MountainLab overview for SCDA - June 2016

Jeremy Magland and ALex Barnett

June 14, 2016

1 MountainLab Components

Component	Description	Where it runs	Dependency
Matlab	Processing and Visualization within matlab environment	Workstation	Linux or OS X. Matlab, MountainSort, MountainView.
Matlab wrappers	Matlab wrappers to MountainSort and MountainView. Can be mixed with the pure matlab component.	Workstation, Linux or OS X	Matlab, MountainSort, MountainView
MountainSort	Spike-sorting specific processing routines (Filter, whiten, cluster, etc.)	Workstation or Server, Linux or OS X	C++/Qt5, LAPACK
MountainProcess	Processing engine (spike-sorting independent). Queue and run scripts and processes. Batch processing.	Usually Server, Linux	C++/Qt5
MountainView	GUI for visualizing raw data and spike sorting results. Either locally or remotely.	Workstation, Linux or OS X	C++/Qt5
MountainBrowser	Desktop GUI for browsing spike sorting result. This component will be replaced by a true website / HTML5 solution	Workstation, Linux or OS X	C++/Qt5
mpserver	Queue processing scripts on server	Server, Linux	nodejs
mdaserver	Serve chunks of .mda files	Server, Linux	nodejs
mbserver	Serve .json files	Server, Linux	nodejs
mlproxy	Proxy server for mpserver, mdaserver, mbserver	Server, Linux	nodejs

1.1 Conventions for spike sorting data

The **raw dataset** (aka timeseries) is assumed to be a $M \times N$ array of voltage measurements, where M is the number of channels (electrodes) and N is the number of timepoints. Unless otherwise specified, event times are in index units.

The **sample rate** is the rate at which the original data was acquired. The samplerate variable is always in units of Hz. So samplerate=30000 means that timepoint 30,000 occurs at acquisition time 1 second.

Generally speaking, we use the following conventions:

M number of channels or electrodes

N number of timepoints in the timeseries

K number of clusters / spike types

L number of events / spikes

T length of a clip (in timepoints) / time window around a spike (typically 100 timepoints)

One-based indexing is used for timepoints, channels, and cluster labels. Thus the first channel is channel 1, the first timepoint is timepoint 1, and the first cluster is 1. A label of 0 represents an unclassified spike event. The **center of a clip** of size T is always at the integer part of $(T+1)/2$ using one-based indexing.

The output of sorting is stored in a **"firings"** matrix (firings.mda) of dimensions $R \times L$ where $R \geq 3$ and L is the number of events (double precision). Thus each column corresponds to a detected firing event. Information in the rows of the firings matrix are defined as follows:

First row (optional) Primary channel number of the spike type, e.g., the channel where the average spike shape has the largest peak. If this information is not provided (by the sorter) this row may be filled with zeros.

Second row (mandatory) Event times, or timepoints where the events were detected. These may have a fractional component.

Third row (mandatory) Integer labels assigning membership to a cluster or spike type. A value of zero indicates an unclassified (e.g., noise or outlier) event.

Fourth row (optional) Peak amplitudes, determined by the sorting algorithm.

Fifth row (optional) Outlier scores as determined by the sorting algorithm. The larger the score, the more outlier-ish.

Sixth row (optional) Detectability scores as determined by the sorting algorithm. Spike events that are close to noise are given a low score. Roughly speaking, a score of 6 would correspond to an event that is 6 standard deviations above the noise.

Seventh row and up (optional)) Unused for now.

Electrode geometry (geom.csv or geom.mda) is stored in a $2 \times M$ or $3 \times N$ array of 2D or 3D coordinates.

1.2 Matlab and wrappers

There are four types of matlab functions for performing spike sorting and visualization.

In-memory processing conventional Matlab processing functions

In-memory visualization launches Matlab figures

Wrappers to MountainProcess/MountainSort processing routines operating on files. Uses system calls to mountainprocess.

Wrappers to MountainView visualization of raw data and results, operating on files. Uses (detached) system calls to mountainview.

These four types can be used together in driver scripts via the following functions which convert between in-memory arrays and .mda files: arrayify.m, pathify8.m, pathify16.m, pathify32.m, pathify64.m. These are convenient wrappers to readmda.m and writemda*.m that return the inputs when a conversion is not needed. For example if X is an array, then arrayify(X) will simply return X.

1.3 Multi-dimensional array file format (.mda)

The .mda file format was created by Jeremy, while at Felix Wehrli's NMR lab Penn, as a simple method for storing multi-dimensional arrays of numbers. Of course the simplest way would be to store the array as a raw binary file, but the problem with this is that fundamental information required to read the data is missing – i.e., the number and sizes of the dimensions as well as the data type (e.g., float32, int16, byte, complex float, etc).

One way to handle this is to distribute a text file along with the binary raw file that contains this information – but that's inconvenient. Another possibility is to encode this information in the file name – but that's a bad idea.

A better solution is to provide this information in a header (beginning portion of the data) immediately preceding the raw portion. This is the approach of the Nifti (.nii) file format. However, the pressing questions are (a) in what format should the information be stored? and (b) how much extra information should be provided?

The problem (in my opinion) with the .nii format is that the header is just complex enough to require libraries for reading/writing – one library for each programming language (C/C++, MATLAB, Python, Java, etc). This is the same problem with MATLAB's .mat format (although a lot more difficult to read/write outside of MATLAB itself). This is true for many other formats as well (.hdf5 and DICOM).

There is another problem with all of these formats – they all support various collections of meta information. Each software therefore produces files with different extra information that may or may not be essential for other software reading the files. In other words, there is no guarantee on which information is provided within a particular .nii, .mat or .hdf5 file. In many cases, more than one multi-dimensional array may be included, creating ambiguity as to which should be used.

Motivated by the above issues, the .mda format conforms to the following principles:

- Each .mda file contains one and only one multi-dimensional array of byte, integer, real, or complex numbers.
- The .mda file contains a small well-defined header containing only the minimal information required to read the array, namely the number and size of the dimensions as well as the data format of the entries.
- Immediately following the header, the data of the multi-dimensional array is stored in raw binary format.

The .mda file format has evolved slightly over time (for example the first version only supported complex numbers), so please forgive the few arbitrary choices.

- The first four bytes contains a 32-bit signed integer containing a negative number representing the data format (-1 is complex float32, -2 is byte, -3 is float32, -4 is int16, -5 is int32, and -6 is uint16).
- The next four bytes contains a 32-bit signed integer representing the number of bytes in each entry (okay a bit redundant, I know).
- The next four bytes contains a 32-bit signed integer representing the number of dimensions (num_dims should be between 1 and 50).
- The next 4*num_dims bytes contains a list of signed 32-bit integers representing the size of each of the dimensions.
- That's it! Next comes the raw data.

Matlab functions for reading and writing this format are here: `https://github.com/magland/mountainlab/tree/master/ma`

C++ classes for dealing with .mda files, multi-dimensional arrays in memory, and remote access to .mda files are here: `https://github.com/magland/mountainlab/tree/master/common/mda`

1.4 MountainSort

MountainSort is a command-line program for performing low-level spike sorting routines. These functions operate on files in chunks, and therefore use very little RAM. Here are some examples:

```
> mountainsort bandpass_filter --timeseries=raw.mda &&
  --timeseries_out=filtered.mda --samplerate=30000 &&
  --freq_min=300 --freq_max=5000 --freq_wid=1000

> mountainsort mask_out_artifacts --timeseries=filtered.mda &&
  --timeseries_out=filtered2.mda --threshold=3 --interval_size=200

> mountainsort whiten --timeseries=filtered2.mda &&
  --timeseries_out=whitened.mda

> mountainsort detect --timeseries=whitened.mda &&
  --detect_out=detect.mda --clip_size=100 --detect_interval=10 &&
  --detect_threshold=3.5 --sign=-1 --individual_channels=1

> mountainsort branch_cluster_v2 --timeseries=whitened.mda &&
  --adjacency_matrix=AM.mda --firings_out=firings.mda &&
  --clip_size=100 --min_shell_size=150 --shell_increment=3 &&
  --num_features=10 --detect_interval=10

...
```

All processes are defined based on the following data:

processor_name The name or ID of the processor (e.g., bandpass_filter)

inputs a collection of named input file paths

outputs a collection of named output file paths

parameters a collection of named input parameters (numbers or strings)

For the most part all inputs and outputs will be multi-dimensional arrays stored in the .mda file format. Information about this format with links to tools for converting raw data into this format are found below.

1.5 MountainView

MountainView is the desktop user interface for interactive visualization. Results may either reside on the local machine or on a remote server – the functionality is the same. Detailed information on its usage can be found on the MountainLab forum in this article: `https://mountainlab.vbulletin.net/articles/23-introduction-to-mountai`

1.6 MountainProcess

MountainProcess is the core of MountainLab and provides a framework for processing data on a client workstation or server. It is completely independent of spike sorting. Plug-in processors may be created in any language as they are simply executables obeying a set of rules (specified below). It automatically handles batch and parallel processing using an intelligent system for queueing and running scripts and processes as resources become available. A crucial feature is provenance tracking and the ability to determine whether a particular process has already been performed. Processing may be performed either using a command-line/bash procedures or JavaScript-specified pipelines.

MountainProcess provides the following capabilities

Plug-in processors executables in any language

Batch and parallel processing intelligent queueing system for scripts and processes

Provenance tracking and non-redundant process execution

Secure scripting using JavaScript

To take advantage of provenance tracking and queuing (batch) functionality all of the above mountainsort procedures may also be performed using mountainprocess from the command line. For example, bandpass filtering may be accomplished using

```
> mountainprocess run-process bandpass_filter --timeseries=raw.mda &&
  --timeseries_out=filtered.mda --samplerate=30000 &&
  --freq_min=300 --freq_max=5000 --freq_wid=1000
```

or

```
> mountainprocess queue-process bandpass_filter --timeseries=raw.mda &&
  --timeseries_out=filtered.mda --samplerate=30000 &&
  --freq_min=300 --freq_max=5000 --freq_wid=1000
```

The latter (queuing) command is useful for batch and parallel processing and requires the mountainprocess daemon to be running in the background. This daemon may be launched via

```
> mountainprocess daemon-start
```

The recommend way to run the daemon in the background is using a tmux session:

```
> tmux new -s mpdaemon
> mountainprocess daemon-start
```

Use [Ctrl b+d] to exit the tmux session, or simply close the terminal. Later you can attach via:

```
> tmux attach -t mpdaemon
```

If you are also running the web servers, the start_labcomputer.sh will also run the daemon (see below).

1.7 Scripting processing pipelines

The recommended alternative to writing bash scripts is to assemble the processing pipeline using JavaScript. MountainProcess can be used to queue or run these scripts. There are a number of advantages in terms of syntax, flexibility and security. First, convenience functions may easily be used to wrap commands. For example one way to perform pre-processing could be:

```
X = bandpass_filter(X, opts);
X = mask_out_artifacts(X, opts);
X = whiten(X);
```

Another way, which is more intuitive in terms of intermediate files and multiple output parameters, is

```
bandpass_filter(raw, '@pre1', opts);
mask_out_artifacts('@pre1', '@pre1b', opts);
whiten('@pre1b', '@pre2');
```

In this case the strings beginning with "@" are placeholders for intermediate files. The framework will automatically replace these strings with temporary file names that depend uniquely on the checksums of the input parameters. More details about unique file naming will be described later.

It is important to note that the structure of JavaScript function wrappers (including the above "@" intermediate file strategy) is completely independent of mountainprocess. There is a great deal of flexibility in how pipeline generation is performed.

The API for scripting is minimalistic and involves the following low-level interface methods:

MP.fileChecksum(fname) returns the SHA-1 hash of a local file (needed for purposes described below)

MP.createTemporaryFileName(code) returns the path of a temporary file, in a location managed by mountainprocess, which is uniquely determined by the input string.

MP.runPipeline(json) queues a pipeline of processes encoded by JSON text. This is simply a list of processes to execute, each with a processor name, input/output file names, and input parameters. The syntax is detailed below.

MP.log(message) writes a log message string to the console, for purposes of monitoring and debugging.

1.8 Plug-in processor libraries

In MountainProcess, all data analysis is performed via plugin-processors library which are simply executables obeying a set of rules. First, the executable should have a .mp extension and should be placed in the mountainlab/mountainprocess/processors directory (although it is possible to configure mountainprocess to look in other directories as well). For spike sorting the processor library is mountainsort.mp. When this executable is run as

```
> mountainsort spec
```

it should output JSON text that provides a list of available processors together with their specifications. For example, the following is a reduced version of the mountainsort.mp spec output:

```
{
    "processors": [
        {"description": "","exe_command": "mountainsort.mp bandpass_filter $(arguments)",
            "inputs": [{"name": "timeseries"}],
            "name": "bandpass_filter",
            "outputs": [{"name": "timeseries_out"}],
            "parameters": [
                    {"name": "samplerate","optional": false},
                    {"name": "freq_min","optional": false},
                    {"name": "freq_max","optional": false},
                    {"name": "freq_wid","optional": false},
                {"name": "processing_chunk_size","optional": true},
                {"name": "chunk_overlap_size","optional": true}
            ],
            "version": "0.1"
        },
        {"description": "","exe_command": "mountainsort.mp whiten $(arguments)",
            "inputs": [
                {"name": "timeseries"}
            ],
            "name": "whiten",
            "outputs": [{"name": "timeseries_out"}],
            "parameters": [],
            "version": "0.1"
        }
    ]
}
```

As outlined above, each processor has a name, and set of named input and output files, and a set of named input parameters (some optional). When running a process the arguments are passed in (by mountainprocess) according to the syntax from the section above (for mountainprocess). Note again the processing libraries are simply executable files and can therefore be created using any language.

1.9 File I/O and processing files in chunks

Whenever possible, .mda files should be processed in chunks to avoid loading unnecessary data into memory. This can prevent crashes and allows multiple processes to run simultaneously. For C++ there are a few classes to facilitate this.

Mda in-memory read/write array

DiskReadMda on-disk readonly array, or readonly access to remote .mda file. Read data in chunks.

DiskWriteMda on-disk writeonly array. Write data in chunks.

Here is an example C++ function that extracts clips from a readonly .mda file and returns an in-memory array of clips:

```
Mda extract_clips(DiskReadMda &X, const QList<double> &times, int clip_size)
{
    long M=X.N1(), N=X.N2();
    int T=clip_size;
    long L=times.count();
    int Tmid=(int)((T+1)/2)-1;
    Mda clips(M,T,L);
    for (long i=0; i<L; i++) {
        long t1=(int)times[i]-Tmid;
        long t2=t1+T-1;
        if ((t1>=0)&&(t2<N)) {
            Mda tmp;
            X.readChunk(tmp,0,t1,M,T);
            for (int t=0; t<T; t++) {
                for (int m=0; m<M; m++) {
                    clips.set(tmp.get(m,t),m,t,i);
                }
            }
        }
    }
    return clips;
}
```

1.10 Executing processing scripts remotely

There are two situations where it would be desirable to launch processing scripts from a remote computer. The first would be to perform a batch of processing from the convenience of a web browser. The second is for the viewer (e.g., MountainView) to pull down results derived from datasets on the server. For example, the viewer needs to retrieve the average waveforms for each cluster in order to render in the cluster detail widget. Since the user is free to select various options including the clip size, these data should be generated on demand. It is not practical for the computation to be done on the client end since this would require the entire timeseries to be downloaded. Instead the process should be executed on the server. The desired processing script is therefore sent as a http request to mpserver which forwards the script to the mpdaemon for queuing. The server only responds once the script has run and the output files have been created. The GUI may then read the result in chunks for client-side rendering.

If the computation is taking too long, the user may cancel the request. When the http connection is aborted, a chain of events occurs that terminate the script and any associated processes.

1.11 Caching and Hashing

On the one hand we want the software to just work. On the other hand we want it to be fast. The latter requirement implies that we don't want to perform the same resource-intensive operations more than once. For example, if a 30MB file has been downloaded to the GUI, we will want to cache it locally. Similarly if we perform a 20 second calculation on the server (e.g., extract clips and calculate PCA features) then we don't want to ever repeat that same calculation. The capability of recording what has been done and caching the files needed to automatically load up those results at a later time could lead to complex, less maintainable code, software bugs, and in the worse case, incorrect results. The code could start to look like that previous sentence – we don't want that to happen.

Our approach is program as though we are doing it the long way (with no concern for redundant computations or downloads). Then we add "caching and hashing" layers on top to enable shortcuts whenever possible. We try to avoid complex provenance databases and other structures that have the potential to become out of sync or buggy. Instead we build up a series of "safe" layers, each improving the efficiency by being aware (and taking advantage) of the shortcuts available at the lower layers.

For mountainprocess, the lowest layer is non-redundant execution of processes (we assume that all processors are deterministic). Each process is assigned a hash code that is uniquely depends on the processor name and version, paths and checksums of the input files (footnote later), and input parameters. When the process has completed, the paths and checksums of the output files are associated with the process hash code and recorded in an internal temporary directory. Later, if the same process is to be run, mountainprocess will check whether the process has previously run, and whether the output files match those of the previous run. If so, the execution is skipped.

This first layer is only effective if a process is run with the same set of output paths. As an example, suppose that the GUI needs to extract 200 clips from a large dataset. A request will be sent to the server that will queue a script that will ultimately lead to an "extract_clips" process being, creating a temporary output file. The GUI will then download the temporary file. If at a later time the user requests to view the same 200 clips, then we don't want to repeat this operation on the server. But this requires that we use the same path for the temporary output file. Therefore, we ideally want the name of the path to depend uniquely on the processor name and version, checksum of the input files, and the input parameters.

At this point we have two choices: the loosely coupled (correct) way and the tightly coupled (incorrect) way. In the tightly coupled way, the client (GUI) will say... "Yikes, I need to know the checksum and full path of the timeseries file, and the version of the extract_clips processor". The programmer (me) will then need to expand the API and start doing all kinds of consistency and security checks. The alternative (loosely coupled, correct) way is for the client (GUI) to say... "Well, let me just guess at what the temporary file should be called. There's a good chance that everything is going to work out just fine. And if not, what's the worst thing that could happen? The server will have to do a little extra work and the user will just need to hang tight." In the latter strategy the API stays minimal and the client/server stay decoupled. And, in fact, things do work out just fine!

In addition to saving CPU time on the server, we would also like to save bandwidth and download time. This means client-side caching of downloaded files. The difficulty here is that the .mda files on the server may change (for example if a process was re-run with a new version of the processor, overwriting the same output file). To handle this, all .mda files are downloaded in chunks by making calls to mdaserver (a nodejs web server). When a request for a chunk of the array is received, the server extracts the data (via mdachunk) and saves the result in a temporary .mda file whose name depends uniquely on the checksum of the larger array and the properties of the chunk being extracted. The http response then contains the url needed to retrieve the extracted chunk. This url is guaranteed to be static, so client-side caching may be safely employed. In order for this to be effective, however, the GUI must be consistent in the chunk sizes (and locations) it requests. For example, if I need to retrieve entries 100 through 957 in a vector, it is better to just download the entire chunk that includes these entries. Note that server-side caching is also possible, so that the same chunk only needs to be extracted once.

The above techniques require the development of low-level tools. File checksums need to be computed efficiently (and non-redundantly). Temporary files must be managed efficiently, with old files being deleted when the size of the temporary directory increases past a configurable limit.

1.12 Web servers: mpserver, mdaserver, mbserver

To enable remote access to spike sorting results from client workstations, three servers must be running. These can be conveniently accessed through a single port by using the mlproxy server. The servers are all programmed in nodejs.

The mountainprocess server (mpserver) allows processing scripts to be submitted from a remote client. For example, the GUI may need to extract a collection of clips from a very large timeseries dataset. The API is minimal and consists of only one operation: queueScript. The client passes the text of the script to the server, which is then queued for processing. After the script executes, and all of the associated processes have finished, the http response is returned with information about whether the processing succeeded. Note that the results of the processing are not actually returned. Instead, the script will typically create temporary files, with names known to the client, so that the output data may be retrieved using subsequent calls to mdaserver.

The multi-dimensional array server (mdaserver) is used to ....

2 Mathematical algorithms

2.1 Bandpass filtering

We now use erf (error function) for smooth (in fact analytic) roll-offs at high and low frequencies, passing through 0.5 at the low and high points (`freq_min` and `freq_max`). We then take the square-root of this desired intensity filter to get the amplitude filter $a(f)$ used to multiply in frequency space. This creates -3dB points at the desired low and high values, and minimal ringing. We prefer these to standard filters.

Specifically, if low pass (high-frequency roll-off) is requested, the multiplier in frequency space f is

$$a(f) = \left[\left(1 - \operatorname{erf} \frac{|f| - f_{\max}}{w}\right)/2\right]^{1/2} \tag{1}$$

where w is the width `freq_wid` (a sensible value being 1000 Hz, which is set as a default in the MATLAB interfaces). Note that $|f|$ is required since the DFT grid includes negative frequencies. If a high-pass is requested,

$$a(f) = \left[\left(1 + \operatorname{erf} r \frac{|f| - f_{\min}}{f_{\min}}\right)/2\right]^{1/2} \tag{2}$$

where r is the parameter deciding the amount by which very low frequencies are killed (currently hard-coded as `relwid=3.0` which provides around 10^{-5} filtering at $f \approx 0$). A bandpass is the product of the above two formulae. The DC (zero frequency) component is killed exactly whenever high-pass is requested, ie $a(0) = 0$.