LSCE Project Documentation

Brought to you by:

Pedro Rittner

Detian Shi

Nicholas Beaumont

Leina Sha

Danielle Lee

Anthony Chen

Prerequisites

The LSCE software requires that the user have the following programs and libraries installed on his/her machine prior to using the software:

- Python 2.7.0 or later
- Numpy 1.5 or later
- Scipy 0.8 or later
- Matplotlib 1.0 or later
- h5py 2.0 or later
- HDF5 1.8 or later (note that this is usually a requirement for installing h5py)

All of these libraries, with the exception of the HDF5 library, can be installed via PIP or your Linux distribution's package manager (e.g. yum on RPM-based systems or apt-get in Debian-based systems). If you are using Windows XP SP 3 or later, please consider downloading Python(x,y) (https://code.google.com/p/pythonxy/) in order to avoid dependency issues.

An example of how to install these libraries in Ubuntu 12.04 LTS (as of Dec 10, 2013) from the command line/terminal follows:

```
sudo apt-get install python-h5py python-numpy python-scipy python-matplotlib
```

Disclaimer

Note that this document is provided in the form of a User Manual as this was determined to be the most effective means of documenting the software for the client. Documentation of the code itself is provided by the PEP8-compliant docstrings included in the source code. Therefore, this User Manual attempts to document the features of the software from the perspective of an end user attempting to use this software in a practical application.

Data Importer

The Data Importer module in this project is designed to import ".raw" data files created by Multichannel Systems' "MC_DataTool" as a series of numpy arrays representing time-continuous data from each electrode in the device.

Importers are extremely specific to the type and format of the data they are importing. Therefore, the only requirement of an importer is that it creates a directory of .npy files which represent individual datasets. An optional configuration file may also be generated, though the user might wish to specify this by hand instead. See the Data Formatter section for more information on configuration files.

Usage of the importer (appropriately called `Importer`) for the 8x8-minus-corners electrode data follows a simple usage pattern:

```
>> import Importer

>> Importer.loadFromRaw("/media/Data/", numFiles=6, type='slice2_', Fs=20e3, saveMat=False)
```

The first (and only required) argument is the directory in which the RAW data files can be found. `numFiles` is the number of time slices. `type` is the prefix of the file type, which also determines the output name. In this case, resulting files might be therefore be named `/media/Data/slice2_/Electrode_12_master.npy` for the above example. `Fs` determines the sampling frequency rate in Hz. Finally, `saveMat` determines whether Matlab .mat or .npy files will be outputted. This is provided for legacy purposes only; the entirety of the LSCE software is built around .npy datasets.

Once the above code has executed, the data directory specified in the call to `loadFromRaw` will contain the parsed data, ready for being passed to the Data Formatter as described in the section below.

Data Formatter

The Data Formatter module is designed to be the most general tool in this project, and is designed to gather multiple extremely large numpy array dumps together in one HDF file such that they can be manipulated and viewed in a meaningful way.

The `DataFormatter.formatData` method is the primary way to invoke this module, which when given a source folder, destination file, and optionally the name of a configuration file, will copy every numpy array dump (.npy file) in the source folder into a 'raw_data' group inside of the destination hdf5-format file.

The datasets created in the hdf file will have the same name as their source file, and come with two metadata attributes by default, namely their data type and shape.

Additional metadata can be specified both for individual datasets and for the raw_data group as a whole by including a configuration file in the source directory. The configuration file should consist of one file whose name is specified in the function call ("config.ini" is checked for by default). The file should have the following format:

```
[raw_data]

sampling_rate = 60

time_taken = 5pm, 12/20/2012

interesting_times = 2:50, 10:32, 43:56

researcher_comment = started w carbachol up top but with aCSF in tubes

[Electrode_44_master]

interesting_times = 485, 1020, 1420
```

In the above file, [raw_data] refers to data that will be included in the attributes of the raw_data group within the created hdf5 file. [Electrode_44_master] in this case refers to a single file within the directory by the same name, and data assignments following that heading will be added to that dataset's individual attributes. Note that all metadata is optional: you may include any non-code data string. In fact, it is not necessary to pass a configuration file at all; the argument is completely optional. The "interesting_times" and "time_taken" metadata fields are included in this dummy configuration file as examples of the variety of formats that metadata values can take.

We now present a simple example of how to use the Data Formatter module. Let's assume that our imported data is in the directory "E:\LSCE\slice2" (the 'imported data' being the .npy files we imported using the Importer). Let's further assume that we wish to call our resulting HDF5 file "my_hdf5_file" and that we have a configuration file called "my_config.ini". An example call to this follows:

```
>> import DataFormatter

>> import hd5py

>> DataFormatter.formatData("E:\\LSCE\\slice2",
```

```
"C:\\ThesisData\\my_hdf5_file", config="my_config.ini")

>> hdf5_file_object = h5py.File("fulldata.hdf5", "r+")

# Do things here, close once you are finished

>> hdf5_file_object.close()
```

An important note from the call above: the Data Formatter will, as per PEP8 guidelines, do its best to follow any absolute or relative paths you give it for either the data directory, hdf5 file, and configuration file. The above example gave the absolute paths for the data directory and hdf5 file, and the configuration file was in the current working directory of the command line (since no absolute path was given it was assumed to be relative).

From this point we refer to the h5py module documentation (http://www.h5py.org/docs/) for how to use the `hdf5_file_object` we have created. Of paramount importance are the `flush` and `close` functions. `flush` will write the changes to the file withouting closing it. `close` will flush and then close the file object. Treat these file objects as you would any file object in Python and follow the conventions in the standard library, making sure that you always close the hdf5 file object after you finish using it either in the command line or in a script; failure to do so *will* lead to undefined behavior.

Data Analysis

The data analysis module in this project is designed to allow users to safely analyze data in a sandboxed environment. Specifically, any analysis and experimentation is performed in an isolated environment, protecting the integrity of the original data in the HDF5 file. The data structure representation of the sandboxed environment is *data_analysis* object, found in DataAnalysis.py.

The *data_analysis* object stores a file descriptor to the formatted HDF5 file, meta-data found in the HDF5 file relating to data analysis applications and a reference to a staged dataset. These concepts are explained in the following example of a possible workflow.

```
>>> from DataAnalysis import data_analysis

>>> sandbox = data_analysis('data.hdf5')

File 'data.hdf5' has been loaded.

>>> sandbox.load_dataset('Electrode_12_master','raw_data')

Dataset Electrode_12_master loaded and staged as 2013-12-10 13:02:24.878929.

>>> sandbox.rename_dataset('temporary_data')

Dataset 2013-12-10 13:02:24.878929 renamed temporary_data.

>>> sandbox.run_analysis(example_filter, sampling_rate = True)

>>> sandbox.save_dataset('analyzed_data')

Dataset temporary_data saved and renamed analyzed_data.

Dataset analyzed_data staged.
```

In the example, after importing the *data_analysis* function from the DataAnalysis.py module, we create a *data_analysis* object called `sandbox`. This is the data analysis object initialized with a reference to the HDF5 file to be analyzed. Specifying the filename in the initialization of the *data_analysis* is not required, a file can be loaded at a later time using the method *load_file*.

After the *data_analysis* object is initialized and the file is loaded a dataset is staged using the method *load_dataset*. When the method *load_dataset* is called, the specified dataset is copied and moved to a temporary group within the HDF5 file. On system exit the group and all datasets within it will be removed. Following the sandbox pattern, staged datasets never reference original datasets, only copies. By default when a dataset is loaded it is given a timestamped name. This behavior can be changed by specifying a rename as a parameter in the *load_dataset* method, or the staged dataset can be renamed at a later time using the method *rename_dataset*. A *data_analysis* object can only have one staged file at a time, and any analysis functions operate using the staged data.

In this example a non-existent analysis function *example_filter* is run on the staged data.

Data Visualization

The Data Visualization module in this project is implemented specifically for an 8x8 arrangement of electrodes (8 rows of electrodes with 8 columns) with the corner electrodes missing. For future use, this implementation may be changed / extended by changing the constant variables in the corresponding datavisualization.py file. (See comments in file)

Implementation

The Data Visualization module displays the electrode data in two main views.

View 1 - Viewing all electrode data concurrently

View 1 is generated for a general viewing purpose. It will help the users identify patterns in the data for the electrodes as a whole and enable specific electrodes of interest to be identified.

View 1 graphs data for all electrodes concurrently. This view generates 60 graphs (8x8 with the corners missing) arranged in the way the electrodes were placed (ie electrode in column 1 row 1 would be graphed in column 1 row 1). Since the data is large and cannot fit on one graph, this view is scrollable. The users can stipulate what time window they want each graph to span (i.e. one view of the graph only views 3 seconds worth of data) and then scroll the graph to see data for later times. When the view is scrolled each graph for every individual electrode is scrolled concurrently. This is so that users can identify electrodes of interest by comparing their data to the overall electrode data. There are no axis labels or ticks on the graphs because this slows performance. In addition, this level of detail is not required for the purpose of assessing general trends and identifying electrodes of interest.

When you click on an individual graph in View 1, you will be brought to View 2, with the time window at precisely the time specified by the scroll bar in View 1

View 2 - Viewing specific electrode data

View 2 is generated so that users can view data for electrodes of interest. This view provides an in depth look at electrode data.

This view has axis labeled and ticks on the graph. It also has zoom functionality, movement functionality (enabling the focus of the graph to be moved) and saving functionality.

How To Use

The data visualization requires data in the form of 2D arrays. To achieve this, the data visualization should be used after the data importer and data formatter. Here is an example of how the different components should be used in conjunction (note that lines with # are comments, not code):

```
>> import Importer
```

```
>> import hd5py
>> import DataFormatter
>> import datavisualization
# This imports the data
>> Importer.loadFromRaw("E:\\LSCE\\110112")
# This formats the data from slice2_ and saves it as "fulldata.hdf5"
>> DataFormatter.formatData("E:\\LSCE\\110112\\slice2_", "fulldata")
# Opens the data file that was just created as an hdf5 file object
>> tmp = h5py.File("fulldata.hdf5", "r+")
# Empty dataset list we will use to plot
>> data = []
# Append all the raw datasets
>> for dataset in tmp["raw_data"].keys():
      data.append(tmp["raw_data"][dataset])
# This will visualize an 8x8 data grid based off of the dataset we built
>> datavisualization.analyze8x8data(data=data, samprate=1000, time=5)
# Close the file once we are done with it
>> tmp.close()
```

Functions

Here are a list of relevant functions in the `datavisualization` module that can be executed (for detailed implementation see comments in file):

`analyze8x8data(data, time=1, samprate=2)`

Function which produces a visualization of 8x8 electrode data with a main view (graph of each electrode's data, arranged together according to the electrode positions) and zoom in view (graph of single electrode data).

Data = 2D Array of y values to be plotted

Time (in seconds) = the amount of time the graph should span in each window should be passed in as an integer.

Samprate = sampling rate, i.e how many data samples per second should be passed in as an integer

`analyze8x8Group(data, time=1, samprate=2)`

Function which produces a visualization of 8x8 electrode data with a main view (graph of each electrode's data, arranged together according to the electrode positions) and zoom in view (graph of single electrode data).

Data = 2D Array of y values to be plotted

Time (in seconds) = the amount of time the graph should span in each window should be passed in as an integer.

Samprate = sampling rate, i.e how many data samples per second should be passed in as an integer

`analyzesingle(data, time, samprate)`

Function which produces visualization of single electrode data.

Data = Array of y values to be plotted

Time (in seconds) = the amount of time the graph should span in each window should be passed in as an integer.

Samprate = sampling rate, ie how many data samples per second should be passed in as an integer.