CMBHOME: A Custom MATLAB Class for Neural Data

Andrew Bogaard[a]
The Center for Memory and Brain at Boston University

(Dated: 11 November 2010)

CONTENTS

I. INTRODUCTION

This documentation for the MATLAB Toolbox, CMBHOME, is meant to be a reference sheet for the standard data structure and its functionality, and does not include much information about analysis methods. At the end of this document you will find a cookbook series of commands meant to show you how to load data into a CMBHOME Session class object, perform basic data cleanup, and begin handing/analyzing your data.

The CMBHOME Toolbox lies within a MATLAB package directory called +CMBHOME:

- +CMBHOME/
  - @Session/
  - @Spike/
  - @LFP/

The directories prepended by the @ symbol are classes in the package.

The three classes were selected because they represent three distinct datatypes we deal with at the Center for Memory and Brain (CMB). Most of this documentation highlights how the Toolbox interfaces with in vivo single-unit recording experiments, (a Session class object with LFP and Spike objects within) but each class can be used independent of the other.

The way data is stored in a custom class object looks similar to the way data is stored in custom fields of a MATLAB struct:

```
root.x % vector of x positions in Session
% object root
mystruct.x % vector of x positions in custom
% field x in struct
```

But classes are more powerful because you can associate them with methods (functions) for handling and analyzing your data. For example, if you want to retrieve all x positions between t_start and t_stop from a struct, you might make a function to do this:

```
epoch_x = GetEpochData(struct.x, struct.ts, ...
t_start, t_stop);

epoch_y = GetEpochData(struct.y, struct.ts, ...
t_start, t_stop);
```

[a] Electronic mail: abogaard@bu.edu

Or you can make dependencies within a custom class object so that one property affects another:

```
root.epoch = [t_start, t_stop];
```

Now, root.x and root.y are equivalent to epoch_x and epoch_y above, because they both depend on the epoch property.

II. GETTING THE TOOLBOX

The CMBHOME package currently lives at
`http://conte.bu.edu/repos/abogaard/+CMBHOME`

You will also need
`http://conte.bu.edu/repos/abogaard/chronux`

if you plan to use the methods which interface with the Chronux package (see Table II).

Use svn to "checkout" the two folders above. Both directories must reside in the same parent directory. If you use Windows, tortoise is an svn client that offers a pretty seamless integration to the native file browser. Otherwise, use the command

```
svn checkout http://conte.bu.edu \
    /repos/abogaard/+CMBHOME
svn checkout http://conte.bu.edu \
    /repos/abogaard/chronux
```

to make a local copy of the package.

The toolbox is rapidly developing as of May 2010, so make sure and "svn update" the svn directories often.

Once you have a local copy of the toolbox, add the parent directory to your MATLAB path:

```
addpath(path, 'path/to/toolbox/parent/directory')
```

You could then save the path using
`savepath ,`

but as Ben Kraus points out,

> When you use savepath (or save the path using the GUI in MATLAB), it does one of two things (at least on Windows):
>
> 1) It can overwrite your system default pathdef.m. I don't like this for many reasons: you shouldn't be changing system files in case multiple people use the computer, your changes aren't copied to new versions of MATLAB when you upgrade, your changes are in your user directory so they aren't backed up, your changes are hard to find when you switch computers, etc. etc. etc.
>
> 2) It can create a new "pathdef.m" in your startup directory. This file is a copy of the *original* "pathdef.m" with your desired directories added. From this point on, the system default pathdef.m is masked by your new customized pathdef.m.
>
> Either options work fine as long as you never upgrade or reinstall MATLAB. If you are using option 1, then reinstalling erases your pathdef.m and upgrading creates a new pathdef.m and ignores the old.
>
> If you use option 2, then default pathdef.m changes but your personal copy of pathdef.m does *not* change to reflect the new and/or removed directories. Now your pathdef.m is out of sync with the system pathdef.m, and this can cause some problems.

So instead, he adds,

> What I recommend instead of using the "savepath" function, is to create a file called "startup.m". In that startup.m, make a call to "addpath" and use that mechanism to add your desired directories to the path. This will always start with the default pathdef.m, and update it to contain your personal directories

III. IMPORTING YOUR DATA

So far, there are functions for importing recording files from Neuralynx and Plexon rigs. When you import a recording session with the CMBHOME.Import class, a Session class object, named root, is created with:

- event flags (labels and timestamps) (root.event = string_label, timestamp)
- the date the object was created, (root.date_created = serial date number
- position and head tracking information (if exists), (root.b_x, root.b_y, root.b_headdir)

- single unit spike times, root.spike(tetrode_index, cell_index)
- paths to the raw data files from which is was created (root.path_raw_data)
- a link to the LFP files (if they are very large), or LFP signals in the (object. root.b_lfp)
- . . . other derivates of the data above

A property prepended with "b_" indicates a hidden *base field* that the user does not deal with directly. Instead, this is just where data is stored in the innerworkings of the object. For every "b_" property, there is an analog property without the "b_" prefix that returns data relevant to internal state properties (see Internal State Properties).

A. Neuralynx

You may import your Neuralynx recording files manually (with user prompts), or in *batch mode*. To import your files manually, type

```
CMBHOME.Import.NL
```

which has optional arguments

- **base_path** The directory that contains your recording files. If **batch** is specified, the Import function recursively searches all directories in **base_path** for recording files. (optional if batch is not set to 1, default=current directory)
- **save_path** The directory where you would like to save your session object .mat file. (optional, default=current directory, or **base_path**, if set)
- **fix_pos** Indicates whether position tracking data should be corrected and smoothed. See Session.FixPos for details about the smoothing. (optional, default=0)
- **fix_headdir** Indicates whether head direction tracking data should be corrected and smoothed. See Session.FixDir for details about the smoothing. (optional, default=0)
- **batch** Indicates whether you want the Import function to iterate through all directories one level deeper than **base_path** for recording files (see below for requirements). Requires that **base_path** is set.
- **n_tetrodes** Number of tetrodes to look for. (optional, default=12)

Examples:

```
% prompts user to select files in Session5 folder
CMBHOME.Import.NL('base_path', 'C:\data\Session5')
% searches through all directories in 'data' for
% recording files which follow naming convection
CMBHOME.Import.NL('base_path', 'C:\data', 'batch', 1)
```

BATCH MODE

The batch mode functionality is flexible, despite the fact that Neuralynx sessions spit out multiple files. I list the rules the batch mode follows below.

Batch mode automatically saves each Session object in the folder that contains the session recording files by naming convention CMBH_ <foldername >.mat. For each directory one level higher than **base_path**, Import.NL looks for:

1. An events file named "Events.nev" (the default Neuralynx convention)
2. A video file (.nvt) with naming scheme *OrdCor.nvt", where * denotes a wildcard. If such a file does not exist, it selects the .nvt file with the shortest file name (since often I noticed superfluous nvt files).
3. Cluster files with isolated units in .mat files with an Nx2 array of CellNumber x SpikeTime. The .mat file names must have the tetrode number somewhere in the filename ex. Sc12.mat and Sc2.mat are cluster files for tetrodes 12 and 2. The cluster files must either be in a folder named ClusterCut/, or follow naming convention Sc*.mat, where * is a wildcard. Import.NL interprets the filename and automatically indexes the tetrodes.

B. Plexon

You may import your Plexon recording files manually (with user prompts), or in *batch mode*. To import your files manually, type

```
CMBHOME.Import.PLX % or
root = CMBHOME.Import.PLX; % without the return
% variable, nothing is returned to workspace,
% but it is saved to folder with .plx data by default
```

which has optional arguments

- **base_path** The directory that contains your recording file(s). If **batch** is specified, the Import function recursively searches all directories in **base_path** for recording files. (optional if batch is not set to 1, default=current directory)
- **save_file** Indicates whether Import.PLX will save each Session object to the disk (default=1). If this is set to 0, you need to call
  ```
  root = CMBHOME.Import.PLX;
  ```
  so that the Session objects are at least returned to the workspace. If you are importing many .plx files at once, the function will return a cell array of Session objects (which could get quite large, if you aren't careful). Note that each root Session object in the cell array has a property ".name" with a reasonable name to save it by, as well as the property ".path_raw_data" which informs the user where each Session object originated. Returning the objects to the workspace, instead of writing to disk, is useful if you want to do extra post processing after each import.
- **fix_pos** Indicates whether position tracking data should be corrected and smoothed. See Session.FixPos for details about the smoothing. (optional, default=0)
- **fix_headdir** Indicates whether head direction tracking data should be corrected and smoothed. See Session.FixDir for details about the smoothing. (optional, default=0)
- **batch** Indicates whether you want the Import function to iterate through **base_path** and all folders one level higher for .plx recording files. Requires that **base_path** is set.

BATCH MODE

The batch mode functionality is straightforward for Plexon users. For **base_path** and every directory one level higher, Import.PLX looks for *.plx files, and makes a .mat file with the same filename prefix in the same folder.

C. NSpike

Under development

D. Axona

A function exists to import from axona .txt files. Meet with Ehren to learn how to export your data like this.

E. Anything else

It is easy to import any plaintext data, or matlab variables into the toolbox. The methodology is as follows: build the Session object, an 2-d array of Spike objects ordered by [tetrode index, cell index], and add the Spike object to the Session"spike" property.

The syntax is as follows. Lets presume we have position data x_pos, and y_pos, head direction data head_angle, and timestamps ts. To create a Session object, type:

```
import CMBHOME.*
root = Session('name', 'MySessionObj', ...
                         'b_ts', ts, ...
                         'b_x', x_pos, ...
                         'b_y', y_pos, ...
                         'b_headdir', hd_angle,);
```

Build the spike object array. For example, cell (tetrode 4, cell 1) with spike times spike_ts might look like:

```
import CMBHOME.*
spike(4, 1) = Spike('ts', spike_ts, ...
                    'vid_ts', ts);
```

Notice that we pass the video timestamps to the Spike function. This aligns the cells spike times to the timestamps for quicker analysis down the road. Type help CMBHOME.Spike.Spike to learn about other Spike object properties.

F. Loading a Session Object into MATLAB

Once you have a .mat with your object inside, you can simply drag the file into the workspace to work with that session. Or, if you want to write a custom script to load relevant session objects, you can dynamically load any of the objects using

```
load('/path/to/object.mat');
```

It is useful to keep a naming convention for all of your session objects. By default, it is "root". This way, whenever you dynamically load an object, your functions know how to address the data. If you want to load multiple objects into the workspace at once, you can load them into a struct array:

```
a(1) = load('/path/to/object.mat');
a(2) = load('/path/to/object2.mat');
```

Now, a(1).root and a(2).root both exist in the workspace. This is better than using "eval()" to rename the object itself. You can also merge and split objects (see Merging and Splitting Session Objects)

IV. USING THE TOOLBOX

A. Access Your Data

Accessing your data in a Session object is similar to accessing you data in a struct. Instead of your own custom fields, though, there are standardized "properties".

```
root.headdir % standard property headdir
%  in root Session object
mystruct.hd % custom hd field in a struct
```

In the example above, both hd and headdir refer to the angle the animals head is facing. The Session class defines what the head direction angle vector is called, and also that its range is [-180, 180). This standardization is convenient for many reasons (http://en.wikipedia.org/wiki/Standardization).

See the Properties Reference Sheet for naming conventions, etc.

1. Internal State Properties

The following properties, when set, determine how the data from the session is returned to the user.

- **root.epoch** An Nx2 array of timestamps like [t_start, t_stop; ...]. If N==1, then all tracking data, spike train data, etc, returned will be vectors of data which lie within root.epoch(1) and root.epoch(2) inclusive. If N>1, then all data returned will be cell arrays of length N.

- **root.cell_thresh** A two element vector where root.cell_thresh(1) indicates a threshold of firing frequency in Hz. The optional root.cell_thresh(2) is either 0 or 1 and indicates whether the threshold applies to the average firing frequency over the entire session, or applies to the time course of firing frequency throughout the run. This is useful in detecting tetrode drift. This property affects the dynamic root.cells array: only cells which exceed root.cell_thresh are returned.

- **root.spatial_scale** Sets the ratio cm/pixel used often in rate map analysis and plotting functions

- **root.fs** Sampling frequency of the LFP signal

- **root.fs_video** Sampling rate of the tracking data

2. Merging and Splitting Session Objects

Sometime soon, there will exist methods for merging two or more Session class objects together or splitting one apart. This could be useful for analyzing like-trials together.

B. Analyze Your Data

1. Built-in Methods

2. Custom Methods

VI. PROPERTY AND METHOD REFERENCE SHEET

To consolidate help documentation, this table has been removed. Help documentation is *rather thorough in the MATLAB code, and is most easily browsed using the commands:

```
>> doc CMBHOME.Session
```

```
>> doc CMBHOME.LFP
```

```
>> doc CMBHOME.Spike
```

V. HANDS-ON TUTORIAL

Once you have the most recent version of the toolbox copied to your computer, load the supplied Session object (http://conte.bu.edu/repos/abogaard/+CMBHOME/CMBH_sample.mat) to your MATLAB workspace and work through the tutorial.

In this tutorial, we will review the similarities between Session object properties and struct fields. We will also review the relationship between Session objects and methods, which are really just functions packaged together with the class.

We will explore video recording data, and spiking data. We will visually inspect all cells that fire more than 1Hz over the course of the recording session.

We will recognize a head direction cell by its firing rate map and directional rate map, and test for how strongly tuned for head direction it really is with Watson's U2 test.

Also, we will explore the LFP functionality, and perform basic analysis including filtering and solving for phase of theta oscillations and how it pertains to cell firing.

After updating the Session object with the analysis above, we will save it to the local computer.

Let's start. If your MATLAB path is not aware of the CMBHOME directory, add it to the path.

```
addpath(path, '/path/to/repository') % parent directory to +CMBHOME/
```

The most helpful tool throughout your use of the CMBHOME package will be auto-generated help documentation,

```
doc CMBHOME.Session

doc CMBHOME.Spike

doc CMBHOME.LFP

help CMBHOME.Session.functionname

help CMBHOME.Session.propertyname
```

Don't forget to use those commands!
Now, load the object into the workspace. Once you see "root" in the workspace we're ready to begin. The first bit of code will explore properties that store data, much like fields in structs.

```
x = root.x; % what is x? a vector... x positions

ts = root.ts; % timestamps vector (for video samples)

root % print to screen all properties in the object
```

So, properties are named logically. If you ever wonder what a property is, type "help CMBHOME.Session.<property name>". What about cells? These Session objects contain isolated units with spikes in a property, root.Spike. Mostly, all methods used to interact with these cells lives in the root object. See below:

```
root.cells % print to screen all of the cells that exist in Session object

size(root.cells) % how many cells are cut? root.cells is an N x 2 array of [tetrode num, cell num]
```

So, there are 30 cells in this session.

An interesting aspect of the cells property is that it relies on root.cell_thresh, a property that sets a threshold for minimum firing activity

```
help CMBHOME.Session.cell_thresh % what is cell_thresh?

root.cell_thresh = 1;  % sets cell threshold to 1 Hz across entire session

size(root.cells, 1) % root.cells is dynamic: 11 cells with low firing rates are not included now.
```

This concept, the dynamic property, is very important to understand. It makes the custom class much more powerful than a struct, because we introduce the "epoch" property. The epoch property defines (a) period(s) of time for which the object returns data. The next example will make this clear.

Now, lets plot the animal's trajectory on the circle maze (from time=15300 seconds to 15800 seconds), with spikes overlaid as dots.

```
root.epoch = [15300, 15800]; % sets active epoch to seconds 15300 through 15800s

figure; % make a new figure

plot(root.x, root.y, 'Color', 'k') % plots trajectory

hold on % holds plot for more

scatter(root.spk_x([6, 1]), root.spk_y([6, 1]), '.r'); % scatters cells 6,1's spikes
```

Do you notice in the plot that there were jumps back to the origin? This happens often in video tracking: samples that are occluded by the cable, or reflections, can distract the tracking system and cause for jumps that are pretty clearly impossible without teleportation. Usually, they are easy to detect and remove. There are two functions, or METHODS, in the CMBHOME package meant to fix tracking skips in the position and directional data. They are FixDir and FixPos.

Check out how a method is invoked:

```
root = root.FixPos;
```

These functions are called methods because they can be used with the following syntax: Class.Method. How would you add your own FixPos script to the package?

```
function root = MyFixPos(root)
% this function is a method to the CMBHOME Session class object. It resides in the directory
% +CMBHOME/Session/MyFixDir.m.

pseudo-code:
for samples i and i+1
    if distance between last root.b_x and root.b_y samples and the current is > big
        root.b_x(i) = better guess
        root.b_y(i) = better guess
    end
end

end % end function
```

Thus, we have returned the root object with some of its basic properties changed. It MUST reside in the class directory for which it is a method. Now, back to playing with epochs.
First, lets see if FixPos worked.

```
plot(root.x, root.y) % plots trajectory
```

Nice! OK, now lets plot the animals trajectory during the entire recording session. Just change the epoch like below, and run the same plotting functions.

```
root.epoch =  [root.b_ts(1), root.b_ts(end)]; % sets active epoch to entire session

plot(root.x, root.y) % plots trajectory
```

Note that the command, “plot(root.x, root.y)”, did not change, yet the output (the figure) did. Why? root.ts, root.x, root.y, root.headdir, root.spk_x... (see tables below) are all dynamic, and depend on root.epoch. Thus, root.x is always (always!) x data from the session that happened in the epoch defined in root.epoch. So within a function, from the command line, from a GUI, you can always grab data from within a window of time with a one liner. root.epoch is an Nx2 array of timestamps like [start, stop; start, stop]... If one epoch is set, each of these properties return a vector. If multiple epochs are set, each of these properties returns a cell array of vectors representing data from each epoch. The entire session data vectors are stored in “base” properties, like root.b_x.

Onward. Let’s visually inspect the firing of cells, and how it pertains to video data. There is a built-in GUI for this called Visualize2.

```
root.Visualize2; % run Visualize2
```

Check “Trajectory” and “Polar Rate Map” and browse through cells. Note that cell 6,1, among many others, has clear directional tuning. Also note that the Watson’s U2 score is greater than 5: this is a head direction cell.

In this experiment, the animal started on a pedestal, and then was transported to a circular track. Lets see if the head direction cell firing was similar in both settings.

In Visualize2, click Change Epochs. Select ’pedestal start’ and ’pedestal stop’ from the drop-downs. Click done. Note that the Trajectory is limited to a small space now (the pedestal). Make note of the Watson’s score on the polar rate map.

Click Change Epochs again, and this time select Circle Start and Circle Stop from the drop-down. Click done. Now, the trajectory is on the circle track, and the Watson’s score reflects this epoch.

This is useful, but a slow way to do analysis. Lets try the command prompt.

Close all Visualize2 figures.
Here are two possible ways to use root.event to assign times in root.epoch:

```
% select Pedestal Start and Pedestal Stop, add another epoch, select Circle Start and Circle Stop
root = root.SetEpoch;

% a native method which returns timestamps for event strings that exist in root.event in the
% shape of the cell array handed to it
root.epoch = [root.Label2Time({'pedestal start',  'pedestal stop'; 'circle start', 'circle stop'; });
```

Now, since we have set two epochs, we can quickly calculate the WatsonsU2 statistic for both epochs.

```
w_scores = root.HDWatsonsU2([6, 1]); % returns U2 scores for both epochs on and off the track.
```

For this project, it was necessary to plot the spiking of cells during each lap. It was necessary to write a script to detect the onset of laps. This code does not belong to the Toolbox, so how do we add it? Find laps.m in the tutorial zip file, and move it into the @Session folder (whose parent directory is +CMBHOME). Doing this adds the function, laps, as a “method” to the Session class. Thus, the syntax root.MethodName can be used. When writing a function, the header must include the class object as it’s first argument (ex. function root = laps(root)). See the function laps below

```
function laps = laps(self)
% root = root.laps;
%
% This function parses data recorded on mark's circle track into laps.  It
% accepts the CMBHOME Session class, and finds times corresponding to
% circle start and circle stop
%
% cma first ed, 2008
% andrew january 6 2010
% andrew june 1 2010 updated for CMBHOME

self.epoch = [self.Label2Time('circle start'), self.Label2Time('circle stop')];

... (code omitted)

laps;

end
```

Since the class object was loaded into the workspace before this method existed in the class folder, this instance of the root object cannot see root.laps. Unfortunately, you must clear your workspace, and add the object again. This may require use of “clear classes”. You do not have to do this if you make changes to functions in the class folder. But first, lets save our work! Make sure you do not clear the LFP from our object!

```
% asks user whether to save LFP, and if directory doesnt exist or overwrite impending prompts again
root.Save
```

You can also use the Save method from within functions you without prompting the user:

```
help CMBHOME.Session.Save % how to use save

root.Save([0 0]); % prompts if path does not exist, saves LFP
```

Now, clear the workspace: ”close all; clear all;”. Load your saved object back into the workspace.
Once the root object is back in the workspace we will use our new function, and plot a raster of our cell’s spikes aligned to the onset of each lap.

```
root.user_def.laps = root.laps;

root.epoch = root.user_def.laps; % set epochs to be distinct laps

% built in function for plotting raster in time
root.plot_raster([6 1]), ylabel('Lap'), xlabel('Time from Onset (sec)')
```

It turns out this isn’t the best way to visualize ‘in-field’ spiking, since the laps vary greatly in duration. Thus, the animal is not always facing the same direction at the same point in time. It makes more sense to plot spikes during each lap against head direction, or a linearized version of the track. How would you do it?

```
close all; % close all figures

import CMBHOME.Utils.* % add access to functions in Utils folder of Toolbox

PipeRaster(root.spk_headdir([6, 1])); % vectorized function for plotting a raster

ylabel('Lap');
xlabel('Head Direction');
```

PipeRaster accepts a cell array of vectors, and plots the value of each vector sample along the x-axis, and at the index it resides in the cell array on the y axis. So here, root.spk_headdir([6 1]) is a cell array of vectors of head direction samples at the spike times of cell 6, 1. Each cell array index aligns to an epoch in root.epoch.
Pretty simple, considering that we are grabbing the closest head direction sample to every spike that occurs within every one of twenty three laps in one command (root.spk_headdir). Also try spk_x, spk_y, spk_ts, spk_vel, spk_theta.

Now, lets look at how LFP data is handled. root.b_lfp is an array of LFP objects.

```
close all % closes all figures

root.b_lfp(4) % print to screen properties index 4 loaded lfp.
```

The properties you see listed now belong to the LFP class (doc CMBHOME.LFP). We’re looking at the LFP class object from data loaded from file root.path_lfp4.
The LFP class has methods (functions) built in for filtering a continuous signal, solving for instantaneous phase, detecting ripple events, REM epochs, and more. Some of these are implemented as dynamic properties, meaning they will be solved for on the fly if they are not saved in the object.

If you will be accessing this property often, it may make sense to save it to the object. Try this:

```
% how long does it take to get the filtered signal on the fly?

tic; root.b_lfp(4).theta; toc

root.b_lfp(4) = root.b_lfp(4).AppendTheta; % adds the theta-filtered signal to the object

root.b_lfp(4) = root.b_lfp(4).AppendThetaPhase; % adds phase

tic; root.b_lfp(4).theta; toc % how long does it take now?
```

What just happened? The CMBHOME.LFP.theta object returns the theta filtered signal for CMBHOME.LFP.signal. It first looks in CMBHOME.LFP.b_theta, but if it hasnt been solved for already, it runs the method CMBHOME.LFP.ThetaFilter on CMBHOME.LFP.signal.

CMBHOME.LFP.AppendTheta adds the theta filtered signal to the b_lfp property, so it doesn't have to be solved by the Buttersworth filter method every time. That way, if you're looping you can call the theta property without filtering the raw signal every time.

What if you want to make your own filtered signal? We could add it to the package, or you can write a function (method) that will add it to CMBHOME.LFP.user_def.my_vector. Then, future methods you write can just access CMBHOME.user_def.my_vector.

The root.active_lfp property defines the LFP data source for the dynamic root.lfp property, which is dependent on root.epoch.

```
root.active_lfp = 4; % set our active lfp

root.lfp % look familiar? this is the snapshot(s) of root.b_lfp(4) for each lap in root.epoch

root.epoch = root.epoch(1,:); % set one epoch to first lap

root.lfp % now its a snapshot of root.b_lfp(4) during the first lap
```

root.lfp is a CMBHOME.LFP object just like root.b_lfp, but it is dynamically generated and depends on root.active_lfp and root.epoch. Thus, root.lfp is an LFP object with data from root.b_lfp(root.active_lfp) for epochs root.epoch. If multiple epochs are selected (remember, root.epoch is an Nx2 array of timestamps), then the properties of root.lfp contain cell arrays, not vectors.

```
figure;

subplot(4, 1, 1), plot(root.lfp.ts, root.lfp.signal); % plots LFP signal in that epoch
subplot(4, 1, 2), plot(root.lfp.ts, root.lfp.theta); % plots filtered LFP signal in that epoch
subplot(4, 1, 3), plot(root.lfp.ts, root.lfp.theta_phase); % plots instantaneous theta phase
subplot(4, 1, 4), root.plot_raster([6 1]); % plots cell 6,1's spikes during that epoch
```

Cool! OK, close figures.

Now, lets visualize LFP spectrum data. This sample data file includes two loaded LFP channels, but there are 14 other LFP data files specified in root.path_lfp. Notice that the path for those data files are on another computer - you won't be able to load them here. Whenever you want to share data between computers (and you need the LFP data), it is probably easiest to import the LFP data into the Session object (see help CMBHOME.Session.LoadLFP), save the Session object, and then share it between computers. On the other hand, if the raw data files exist on your local machine, you can update root.path_lfp with the local files (see help CMBHOME.Session.AddLFP) to reduce Session object .mat file size.

```
root.epoch = [15300 15800]; % set epoch back to whole session

root.VisualizeLFP; % load VisualizeLFP control panel and figure window
```

While in VisualizeLFP, check Chronux Spectrogram, and CSC4 from the drop-down. Click update, and this will plot LFP power spectrum in time during the task. Notice the prominent theta rhythm that coincides with running around the track! Very cool! Also, play with the time-bandwidth parameters to see how it affects your plot.

So far, only the chronux toolbox functions are added to VisualizeLFP, but I plan to add custom MATLAB-based scripts for generating the same plots for comparison

Close all your figure windows.

More to come sometime soon! Like how to port CMBHOME function for use on the side, how to write your own vectors that align with the video timestamps, or LFP timestamps, and much MUCH MORREEE!