# New Proposed Standard for Matlab

## Introduction

Matlab is becoming a critical tool in SNL-E. Currently, the server contains a standard set of code which includes several important functions and clear documentation. However, some key elements are missing and there is no clear path for expansion. As a result, several authors have developed private schemes to meet their individual needs, but these code bases are mutually incompatible.

We here propose a set of universal standards for all present and future Matlab code to follow. The proposal encompasses all information accessible in Matlab. It resolves ambiguities and redundancies of the current schemes, and creates a simpler and easier workflow for users. In addition, the proposed standard provides a modular framework that can easily be expanded to meet future needs. If this proposal is carefully adopted, all our code will work together and everyone will benefit.

To adopt this new proposal, existing code will need to be updated, both on the server and in our private collections. We believe the cost of this one-time upgrade is far outweighed by the new proposal's simplicity, modularity, and expandability.

This proposal has three main sections: 1) file storage, 2) nomenclature and data types, 3) function style and standard functions.

## 1. File Storage

The proposed standard consists of Matlab functions and documentation. Both will be stored on the server in /snle/lab/Development/RRS/matlab-source/ . Matlab function text files will live in this directory, potentially within subdirectories. Backups will be in a folder called "backup" in the same directory. Backup functions will keep their file names but be located in a folder with the appropriate date. The documentation will be in a folder called "documentation", and will consist of text files and intaglio files.

Summary:

.../matlab-source/<function>.m
.../matlab-source/<function_folder>/<function>.m
.../matlab-source/backup/<date>/<function>.m
.../matlab-source/documentation/

## 2. Nomenclature and Data Types

The fundamental unit of the lab's data is a single recording. Obvius calls it a "series", in Vision it corresponds to a neurons file. Under this proposal, Matlab will store all information pertaining to a single recording in a struct called "dataset".

Below is a list of the fields in "dataset". For each field, the standard name is given along with a brief description of what it stores. These names were chosen to be as similar as possible to the standard currently on the server. Note that some pieces of information exist for every dataset (e.g. cell ids) while others do not (e.g. STAs). For the dataset-specific items listed below, we believe these are sufficiently common that they need to follow a standard nomenclature.

**nomenclature for "dataset"**

names specifying the dataset

| | | | |
|---|---|---|---|
| dataset. | names. | experiment | e.g. '2007-07-05-1' |
| | | condition | e.g. 'rf-0-mg' |
| | | neurons_path | e.g. '/snle/lab/Experiments/.../data000.neurons' |
| | | params_path | e.g. '/snle/lab/Experiments/.../data000.params' |
| | | ei_path | e.g. '/snle/lab/Experiments/.../data000.ei' |
| | | sta_path | e.g. '/snle/lab/Experiments/.../data000.sta' |

piece information

| | | | |
|---|---|---|---|
| dataset. | piece. | rig | 'A','B',etc |
| | | preparation | 'isolated','rpe' |
| | | map | Mx2 matrix of array map |
| | | aperture | Lx2 matrix of stimulus aperture map |
| | | eccentricity | temporal equivalent (mm) |
| | | distance | distance from fovea (mm) |
| | | angle | angle with respect to line between fovea and optic nerve (clock face?) |
| | | array | number of the recording array |
| | | electrode_map | format TBD, gives the spatial locations of all electrodes |

neurons file information

| | | |
|---|---|---|
| dataset. | cell_ids | Nx1 vector of cell ID's |
| | spikes | Nx1 cell array of spike times |
| | channels | Nx1 vector of electrodes |
| | triggers | column vector of trigger times |
| | duration | duration of recording (sec) |
| | seconds_per_sample | typically 0.00005 |

EIs

| | | |
|---|---|---|
| dataset. | eis. | format TBD, possibly a Nx1 cell array of XxYxT matrices |

stimulus information

| | | | |
|---|---|---|---|
| dataset. | stimulus. | type | string specifying stimulus type |
| | | ... | [following obvius conventions, e.g. rgb, interval, stixel_height] |

STAs

| | | | |
|---|---|---|---|
| dataset. | stas. | stas | Nx1 cell array of 4-D matrices (x,y,rgb,time) |
| | | summaries | Nx1 cell array of 2-D matrices (x,y), spatial summary of the STA |
| | | summary_type | string indicating how spatial summaries were computed (e.g. 'svd') |
| | | contours | format TBD |

vision classification & fits

| | | | |
|---|---|---|---|
| dataset. | vision. | cell_types | cell array specifying the classification, see below |
| | | sta_fits | Nx1 cell array of structures storing sta fits |

obvius classification & fits

| | | | |
|---|---|---|---|
| dataset. | obvius. | cell_types | cell array specifying the classification, see below |
| | | sta_fits | Nx1 cell array of structures storing sta fits |

"working list" of cell types

| | | |
|---|---|---|
| dataset. | cell_types | cell array specifying the classification, see below |

default values

| | | |
|---|---|---|
| dataset. | default_fits | default fits ('vision','obvius',etc), see below |

Below are several sections explaining the data types that are stored in "dataset". Each section gives an explanation of the new standard along with a detailed rationale.

**cell_id versus cell_number**

Currently, cells are commonly referenced by their cell id. While the cell id is convenient for calling external java functions, it is not convenient within Matlab. For example, to get the spike times of cell id 100, you need to first get the *cell number*, i.e. the ordinality in the list of cell ids:

```
cell_number = find( dataset.cell_ids == 100 );
dataset.spikes{cell_number}
```

For a long list of cells, translation is even more painful:

```
clear cell_numbers
for cc = 1:length(list_of_cell_ids)
    cell_numbers(cc) = find( dataset.cell_ids == list_of_cell_ids(cc) );
end
dataset.spikes{cell_numbers}
```

Under the new proposal, the cell number will become the standard reference for each cell. Note that the cell number is uniquely defined by the list of cell ids found in the neurons file.

This scheme has the potential drawback that translation is still required, albeit in the opposite direction (from number to id). Fortunately, the cell id is used less frequently, and translation is very easy, even for a long list:

```
cell_id = dataset.cell_ids(cell_number);
cell_ids = dataset.cell_ids(cell_numbers);
```

Thus the cell number convention is generally preferable to cell ids: it works well with commonly-used Matlab variables, and can be easily translated to call external java commands.

**new format for listing cell types**

Currently, cell types are stored in two variables, "cell_type" and "cell_types". This scheme suffers from three problems. First, the variable "cell_type" is redundant, which means any update to "cell_types" requires re-computing "cell_type". Second, cell names can not contain spaces or other reserved characters. Third, these variables do not permit a cell to belong to more than one type. Overlapping cell types can serve important purposes, e.g. having two cell types called "ON parasol" and "ON parasol high quality".

The new scheme solves all three problems with a single variable called "cell_types". It is a cell array of two-field structs, e.g.:

```
cell_types{1}.   name          "ON parasol"
                 cell_numbers  [1 2 5 6 8 10 15]
```

This variable will be stored as a field at several locations in "dataset" (see above). Note that this new scheme uses the cell number convention.

This scheme might seem to have drawbacks that make it less convenient for the user. These potential drawbacks are remedied by an important new function explained below ("get_cell_numbers").

We also propose a standard order for listing cell types: 1) ON parasol, 2) OFF parasol, 3) ON midget, 4) OFF midget, 5) SBC. If this standard is followed, the user will be able to call "dataset.cell_types{1}" and always get the ON parasol cells. Under the previous scheme ("cell_type"

and "cell_types"), the user needed to know the exact name of a cell type in order to reference it. These names vary, so the user needed to manually check whether it was "ON_parasol", "ON_PARASOL","on_parasol","on_y", etc. When cell types are listed in the proposed standard order, the user will be freed from having to specify the exact name.

**working list of cell types ("dataset.cell_types")**

What good are lots of cells if you can't classify them? As we all know, what really separates the Men [primate] from the Boys [salamander] is cell classification.

When a function or the user needs to know which cells belong to which type, the standard place to check is "dataset.cell_types". This field serves two important needs:

1) The server contains two cell classifications (from vision and obvius) which sometimes conflict. It is up to the user to decide which classification is right. Once decided, the user can put the classification into the field "dataset.cell_types" using a simple function ("use_classification", see below). By storing the preferred classification in a standard place, functions will not require an extra argument specifying to use the obvius or vision classification. Of course, optional arguments to a function can still specify a particular classification.

2) The stored vision and obvius classifications are usually quite accurate. However, sometimes the user will want to modify the classification on the fly, and that's why a special field is needed to store the "working list" of cell types. For example, the user can omit cells that don't meet a certain quality criterion, or generate a new cell type such as "L-cone ON midgets". By making a standard place to store these temporary cell types, the user will not be tempted to invent new fields and modify functions to look there. Instead, all functions can simply use the "working list".

**default STA fits ("dataset.default_fits")**

There are also two versions of STA fits: vision & obvius. In most cases, the user will simply want to select one set of fits and use it for all analyses. The flag "dataset.default_fits" is a standard place to store this choice. Functions that operate on STA fits should check here to see which fits to use.

## 3. Function Style and Standard Functions

This section describes standard function usage, how to initially load values into "dataset", and details several important functions.

**Standard Usage**

Most high level functions should take the entire "dataset" variable as their primary argument, followed by required parameters and an optional struct of extra parameters. For example:

```
function my_function( dataset, <parameter_1>, ..., <parameter_n>, params )
```

Having functions read in the entire "dataset" variable is the heart of this proposal. This usage gives the function complete access to any needed data with a single argument, rather than a long list of arguments specifying each piece of information. In addition, the user does not need to know the internal data structure, thus creating an easier workflow which is less prone to error. Most importantly, if the author later decides that the function needs to access additional information, there's no need to pile on extra arguments that break previous usage.

For functions that save analysis results, the entire dataset variable should also be read out. For example:

```
function [dataset] = my_function( dataset, <parameter_1>, ..., <parameter_n>, params )

% perform analysis
...

% save results
dataset.new_field = <analysis result>
```

This usage also has the benefits described above: it creates a simple workflow, and allows for arbitrary expansion without breaking previous usage.

One potential drawback of this usage is that a poorly written function can corrupt the "dataset" variable. For example, a function might put incorrect information in one of the standard fields. Nevertheless, we believe the the risks of this usage are far outweighed by its benefits. As always, writers of code must be cautious and vigilant.

Moreover, a simple convention greatly reduces the risk of unintentional corruption: save results to temporary variables first, and only modify "dataset" at the end of the function:

```
function [dataset] = my_function( dataset, <parameter_1>, ..., <parameter_n>, params )

% first analysis
...
analysis_result_1 =  <result>

% second analysis
...
analysis_result_2 =  <result>

% save results
dataset.new_field_1 = analysis_result_1;
dataset.new_field_2 = analysis_result_2;
```

With this usage, any author can see at a glance exactly which fields the function modifies. This practice guards against the function modifying fields it shouldn't.

The argument "params" is optional for the user and it contains optional parameters. "params" is a struct whose usage is similar to the keyword arguments in obvius:

```
clear params
params.plot_outlines = true;
params.outline_color = 'k';
my_function( dataset, params )
```

Alternatively, everything can be done in one line:

```
my_function( dataset, struct('plot_outlines',true,'outline_color','k') )
```

Internally, the function will look for optional fields in "params". It will use any specified values, and otherwise revert to a default.

The "params" argument provides two primary benefits. First, it is a standard means to overload functions. Second, "params" can be used to expand a function's input without breaking previous usage or cluttering the list of arguments. The "params" argument should appear in every function.

Below, several concrete examples illustrate the proposed function style.

**load_data**

```
dataset = load_data( dataset, params )
```

This function puts all relevant information into "dataset", adding and filling in fields according

to the above nomenclature. Information is only loaded if the appropriate fields of "dataset.names" are non-empty. If a field is empty, the corresponding information is not loaded.

To load a dataset for the first time, the user specifies the parameters and calls "load_data". For example:

```
% specify parameters
dataset.names.experiment = '2007-07-05-1';
dataset.names.condition = 'rf-0-mg';
dataset.names.neurons_path = '/snle/lab/Experiments/.../data000.neurons';
dataset.names.params_path = '/snle/lab/Experiments/.../data000.params';
dataset.names.sta_path = '/snle/lab/Experiments/.../data000.sta';

% load dataset
dataset = load_data( dataset );
```

Alternatively, the arguments can be specified in one command:

```
dataset=load_data(struct('names',struct('experiment','2007-07-05-1',...
     'condition','rf-0-mg')) );
```

The primary advantage of this format is consistency, both inside of wrapper functions and externally for the user. For example, "load_data" might simply call lower level functions to load the various parts of the struct:

```
dataset = load_series( dataset, params );
dataset = load_neurons( dataset, params );
dataset = load_params( dataset, params );
dataset = load_stas( dataset, params );
```

This consistent usage also provides flexibility and modularity for the user. If, for example, the user has so far been working only with spike times, and wants to incorporate STAs, "load_stas" can be called without having to look up the file path again (and potentially get it wrong):

```
dataset = load_stas( dataset );
```

This function will seamlessly incorporate new information into the extisting struct, and all pre-existing fields, including the results of any analysis, will remain untouched.

Here, the optional set of parameters in "params" might indicate to NOT load a particular kind of data, even if the file exists. For example, the user might want only the cell IDs, but not the spikes times (which can take much longer to load):

```
dataset = load_neurons( dataset, struct('load_spikes',false) );
```

**plot_psth**

```
h = plot_psth( dataset, stard_time, end_time, params )
```

This hypothetical example illustrates an important purpose for the "params" argument. In addition to optional parameters such as bin size and figure location, it can contain a meta-parameter field ("params.meta"). If, for example, Martin needed very specific settings for plotting a psth in a paper, he could specify them inside the function "plot_psth". No other users would need to worry about it, and Martin could call the function like this:

```
h = plot_psth( dataset, 'ON parasol', stard_time, end_time, struct
('meta','martin_2008_02_nature_figure_1') )
```

**use_classification**

```
dataset = use_classification( dataset, new_classification )
```

This simple function puts the appropriate list of cell types into "dataset.cell_types". Here's the code:

```
function dataset = use_classification( dataset, new_classification )

switch new_classification
    case 'obvius'
        dataset.cell_types = dataset.obvius.cell_types;
    case 'vision'
        dataset.cell_types = dataset. vision.cell_types;
    otherwise
        error('Only ''obvius'' or ''vision'' cell classification can be used (not %s).',...
            new_classification)
end
```

**get_cell_numbers**

```
[cell_numbers, cell_type] = get_cell_numbers( dataset, cell_specification );
```

Functions frequently need to operate on more than one cell. There are several ways to specify a group of cells, and which specification is best varies for different functions. For example, a function which makes a collection of serial plots only needs to read in a list of cell numbers.

```
plot_ccfs( dataset, [1 2 4], ... )
```

In other cases, functions *do* need to know what kinds of cells they are operating on. For example, a function that computes the conformity ratio should display the name of the cell type alongside the result. If the function did not know the cell type, the user would need to manually keep track of which output corresponded to which input, a workflow that invites error.

```
compute_conformity_ratio( dataset, 'ON parasol', ... )
```

Finally, some functions need to read in several cell types. For example, a function that overlays the average RF profiles of several types needs a list of cell types.

```
compare_rf_profiles( dataset, {'ON parasol','OFF parasol'}, ... )
```

Specifying cells with such flexibility is important for users. However, it can be daunting for the code to handle so many cases.

This proposal provides complete flexibility with a single function, "get_cell_numbers". This function takes in one argument, "cell_specification", which can be a list of cells, the name of a cell type, or a list of names. It returns the desired cell numbers, along with the name(s) of the specified cell type(s), if applicable. Here is the form of the function, followed by some examples:

```
[cell_numbers, cell_type] = get_cell_numbers( dataset, cell_specification )


cell_specification = 'OFF parasol'  >>>

cell_numbers = [1 3 5 8 15]
cell_type = 'OFF parasol'
```

```
cell_specification = [1 2 4]  >>>

cell_numbers = [1 2 4]
cell_type = []  % if no name is specified, the field is empty


cell_specification = {'ON parasol','OFF parasol'}  >>>

% if a list of names is given, a list of numbers and
% names is returned as a cell array

cell_numbers{1} = [2 4 7 16]
cell_type{1} = 'ON parasol'
cell_numbers{2} = [1 3 5 8 15]
cell_type{2} = 'OFF parasol'


cell_specification =  {1,2}  >>>

% if a cell array of numbers is given, it is interpreted as
% the ordinal number of the desired cell type

cell_numbers{1} = [2 4 7 16]
cell_type{1} = 'ON parasol'
cell_numbers{2} = [1 3 5 8 15]
cell_type{2} = 'OFF parasol'


cell_specification =  {1}  >>>

% if there's only one element in the list, the function
% does not return a cell array

cell_numbers = [2 4 7 16]
cell_type = 'ON parasol'
```

For most functions, the first or second usage is appropriate. "get_cell_numbers" can be easily incorporated into existing functions by prepending a brief boilerplate:

```
my_simple_function( dataset, cell_specification, params )

% get cell numbers
cell_numbers = get_cell_numbers( cell_specification );
% if it's a cell array, reject
if class(cell_numbers) == 'cell'
    error('This function only accepts a single list of cells.');
end

...
```

This usage can be seamlessly integrated with any current functions that don't care about cell type. At the same time, "get_cell_numbers" serves other functions for which knowing cell type names are critical.

The proposed code for "get_cell_numbers" is in an appendix below.

## Closing

This proposal aims to make the Matlab code as simple, modular, inclusive, and expandable as possible. Internally, a single, parallel structure stores all data and analysis, making them accessible to every function. Externally, the user can operate on the data without worrying about its internal details. In the future, the proposed structures can be easily expanded to incorporate new needs.

In short, we believe this new proposal can serve SNL-E's Matlab needs indefinitely.

## Appendix

### code for "get_cell_numbers"

In a few places, a description of code yet to be written is summarized between "<" and ">".

```
function [cell_numbers, cell_type] = get_cell_numbers( dataset, cell_specification )

switch class(cell_specification)

    case 'double'  % a list of cell numbers
        cell_numbers = double(cell_specification);
        cell_type = [];

    case 'char'  % a cell type name
        [cell_numbers, cell_type] = get_cell_numbers_by_type( dataset, cell_specification);

    case 'cell'  % a list of names or cell type numbers
        % handle one element of the list at a time
        for cc = 1:length(cell_specification)
            [ cell_numbers{cc}, cell_type{cc} ] = ...
                get_cell_numbers_by_type( dataset, cell_specification{cc} );
        end

        % if only one cell type was specified, don't return a cell array
        if length(cell_specification) == 1
            cell_numbers = cell_numbers{1};
            cell_type = cell_type{1};
        end

    otherwise
        <give an appropriate error>
end

function [cell_numbers, cell_type] = get_cell_numbers_by_type( dataset, cell_specification );
% internal function
% return the cell numbers of the type specified in 'cell_specification'

switch type(cell_specification)
    case 'number'
        cell_type_number = cell_specification;
    case 'char'
        <loop through cell types looking for a type with the given name>
        cell_type_number = <result>;
    otherwise
        <give an appropriate error>
end

cell_numbers = dataset.cell_types{cell_type_number}.cell_numbers;
cell_type = dataset.cell_types{cell_type_number}.name;
```