output neurons calculated using a Gaussian window function. (E) Coefficient of variation for 100 cells of the input layer. (F) Average rate for the 100 sample cells. (G) Spike count histogram for the input neurons. (H) Fully connected feed-forward network.
Thibeault et al. Analyzing spiking data with HRLAnalysis™

Listing 1 | Pseudocode for creating Figure 1.

```python
# Import the Library
from libHrlAnalysis import *

# Setup Matplotlib subplots

# Initialize the analysis object for processing voltage files.
analysis = HrlNeuralAnalysisVoltage(startTime, stopTime, startIndex, stopIndex, fileNames, numCells, True, threshold)

# Figure 1A: raster plot of the spiking activity
raster = analysis.getBinnedActivity()
rasterPlot.scatter(raster.times, raster.spikes)

# Figure 1B: This function returns the voltage and is implemented by the derived class
# HrlNeuralAnalysisVoltage
voltages = analysis.Voltages()
voltagePlot.plot(voltages.voltage)

# Figure 1D: The population rate over the # analysis window.
windowRate = analysis.getGaussWindowRate(10,10)
ratePlot.plot(windowRate.rates)

# Figure 1E: Coefficient of Variation for each # of the neurons.
cov = analysis.getCov()
covPlot.bar(cov.cells, cov.cov)

# Figure 1F and 1G: The individual cell.
rateBinInfo = analysis.getRatesWithBins(100)
cellsRatePlot.bar(rateBinInfo.cells,
rateBinInfo.rates)
binnedPlot.bar(rateBinInfo.freq,
rateBinInfo.counts)

# Figure 1A: raster plot of the spiking activity
raster = analysis.getBinnedActivity()
rasterPlot.scatter(raster.times, raster.spikes)

# Figure 1B: This function returns the voltage and is implemented by the derived class
# HrlNeuralAnalysisVoltage
voltages = analysis.Voltages()
voltagePlot.plot(voltages.voltage)

# Figure 1D: The population rate over the # analysis window.
windowRate = analysis.getGaussWindowRate(10,10)
ratePlot.plot(windowRate.rates)

# Figure 1E: Coefficient of Variation for each # of the neurons.
cov = analysis.getCov()
covPlot.bar(cov.cells, cov.cov)

# Figure 1F and 1G: The individual cell.
rateBinInfo = analysis.getRatesWithBins(100)
cellsRatePlot.bar(rateBinInfo.cells,
rateBinInfo.rates)
binnedPlot.bar(rateBinInfo.freq,
rateBinInfo.counts)
```

Specialized child objects that derive from the base class are then defined to process different spike data formats—this can be from files, live recordings or running simulations. These are only responsible for implementing the virtual function buildDataStructures and constructing the internal data structures, described below. In addition, these derived classes can extend the core functionality by adding additional analysis functions.

The execution of these core functions is controlled from the Python interface. As an example, the provided derived class that handles the HRLSim™ data format would be instantiated in Python using

```python
from libHrlAnalysis import *
analysis = HrlNeuralAnalysisHRLSim(
sampleStartTime, sampleStopTime, neuronStart, neuronStop, cellIndex, fileNames)
```

The constructor reads in the bounds of the analysis window, sampleStartTime and sampleStopTime, which define the region of interest within the experiment. The range of neuron indices is then defined by the neuronStartIndex and neuronStopIndex variables. Finally, a vector, composed of the file paths containing the spikes, is passed in as the fileNames variable.

At this point analysis is not completely initialized since buildDataStructures has not been called—it is bad practice to call a virtual function from a constructor (Sutter and Alexandrescu, 2004). The user does have the option of directly calling an analysis function since the core library will check if the internal structures have been constructed and buildDataStructures will be called if not. Alternatively, buildDataStructures can be called directly using

```python
analysis.buildDataStructures()
```

The resulting data structures are described below.

2.2.2. Data structures

The two main storage containers constructed in the buildDataStructures function, cellActivity and spikeActivity, organize the spike information by cell or time (see Figure 3). Internally, spikeActivity is a Standard Template Library (STL) vector (Stepanov and Lee, 1995), of STL pairs, containing spike time and cell index, ordered by spike times. Similarly, the cellActivity structure is implemented with a two-dimensional STL vector, where the first dimension corresponds to the neuron and the second contains the spike times for that neuron. The core analysis methods, described below, will utilize the structure that offers the best performance.

Placing the information in redundant structures sacrifices memory but provides the highest performance when a user has the need for the associated analysis methods—it is more efficient computationally to build these with the raw data than convert from one to the other. This is the case in the example derived classes provided with the suite. However, there is no requirement that a derived class build both structures. In addition, future releases of the library will give the user the option of using just-in-time construction of only the structure that is required when they call an analysis function.

With this design the base object encapsulates both the data and the analysis methods—sacrificing generality for optimization and consistency. This runs counter to the current trends in computational neuroscience where most, including the authors, are pushing for more collaborative projects and standardized data formats. The decision for internal structures was based on the
FIGURE 2 | Context diagram. The HrlNeuralAnalysis base class is the core of HRLAnalysis™. This is written in C++ and contains the data structures and analysis methods. Derived classes are created in either C++ or Python to handle the conversion of the data file input into the two core structures. These classes are responsible for implementing the buildDataStructures virtual function. The access control and visualization is then provided through the Python interface.

FIGURE 3 | Core Data Structures (C++). The spikeActivity, top, stores sorted pairs of spikes times and cell indexes in a vector. The cellActivity structure is two-dimensional vector containing spike times for each of the cells.

need for performance. However, as we demonstrate below this does not impede interoperability.

2.2.3. Analysis functions
HRLAnalysis™ provides a useful set of basic functions for spiking probability and statistics. This particular collection of methods was selected based on the initial needs of the authors for dealing with large amounts of simulated data. They therefore do not cover the full breadth of possible spike-train analysis. Additional functions are constantly being developed and, once mature, will be included in future releases. The currently supported functions, listed in Table 1, are briefly described below. For more information about the implementation of these refer to the references or the available codebase.

The spike-count rate, $r$, of a single neuron is a simple way to quantify the average activity over a window of time. Despite the loss of temporal information this metric is useful in characterizing the response of a neuron to a stimulus (Dayan et al., 2001). These can be used to find the tuning curves of neurons and their distribution within a population can reveal stimulus selectivity or redundancy in the network. In HRLAnalysis™ the spike-count rate for each of the neurons is found using getCellRates, the spike-count histogram is returned by getRateBins, and both can be computed using getRatesWithBins—the combined method reduces redundant computations.

The loss of temporal resolution can be partial restored by approximating the firing rate, $r(t)$, of a single neuron or a whole population. This can give a better representation of the stimulus tuning. HRLAnalysis™ provides two methods for approximating the firing rate. The getWindowRate method counts spikes along a rectangular window that is slid over the spike trains; the window size and step size are defined by the user. In addition, these can also be filtered using a Gaussian window function that smooths the response by weighting the influence of the cells before and after the current time. This is provided by the getGaussWindowRate.

Although important, these rate codes can fail to capture important spike response variability or information about disparate cells within a population. For a single neuron this variability can be captured by the inter-spike interval coefficient of variation (COV)—assuming a stationary rate. The getCOV function performs this calculation. The COV is useful for quickly identifying neurons with irregular spiking activity over the period of analysis. In addition, this is a necessary condition for identifying Poisson firing (Dayan et al., 2001).

The COV reduces neural variability to a single number but often the quality of that variability is important. One example is in capturing the bursting of a spike train. Bursting can be an indication of a pathological condition, such as in Parkinson’s disease (Rubin et al., 2012). It can also be an important indicator of neuronal modulation (Hanes et al., 1995). The getBursting function is used to find the regions of bursting for each of the neurons in a population. This can be used to explore metrics such as the synchronization of bursting or the overall burst rate of a population.

Along the same lines, the correlated firing of cells within a population can both indicate a pathological condition (Walters and
Thibeault et al.

Analyzing spiking data with HRLAnalysis™

Table 1 | HRLAnalysis™ spike analysis functions currently available.

| Function name | Description | Structure | References |
|---------------|-------------|-----------|------------|
| get Bursting  | Identify periods of bursting for individual cells using Poisson surprise | Cell activity | Hanes et al., 1995 |
| get Pair Synchrony | Bivariate spike dissimilarity between two cells using the SPIKE-distance method | Cell activity | Kreuz et al., 2013 |
| get Population Synchrony | Bivariate spike dissimilarity for the entire population using the SPIKE-distance method | Cell activity | Kreuz et al., 2013 |
| get COV | Calculate the coefficient of variation for cells that had more than 10 spikes | Cell activity | Dayan et al., 2001 |
| get Cell Rates | Calculate the individual cell spike count averages | Cell activity |  |
| get Rate Bins | Bin the number of cells based on average spike counts | Cell activity |  |
| get Rates With Bins | Combines get Cell Rates and get Rate Bins | Cell activity |  |
| get Gauss Window Rate | Approximate the fire rate of the population using a Gaussian window | Spike activity | Dayan et al., 2001 |
| get Window Rate | Approximate the fire rate of the population using a rectangular window | Spike activity | Dayan et al., 2001 |
| get Spike Times | Create separate vectors of cell index and spike time | Spike activity |  |

Bergstrom, 2010), as well as reveal details of stimulus encoding (de la Rocha et al., 2007). HRLAnalysis™ provides both average measures of spike-train synchrony as well as instantaneous synchrony between cells over the sample interval using the SPIKE-distance method of Kreuz et al. (2013). The instantaneous dissimilarity between two cells is found using get Pair Synchrony and can be found for the entire population using get Population Synchrony. The results of these can be used to find a single distance metric using either calc SPIKE Distance or calc SPIKE Distance to find the bivariate SPIKE-distance by numerical integration using Simpson’s rule or taking the average of the dissimilarity profiles, respectively. This a useful metric for effectively obtaining a comparable representation of the overall synchrony between spike trains.

It is important to note that all of these functions are implemented in the HrlNeuralAnalysis base class and directly accessible to the Python functions. Once the spike data has been extracted, the use of these methods is relatively simple. Returning to the example above, to calculate the COV for each of the cells the user would call

```python
covInfo = analysis.getCOV()
```

The resulting object contains a vector—seen as a list in Python—of the COVs for each of the cells. All of the analysis functions return similar objects that encapsulate the results. The motivations for this design are described below and for a complete reference refer to the available codebase.

### 2.2.4. Implementation

Algorithmically, the development of the analysis functions was aided by their coupling to the data structures. As mentioned above, those structures are constructed from STL vectors as opposed to creating custom array based storage which may have resulted in increased performance. However, not only would that have made interfacing with Python more complicated, but it would have made the analysis methods more difficult to design and test. In this case correctness was valued over performance.

STL vectors were also used exclusively for local storage in the analysis functions. To increase the performance of these methods memory was reserved before filling the vectors and `push_back` was the only insertion method, resulting in a constant time insertion cost. There were some instances where the size of the vector was unknown *a priori*—such as during burst analysis. For these cases, the use of `push_back` still results in amortized constant time cost (Sutter and Alexandrescu, 2004). STL containers were favored since these are already highly optimized and, based on our previous experience, hand-tuned structures likely would not have resulted in much, if any, performance benefit (Minkovich et al., 2013). Finally, for the same reasons, built-in STL algorithms were used wherever possible (e.g., `sort` and `accumulate`).

Another design consideration was the choice of control structures. All of the analysis functions require iterating over one of the data objects. In C++ this can be accomplished by directly indexing into the container memory and looping over the length of that by incrementing the index. A more portable way of performing this, is through the use of iterators. These are specialized pointers to locations within the structure that can be used to access the data. On the one hand employing iterators makes for more generic and arguably safer code. On the other, there are some instances where this unnecessarily hides details from the compiler that could otherwise be optimized (Sutter and Alexandrescu, 2004). Furthermore, both of these methods lack bounds checking—meaning there is nothing stopping the code from accessing memory beyond that allocated to the structure. The use of `BOOST_FOREACH`—whose underlying implementation employs iterators—removes many of the pitfalls of iterators and leads to much more readable code. Unfortunately, there are some compilers where this can result in a slight performance drop. During initial testing, however, with the core data structures, the performance of `BOOST_FOREACH` was almost identical to using either iterator or indexed based loops. Because of this, its use was preferred during development when the algorithm lent itself. Another option would have been the use of the C++11 standard range-based `for` method. Unfortunately that was not an option for this version of the library.

Another area of concern are memory leaks resulting from passing objects between C++ and Python. Earlier versions of the library relied on dynamically allocated containers being constructed by the user in Python. These were then passed by reference to the C++ algorithms. This design ensured that the memory was appropriate deallocated but was unfavorable for two reasons. The first was that the library would rely on the Python
code to provide the appropriate variable type in the correct position within the parameter list. This made the library difficult to use. The second was that it created extra code that unnecessarily complicated the interface.

To avoid this, all dynamically allocated objects were created with Boost shared pointers. These are class templates that use reference counting to ensure that the object they point to is deleted after all pointers to it are deleted. With these, objects created in C++ can be passed to Python and, when they go out of scope, they are guaranteed to be deleted. This not only made dealing with object ownership between Python and C++ more tractable but also removed container initialization code from the Python interface—greatly improving the readability of the user code.

Although performance was a motivating factor in the design of this library, care was taken to not only follow good programming practices but also produce maintainable and testable code. There are some instances where algorithms could be further optimized but much of that would come at the detriment of readability and extensibility. Because of this, HRLAnalysis™ is not fully optimized. However, as the library matures and as bottlenecks within these functions are identified, the algorithms will be further enhanced. In addition, as we demonstrate with the benchmarks below, the library still outperforms comparable spike analysis options.

2.2.5. Extensibility

A unique aspect of this design, and a useful feature, is that the derived classes can be either C++ or Python. This provides users with different levels of programming experience access to the high-performance core. Listing 2 is an incomplete example of deriving from the HrlNeuralAnalysis base class with Python. The incoming data in this example is stored by cell indexed spike trains. This layout allows the cellActivity structure to be constructed in order, however, the spikeActivity structure needs to be sorted when built this way (this is handled by the call to the C++ function sortSpikeActivity). Alternatively, only the cellActivity structure can be constructed and, if required, the spikeActivity structure can be created in C++ using the buildSpikeActFromCellAct function. This can result in higher performance during the construction of the internal data structures.

Extending the base class, whether in Python or C++, does not limit the implementation to spikes. The derived class can be used to process data in different forms as long as the data contains enough information for spikes to be extracted. For example, spike analysis can be done using voltage recordings with the HRLAnalysis™ core and a derived class that includes methods specific to voltage information.

Generally, encouraging extensions to a software package can lead to instability. Unit, integration and system tests are included with HRLAnalysis™ to combat such software entropy. These are intended to ensure that the core functionality remains stable as users develop their own modules.

Listing 2 | Example of a derived HRLAnalysis™ base object in Python.

In this example the derived constructor prototype matches the base class definition. Notice on line 6 that the base class constructor needs to be called by the derived class. The programmatic loop outlined in line 12 is repeated for each of the spike trains and it is assumed that cell_spike_train() on line 14 is provided by that loop. The data structures are incrementally constructed with calls to append(). These are C++ data structures, so they map to the C++ vector push_back() function. The function call on line 20 sorts the spikeActivity structure by spike time. Note that the spikeActivity structure can be created in C++ by the HrlNeuralAnalysis object using the buildSpikeActFromCellAct function. This should provide a nominal performance increase over the method presented here.

2.3. BENCHMARKS

As a way of quantifying the performance of HRLAnalysis™, a set of benchmarks comparing it the Python analysis suite NeuroTools6 version 0.1.0 were completed. Not all of the analysis functions implemented in HRLAnalysis™ have corresponding methods in these packages but a comparable subset was selected. These are listed in Table 2. Only the time to complete the analysis function is measured. This was due to the extended time it took to load the data into NeuroTools which would have drastically altered the results.

For test data, Poisson spike trains 10 s long with a target rate of 10 Hz were generated for networks ranging from 1K to 10M cells. However, for NeuroTools the largest network that could be analyzed in memory was 1M cells. The spike trains were saved in binary files with spikes, timing, and size information stored as 32-bit integers. The performance of the analysis methods are dependent on both the number of cells and the number of spikes generated—as both are increased not only does the timing increase but also the file IO. Using independent Poisson generators, rather than a large test network of model neurons, results in a number of spikes that grows somewhat linearly with the number of cells. This was important for exploring the scalability of the library. The number of neurons and spikes generated are included in Table 3.
Faster. This is a significant performance difference but one that
exists. This is a result of the underlying implementation in the
language.

E5550 2.67 GHz CPUs and 48 GB of memory running CentOS
for the best performance of those was reported. These
benchmarks were completed on a server with dual Intel Xeon

The performance that these benchmarks illustrate is not only
important for large neural networks but also for large numbers
of small network simulations. The parameter searches described
in the conclusion of Thibeault and Srinivasa (2013) are on the
order of 1.5 billion simulations and take over 23,000 h of
computation time. Roughly half of that is during the analysis using
HRLAnalysis™. As a conservative estimate, assume that only
half of that number was in the actual analysis functions—around
5750 h—and that only a 10 time speedup over NeuroTools is pos-
possible. In that case these simulations would have required 80,500 h
of compute time—3.5 times longer than with HRLAnalysis™.
This is an important factor to consider; especially if researchers do
not have unlimited access to computing resources and are instead
buying time on a compute cluster.

Another important result of these benchmarks is the scaling
of the HRLAnalysis™ implementation with respect to data size. Table 2
contains the timing results for reading in and initializing the
data structures, as well as the time required to run the meth-
ods listed in Table 3. Although the analysis functions presented
have approximately linear scaling, the file IO and data initia-
ization scaling does not. Despite that non-linear trend, the suite runs
faster than real time—for the cases below 1M cells. For 1M cells the suite completed ini-
itialization and all of the methods in under 1.5 times real time.
For the 10M cell case, however, the run time is greater than 10.5
times real time.

These benchmarks are meant only as a reference. Although
performance is important in this project, we are not suggesting
that HRLAnalysis™ is a replacement or a competitor to libraries
like NeuroTools. It is quite the opposite, we feel these packages
complement each other. Both are important tools for neurosci-
entists and the Python language makes exchanges between the
two possible. The only overlapping analysis methods between
HRLAnalysis™ and NeuroTools are those listed in Table 2.

3.2. INTEROPERABILITY
HRLAnalysis™ not only performs well but, through the Python
interface, it can be coupled with other analysis libraries. For
example, in NeuroTools a SpikeList object contains a list of SpikeTrains—this is comparable to the cellActivity object
described here. The constructor for this object takes in a list of
tuple pairs of the form (cell_id,spike_time), a list of all of the cell
ids, the starting time of the data, and the end time of the data.
Converting from HRLAnalysis™ into a NeuroTools object can be
done in the single compound line of code:

```
SpikeList(
    [(spk.second,spk.first)
    for spk in spikeActivity()]
)
```

Timings for NeuroTools and HRLAnalysis™ were computed
using the Python Time module. Each benchmark was run three
times and the best performance of those was reported. These
should be considered approximations, as all are subject to dif-
ferent resolutions based on the underlying implementation. The
benchmarks were completed on a server with dual Intel Xeon
E5550 2.67 GHz CPUs and 48 GB of memory running CentOS
5.4. Python version 2.6.5 were used.

3. RESULTS
3.1. BENCHMARKS
Compared to NeuroTools, HRLAnalysis™ is on average 107 times
faster. This is a significant performance difference but one that
is obscured slightly by the getWindowRate results. The speedup
for this function is a consequence of the optimized spikeActivity
structure. It is likely that the performance difference of this func-
tion would drop if the conversion to a time based orientation was
not required in NeuroTools.

Table 2 | Benchmark results.

| Function  | Cells | Analysis time (s) | Speedup |
|-----------|-------|------------------|---------|
| getSpikeTimes | 1k    | 0.0112           | 0.0004  | 30      |
| (raster_plot) | 10k   | 0.1196           | 0.0050  | 24      |
|           | 100k  | 1.2079           | 0.0487  | 24      |
|           | 1M    | 12.1409          | 0.4853  | 25      |
|           | 10M   | –                | 4.8941  | –       |
| getRatesWithBins | 1K   | 0.0019           | 0.0002  | 8       |
| (rate_distribution) | 10K  | 0.0180           | 0.0024  | 7       |
|           | 100K  | 1.1744           | 0.0264  | 7       |
|           | 1M    | 1.7551           | 0.3711  | 5       |
|           | 10M   | –                | 4.1252  | –       |
| getWindowRate | 1K   | 0.0674           | 0.0003  | 209     |
| (firing_rate)  | 10K   | 0.6115           | 0.0020  | 334     |
|           | 100K  | 6.5461           | 0.0182  | 360     |
|           | 1M    | 65.7611          | 0.1808  | 364     |
|           | 10M   | –                | 1.8568  | –       |
| getCOV | 1K    | 0.0169           | 0.0012  | 15      |
| (cv isi)   | 10K   | 0.1734           | 0.0136  | 13      |
|           | 100K  | 1.7140           | 0.1601  | 11      |
|           | 1M    | 17.1406          | 1.6327  | 10      |
|           | 10M   | –                | 15.8049 | –       |

Table 3 | Benchmark results for HRLAnalysis™.

| Cells | Number of spikes | Time (s) |
|-------|------------------|----------|
|       |                  | Build    | Methods | Total   |
| 1K    | 99,263           | 0.0063   | 0.0021  | 0.0084  |
| 10K   | 993,727          | 0.0483   | 0.0230  | 0.0713  |
| 100K  | 9,947,156        | 0.7352   | 0.2544  | 0.9896  |
| 1M    | 99,501,298       | 11.2022  | 2.6699  | 13.8721 |
| 10M   | 994,993,689      | 137.8604 | 26.6810 | 164.5414 |

These results are for the analysis only. All data structures were initialized before
the benchmarks were started. The HRLAnalysis™ functions are in bold and the
 responding NeuroTools functions is included below this in parenthesis.

Table 2 provides a measure of compute time—3
example, in NeuroTools a

The performance that these benchmarks illustrate is not only
important for large neural networks but also for large numbers
of small network simulations. The parameter searches described
in the conclusion of Thibeault and Srinivasa (2013) are on the
order of 1.5 billion simulations and take over 23,000 h of
computation time. Roughly half of that is during the analysis using
HRLAnalysis™. As a conservative estimate, assume that only
half of that number was in the actual analysis functions—around
5750 h—and that only a 10 time speedup over NeuroTools is pos-
possible. In that case these simulations would have required 80,500 h
of compute time—3.5 times longer than with HRLAnalysis™.
This is an important factor to consider; especially if researchers do
not have unlimited access to computing resources and are instead
buying time on a compute cluster.

Another important result of these benchmarks is the scaling
of the HRLAnalysis™ implementation with respect to data size. Table 2
contains the timing results for reading in and initializing the
data structures, as well as the time required to run the meth-
ods listed in Table 3. Although the analysis functions presented
have approximately linear scaling, the file IO and data initia-
ization scaling does not. Despite that non-linear trend, the suite runs
faster than real time—less than the 10 s of simulation time—for
the cases below 1M cells. For 1M cells the suite completed ini-
itialization and all of the methods in under 1.5 times real time.
For the 10M cell case, however, the run time is greater than 10.5
times real time.

These benchmarks are meant only as a reference. Although
performance is important in this project, we are not suggesting
that HRLAnalysis™ is a replacement or a competitor to libraries
like NeuroTools. It is quite the opposite, we feel these packages
complement each other. Both are important tools for neurosci-
entists and the Python language makes exchanges between the
two possible. The only overlapping analysis methods between
HRLAnalysis™ and NeuroTools are those listed in Table 2.

3.2. INTEROPERABILITY
HRLAnalysis™ not only performs well but, through the Python
interface, it can be coupled with other analysis libraries. For
example, in NeuroTools a SpikeList object contains a list of SpikeTrains—this is comparable to the cellActivity object
described here. The constructor for this object takes in a list of
tuple pairs of the form (cell_id,spike_time), a list of all of the cell
ids, the starting time of the data, and the end time of the data.
Converting from HRLAnalysis™ into a NeuroTools object can be
done in the single compound line of code:

```
SpikeList((spk.second,spk.first)
    for spk in spikeActivity())
```
It is this kind of cross tool compatibility that sets Python projects apart from other languages popular in neuroscience. It should be noted that this particular constructor is appropriate only for a small number of cells and spikes. In our benchmarks this conversion took between 1.7 and 1.8h to complete for the 1M cell case. This was due to the way the SpikeList object is constructed in NeuroTools. As the need arises more efficient conversion methods will be implemented and released.

Collaboration is important for data exchange as well as algorithm sharing. The Neo project (Garcia et al., 2014), approaches this by providing a common data object. Developers create interpreters for different open and proprietary file formats that instantiate and return a Neo object. With this, analysis methods can be developed that operate directly on Neo objects while requiring no knowledge of the underlying file format. An example of creating the HRLAnalysis™ structures from a Neo object is illustrated in Listing 3. This is incomplete in that there is no error checking and some of the analysis parameters are not filled in (line 27), but it demonstrates one way a Neo object containing spike trains can be incorporated into HRLAnalysis™. A complete working example is included with the Python tests in the available codebase.

```
import neo
import quantities as pq
import libHRLAnalysis as hrla

class HrlNeuralAnalysisNEO:
    hrla.HrlNeuralAnalysisNEO(
        hrla.HrlNeuralAnalysis)

    __init__(self, neoObj):
        libHRLAnalysis.HrlNeuralAnalysis__init__(
            self, 0, 0, 0, 0, hrla.vector_string())

        self.neoObj = neoObj

    def buildDataStructures(self):
        segment = -self.neoObj.read(cascade=True,
            lazy=False)

        currIdx = 0

        for cell in segment.spikeTrains:
            cell.units = pq.ms

            self.cellActivity().append(hrla.vector_int())

            for spike in cell:
                self.cellActivity()[currIdx].append(int(spike))

                self.spikeActivity().append(hrla.int_pair(
                    int(spike), currIdx))

            currIdx -= 1

        self.sortSpikeActivity()

        # fill in params

        self.params().isDataCompiled = True

        return True
```

Listing 3 | Example of reading in a Neo object.

Unlike the example in Listing 2, the derived class constructor (line 6), takes a Neo object as an argument. The base class constructor still needs to be called though it is filled with null data initially. In the buildDataStructures function (line 11), the spike trains are read from the Neo object and the structures are iteratively constructed. Once again the spikeActivity structure is sorted (line 26). Finally, at this point all of the information about the object should be available and line 27 marks the location where the parameters that were set to null in the constructor (line 7), would be set using that information.

4. DISCUSSION

4.1. SIMILAR PROJECTS

There are a number of analysis packages that have been developed for processing simulations and recordings in neuroscience. The International Neuroinformatics Coordinating Facility’s software database7 contains a large number of projects that span all aspects of neuroscience research. However, we were unable to find a high-performance library for spike analysis that fits our needs.

The aforementioned Python based NeuroTools library focuses minimally on spike train analysis. The broader scope of the project is to reduce the amount of redundant code computational neuroscientists develop and provides functions for designing, simulating and analyzing neural networks. Similarly, the Spykeutils (Pröpper and Obermayer, 2013), library provides methods for analyzing and visualizing spiking information but employs Neo objects as data structures. Neuropy takes a more data-centric approach based on electrophysiological data (Spacek et al., 2009). Information is stored hierarchically based on animal, track and recording. OpenElectrophy deals with large amounts of experimental data by storing it in a MySQL database (Garcia and Fourcaud-Trocmé, 2009). Users can interact with OpenElectrophy through the GUI interface to visualize the raw data as well as perform spike and oscillation detection. The spike detection algorithms are provided but the analysis must be performed by user supplied scripts through the Python interface.

There are similar projects with a focus on high-performance available for MATLAB as well. The FIND package provides both analysis tools for real and simulated single or multi-channel recordings, as well as methods for simulating neurons as point processes (Meier et al., 2008). The Information Breakdown Toolbox is specific to information analysis and takes advantage of MATLAB’s MEX interface to speedup computations (Magri et al., 2009). Similarly, the Spike Train Analysis Toolkit (Goldberg et al., 2009), is compatible with both MATLAB and Octave (Eaton et al., 2008), and provides information theoretic analysis of spiking data. This type of analysis is not currently implemented in HRLAnalysis™.

4.2. PERFORMANCE, EXTENSIBILITY, AND USABILITY

Often when performance is a priority, other desirable software engineering aspects suffer and this project is no different. Although we have aspired to balance performance with extensibility and usability, it is still difficult to achieve all three. As a consequence, this package may have a higher than desired level of specificity.

The complexity of the hybrid language approach may be discouraging to some researchers. In addition, the expectation of a neuroscientist with limited programming experience modifying and adding functions to the core C++ library is unrealistic. However, the option of writing those functions in Python results in a package that not only performs well, but is accessible to users with varying programming experience. It is our intention that as the project attracts more users, these Python prototype functions can be ported into the core library by developers with

7http://software.incf.org/
more experience. This is one of the key benefits of using a hybrid-language design.

4.3. FUTURE DIRECTIONS

The large-scale neural simulations that this project was originally intended to support, all rely on parallel processing for efficient computations. Analyzing the results of those using a serial application is somewhat counterintuitive. So the next logical step for HRLAnalysis™ is to port the codebase to parallel and distributed systems. Fortunately, having the Python interface can make that transition relatively seamless to the end user. As a first step, we plan to implement the analysis functions in OpenCL. This will provide a standardized interface to a large number of hardware platforms.

Although not discussed here in detail, we have begun implementing network connectivity and synaptic weight analysis tools with HRLAnalysis™. These are currently exclusive to file formats output by HRLSim™. However, a more generalized set of tools is planned for the future.

5. CONCLUSION

The HRLAnalysis™ suite offers both high-performance and usability. More importantly, it complements the existing packages available to researchers today. The capabilities of these different projects and the ability to connect them together through Python is exciting and can only be a benefit to the community. Researchers interested in HRLAnalysis™ can access it through the HRL Laboratories Center for Neural and Emergent Systems website.

ACKNOWLEDGMENTS

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 article 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.

REFERENCES

Ananthanarayanan, R., Esser, S. K., Simon, H. D., and Modha, D. S. (2009). “The cat is out of the bag: cortical simulations with 10⁸ neurons, 10¹⁵ synapses,” in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (Portland, OR: IEEE), 1–12.

Crumiller, M., Knight, B., Yu, Y., and Kaplan, E. (2011). Estimating the amount of information conveyed by a population of neurons. Front. Neurosci. 5:90. doi: 10.3389/fnins.2011.00090

Cruz-Albrecht, J., Yung, M., and Srinivasa, N. (2012). Energy-efficient neurons, synapse and stdp integrated circuits. IEEE Trans. Biomed. Circ. Syst. 6, 246–256. doi: 10.1109/TBCAS.2011.2174152

DARPA. (2012). Synapse broad agency announcement (BAA). Available online at: https://www.fbo.gov/spg/ODA/DARPA/CMO/BA08-28/Listing.html

Dayan, P., Abbott, L. F., and Abbott, L. (2001). Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. Cambridge, MA: MIT Press. doi: 10.1016/S0506-4522(00)00552-2

de la Rocha, J., Doiron, B., Shea-Brown, E., Josic, K., and Reyes, A. (2007). Correlation between neural spike trains increases with firing rate. Nature 448, 802–806. doi: 10.1038/ nature06028

Eaton, J. W., Bateman, D., and Hauberg, S. (2008). GNU Octave Manual Version 3. Bristol: Network Theory Limited.

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, C., et al. (2012). A large-scale model of the functioning brain. Science 338, 1202–1205. doi: 10.1126/science.1225266

García, S., and Fourcaud-Trocmé, N. (2009). Openelectrophys: an electrophysiological data- and analysis-sharing framework. Front. Neuroinform. 3:14. doi: 10.3389/neuro.11.014.2009

Garcia, S., Guarino, D., Jailliet, F., Jennings, T. R., Präpper, R., Rautenberg, P. L., et al. (2014). Neoo: an object model for handling electrophysiology data in multiple formats. Front. Neuroinform. 8:10. doi: 10.3389/fninf.2014.0010

Goldberg, D. H., Victor, J. D., Gardner, E. P., and Gardner, D. (2009). Spike train analysis toolkit: enabling wider application of information-theoretic techniques to neurophysiology. Neuroinformatics 7, 165–178. doi: 10.1007/s12021-009-9049-y

Hanes, D. P., Thompson, K. G., and Schall, J. D. (1995). Relationship of pre-saccadic activity in frontal eye field and supplementary eye field to saccade initiation in macaque poisson spike train analysis. Exp. Brain Res. 103, 85–96. doi: 10.1007/BF00241967

Ince, R. A. A., Petersen, R. S., Swan, D. C., and Panzeri, S. (2009). Python for information theoretic analysis of neural data. Front. Neuroinform. 3:4. doi: 10.3389/neuro.11.004.2009

Izhikevich, E. M., and Edelman, G. M. (2008). Large-scale model of mammalian thalamocortical systems. Proc. Natl. Acad. Sci. U.S.A. 105, 3593–3598. doi: 10.1073/pnas.0712231105

Kreuz, T., Chicharro, D., Houghton, C., Andrzejak, R. G., and Mormann, F. (2013). Monitoring spike train synchrony. J. Neurophysiol. 109, 1457–1472. doi: 10.1152/jn.00873.2012

Magri, C., Whittingstall, K., Singh, V., Logothetis, N. K., and Panzeri, S. (2009). A toolbox for the fast information analysis of multiple-site LFP, EEG and spike train recordings. BMC Med. 10:81. doi: 10.1186/1471-2202-10-81

Markram, H. (2006). The blue brain project. Nat. Rev. Neurosci. 7, 153–160. doi: 10.1038/nrn1848

Meier, R., Egert, U., Aertsen, A., and Nawrot, M. P. (2008). FINET A unified framework for neural data analysis. Neural Netw. 21, 1085–1093. doi: 10.1016/j.neunet.2008.06.019

Minkovitch, K., Thibeault, C. M., O’Brien, M. J., Nogin, A., Cho, Y., and Srinivasa, N. (2013). HRLSim: a high-performance spiking neural simulator for GPGPU clusters. IEEE Trans. Neural Netw. Learn. Syst. 25, 316–331. doi: 10.1109/TNNLS.2013.2276056

Pröpper, R., and Obermayer, K. (2013). Spyke viewer: a flexible and extensible platform for electrophysiological data analysis. Front. Neuroinform. 7:26. doi: 10.3389/fninf.2013.00026

Quiroga, R. Q., and Panzeri, S. (2009). Extracting information from neuronal populations: information theory and decoding approaches. Nat. Rev. Neurosci. 10, 173–185. doi: 10.1038/nrn2578

Rubin, J. E., McIntyre, C. C., Turner, R. S., and Wichmann, T. (2012). Basal ganglia activity patterns in parkinsonism and computational modeling of their down-stream effects. Eur. J. Neurosci. 36, 2213–2228. doi: 10.1111/j.1460-9568.2012.08108.x

Spacek, M. A., Blanche, T., and Swindale, N. (2009). Python for large-scale electrophysiology. Front. Neuroinform. 2:9. doi: 10.3389/neuro.11.009.2008

Srinivasa, N., and Cruz-Albrecht, J. (2012). Neuromorphic adaptive plastic scalable electronics: analog learning systems. IEEE Pulse 3, 51–56. doi: 10.1109/MPUL.2011.2175639

Stepanov, A., and Lee, M. (1995). The Standard Template Library, Vol. 1501. Palo Alto, CA: Hewlett Packard Laboratories.

Sutter, H., and Alexandrescu, A. (2004). C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Upper Saddle River, NJ: Pearson Education.

Thibeault, C. M., and Srinivasa, N. (2013). Using a hybrid neuron in physiological inspired models of the basal ganglia. Front. Comput. Neurosci. 7:88. doi: 10.3389/fnncom.2013.00088
Walters, J. R., and Bergstrom, D. A. (2010). “Synchronous activity in basal ganglia circuits,” in Handbook of Basal Ganglia Structure and Function. Handbook of behavioral neuroscience, Vol. 20, eds H. Steiner and K. Y. Tseng (Amsterdam: Elsevier), 429–443. doi: 10.1016/B978-0-12-374767-9.00025-1

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 01 November 2013; accepted: 10 February 2014; published online: 05 March 2014.

Citation: Thibeault CM, O’Brien MJ and Srinivasa N (2014) Analyzing large-scale spiking neural data with HRLAnalysis™. Front. Neuroinform. 8:17. doi: 10.3389/fninf.2014.00017

This article was submitted to the journal Frontiers in Neuroinformatics.

Copyright © 2014 HRL Laboratories LLC. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) or licensor are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.