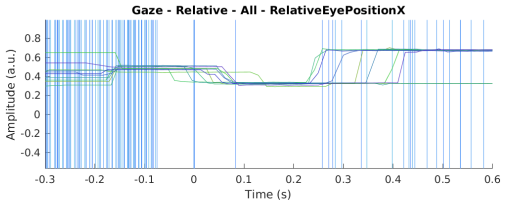
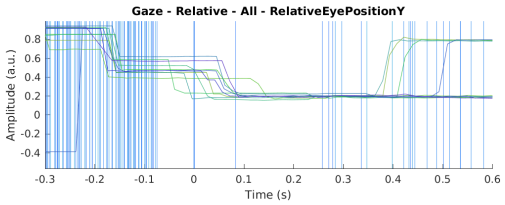
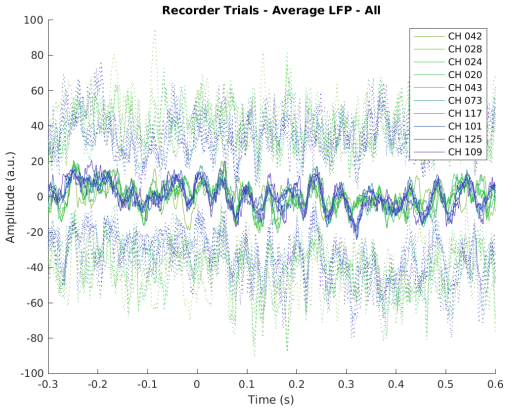
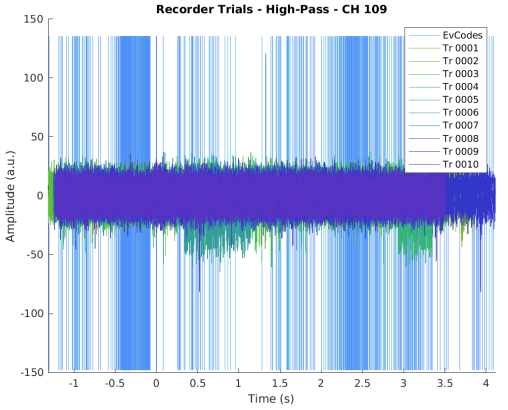
Chris's Experiment Utility Libraries – Function Reference

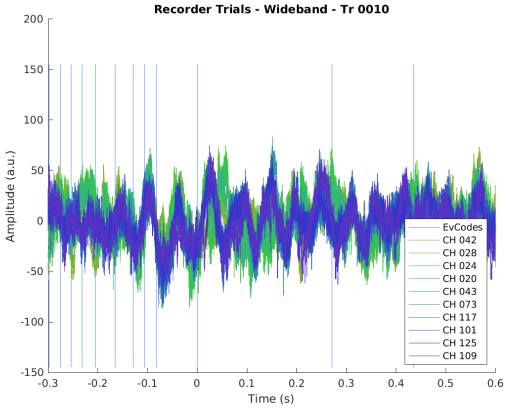
Written by Christopher Thomas – September 14, 2022.











Overview

This is a set of libraries and utilities written to support ephys experiment analyses in Thilo's lab.

This is intended to be a private project for lab-specific code that is not specific to individual experiments. Code that lends itself to reuse outside our lab can be migrated to public projects. Code that's experiment-specific should be in projects associated with those experiments.

Libraries are provided in the "`libraries`" directory. With that directory on path, call the "`addPathsExpUtilsCjt`" function to add sub-folders.

The following sets of library functions are provided:

- **"`euAlign`"** functions (Chapter 2) perform time-alignment of event lists from different sources.
- **"`euFT`"** functions (Chapter 3) provide wrappers for Field Trip functions and combine commonly-used sets of Field Trip function calls.
- **"`euPlot`"** functions (Chapter 4) plot experiment data for testing. These don't make paper-quality plots.
- **"`euTools`"** functions (Chapter 5) are helper functions used by specific tools and scripts.
- **"`euUSE`"** functions (Chapter 7) are functions for reading and interpreting USE data (event codes, SynchBox activity, gaze data, and frame data).
- **"`euUtil`"** functions (Chapter 8) are utility functions that don't fit into other categories.

The following sample code is provided:

- **"`ft_demo.m`"** – Simplest practical script for reading our lab's datasets into Field Trip. This is described in Chapter 1.

Contents

Part I

Examples

Chapter 1

"do_demo.m" Example Script

1.1 README.md

```
# Chris's Field Trip Example Script

## Overview

This is a minimum working script showing how to read data from one of our
lab's experiments into Field Trip and perform processing with it.

About half of this is done using Field Trip's functions directly, and the
remainder is done with wrapper functions that combine many common Field Trip
operations. If you're not sure what a step is doing or how it's doing it,
looking at that function's documentation and function body will help.

To run this script, you'll need Field Trip and several libraries installed;
details are below.

## Getting Field Trip

Field Trip is a set of libraries that reads ephys data and performs signal
processing and various statistical analyses on it. It's a framework that
you can use to build your own analysis scripts.

To get Field Trip:
* Check that you have the Matlab toolboxes you'll need:
    * Signal Processing Toolbox (mandatory)
    * Statistics Toolbox (mandatory)
    * Optimization Toolbox (optional; needed for fitting dipoles)
    * Image Processing Toolbox (optional; needed for MRI)
* Go to [fieldtriptoolbox.org](https://www.fieldtriptoolbox.org).
* Click "latest release" in the sidebar on the top right
(or click [here](https://www.fieldtriptoolbox.org/#latest-release)).
* Look for "FieldTrip version (link) has been released". Follow that
```

```
GitHub link (example:
[Nov. 2021 link](http://github.com/fieldtrip/fieldtrip/releases/tag/20211118)).
* Unpack the archive somewhere appropriate, and add that directory to
Matlab's search path.

Bookmark the following reference pages:
* [Tutorial list.](https://www.fieldtriptoolbox.org/tutorial)
* [Function reference.](https://www.fieldtriptoolbox.org/reference)

More documentation can be found at the
[documentation link](https://www.fieldtriptoolbox.org/documentation).

## Other Libraries Needed

You're also going to need the following libraries. Download the relevant
GitHub projects and make sure the appropriate folders from them are on path:

* [Open Ephys analysis tools](https://github.com/open-ephys/analysis-tools)
(Needed for reading Open Ephys files; the root folder needs to be on path.)
* [NumPy Matlab](https://github.com/kwikteam/npy-matlab)
(Needed for reading Open Ephys files; the "npy-matlab" subfolder needs to be
on path.)
* My [LoopUtil libraries](https://github.com/att-circ-contrl/LoopUtil)
(Needed for reading Intan files and for integrating with Field Trip; the
"libraries" subfolder needs to be on path.)
* My [experiment utility libraries](https://github.com/att-circ-contrl/exp-utils-cjt)
(Needed for processing steps that are specific to our lab, and more Field
Trip integration; the "libraries" subfolder needs to be on path.)

## Using Field Trip

A Field Trip script needs to do the following:
* Read the TTL data from ephys machines and SynchBox data from USE.
* Assemble event code information, reward triggers, and stimulation triggers
from TTL and SynchBox data.
* Time-align signals from different machines (recorder, stimulator, SynchBox,
and USE) to produce a unified dataset.
* Segment the data into trials using event code information.
* Read the analog data trial by trial (to keep memory footprint reasonable).
* Perform re-referencing and artifact rejection.
* Filter the wideband data to get clean LFP-band and high-pass signals.
* Extract spike events and waveforms from the high-pass signal.
* Extract average spike activity (multi-unit activity) from a band-pass signal.
* Stack trigger-aligned trials on to each other and get the average response
and variance of time-aligned trials (a "timelock analysis").
* Perform experiment-specific analysis.

### Reading Data with Field Trip

* `ft_read_header` reads a dataset's configuration information.
```

* ‘ft_read_event‘ reads a dataset’s event lists (TTL data is often stored as events).
* ‘ft_preprocessing‘ is a "do-everything" function. It can either read data without processing it, process data that’s already read, or read data and then process it. At minimum you’ll use it to read data.

For all of these, you can pass it a custom reading function to read data types it doesn’t know about it. We need to do this for all of our data (Intan, Open Ephys, and USE). You can tell it to use appropriate NeuroLoop functions to read these types of data, per the sample code. It can also be told to read events from tables in memory using NeuroLoop functions.

NOTE - When reading data, you pass a trial definition table as part of the configuration structure. Normally this is built using ‘ft_definetrial‘, but because of the way our event codes are set up and because we have to do time alignment between multiple devices, we build this table manually.

We’re using ‘euFT_defineTrialsUsingCodes()‘ for this.

Signal Processing with Field Trip

* Field Trip signal processing calls take the form "‘newdata = ft_XXX(config, olddata)‘. These can be made through calls to ‘ft_preprocessing‘ or by calling the relevant ‘ft_XXX‘ functions directly (these are in the ‘preproc‘ folder). The configuration structure only has to contain the arguments that the particular operation you’re performing cares about.
* ONLY call functions that begin with ‘ft_‘. In particular, anything in the "‘private‘" directory should not be called (its implementation and arguments will change as FT gets updated). The ‘ft_XXX‘ functions are guaranteed to have a stable interface.
* *Almost* all signal processing operations can be performed through ‘ft_preprocessing‘. The exception is resampling: Call ‘ft_resampledata‘ to do that.
* See the preamble of ‘ft_preprocessing‘ for a list of available signal processing operations and the parameters that need to get set to perform them.

Field Trip Data Structures

Data structures I kept having to look up were the following:

* ‘ft_datatype_raw‘ is returned by ‘ft_preprocessing‘ and other signal processing functions. Relevant fields are:
    * ‘label‘ is a {Nchans x 1} cell array of channel names.
    * ‘time‘ is a {1 x Ntrials} cell array of [1 x Nsamples] time axis vectors. Taking the lengths of these will give you the number of samples in each trial without having to load the trial data itself.
    * ‘trial‘ is a {1 x Ntrials} cell array of [Nchans x Nsamples] waveform matrices. This is the raw waveform data.

    * ‘fsample‘ is "deprecated", but it’s still the most reliable way to get the sampling rate for a data structure. Reading it from the header (which is also appended in the data) gives you the wrong answer if you’ve resampled the data (and you often will downsample it).
    * Trial metadata is also included in ‘ft_datatype_raw‘, but it’s much simpler to keep track of that separately if you’re the one who defined the trials in the first place.

* A **header** is returned by ‘ft_read_header‘, and is also included in data as the ‘ft_datatype_raw.hdr‘ field. Relevant fields are:
    * ‘Fs‘ is the *original* sampling rate, before any signal processing.
    * ‘nChans‘ is the number of channels.
    * ‘label‘ is a {Nchans x 1} cell array of channel names.
    * ‘chantype‘ is a {Nchans x 1} cell array of channel type labels. These are arbitrary, but can be useful if you know the conventions used by the hardware-specific driver function that produced them. See the LoopUtil documentation for the types used by the LoopUtil library.
    * ‘nTrials‘ is the number of trials in the raw data. This should always be 1 for continuous recordings like we’re using.

* A **trial definition matrix** is a [Ntrials x 3] matrix defining a set of trials.
    * This is passed as ‘config.trl‘ when calling ‘ft_preprocessing‘ to read data from disk.
    * Columns are ‘first sample‘, ‘last sample‘, and ‘trigger offset‘. The trigger offset is ‘(first sample - trigger sample)‘; a positive value means the trial started after the trigger, and a negative value means the trial started before the trigger.
    * Additional columns from custom trial definition functions get stored in ‘config.trialinfo‘, which is a [Ntrials x (extra columns)] matrix.
    * **NOTE** - According to the documentation, trial definitions (‘trl‘ and ‘trialinfo‘) can be Matlab tables instead of matrices, which allows column names and non-numeric data to be stored. I haven’t tested this, and I suspect that it may misbehave in some situations. Since we’re defining the trials ourselves instead of with ‘ft_definetrial‘, I just store trial metadata in a separate Matlab variable.

What This Script Does

This script performs most of the steps we’ll want to perform when pre-processing data from real experiments:

* It finds the recorder, stimulator, and USE folders and reads metadata from the recorder and stimulator.
* It reads digital/TTL events from the recorder, stimulator, and USE folders. The USE data includes what USE sent to the SynchBox (only USE timestamps) and what the SynchBox sent back (also has SynchBox timestamps).
* It reads gaze and frame data tables from USE.
* It aligns timestamps from all of these sources and translates everything into the recorder’s timeframe.

* It makes sure there's a consistent list with all of the events, since the recorder and stimulator are sometimes not set up to save all TTL inputs.
* It processes the list of event codes to find out when trials should happen. This is done using `euFT_defineTrialsUsingCodes()`, since we need metadata that `ft_definetrial()` doesn't usually look at, and since we're using `TrlStart` and `TrlEnd` to define the trial boundaries instead of fixed padding times.
* It reads the ephys data for all trials, performing filtering:
    * Power line noise and other narrow-band noise is filtered out of the wideband data.
    * The wideband data is low-pass filtered and downsampled to get LFP data.
    * The wideband data is high-pass filtered to get spike waveform data.
    * The wideband data is band-pass filtered, rectified, low-pass filtered, and downsampled to get multi-unit activity data.
* It segments USE event and gaze data per trial as well.
* It computes timelock averages of LFP and MUA data.
* It generates example plots for all of these.

Things that are missing:

* We're not identifying trials with artifacts. This is typically done manually, though automated tools are provided in the LoopUtil library.

* We're not re-referencing the data. Field Trip provides preprocessing options for this, but we'll probably have to do it probe by probe rather than across the whole dataset (common-average referencing the recording channels for each probe).

Miscellaneous Notes

* Right now, data from each device is aligned but stored separately. Eventually we'll want to use `ft_appenddata` to merge data from multiple devices (such as multiple Intan recording controllers, if we're using more than 8 headstages). This can only be done for data that's at a single sampling rate, which may not be practical for wideband and raw spike data.

* Right now, the script doesn't touch spike sorting or try to isolate single-unit spiking activity. The spike sorting pipeline presently needs to work on the entire monolithic dataset, rather than on trials, and that monolithic dataset won't fit in memory. What we're going to have to do is load monolithic data per-probe (64 or 128 channels) and re-save that for the spike sorting pipeline to process (after notch filtering and artifact removal).

This is the end of the file.

1.2 do_demo.m

```
% Short Field Trip example script.
% Written by Christopher Thomas.


%
% Configuration constants.

% Change these to specify what you want to do.


% Folders.

plotdir = 'plots';
outdatadir = 'output';

inputfolder = 'datasets/20220504-frey-silicon';


% Channels we care about.

% These are the channels that Louie flagged as particularly interesting.
% Louie says that channel 106 in particular was task-modulated.
desired_recchannels = ...
  { 'CH_020', 'CH_021', 'CH_022',   'CH_024', 'CH_026', 'CH_027', ...
    'CH_028', 'CH_030', 'CH_035',   'CH_042', 'CH_060', 'CH_019', ...
    'CH_043', ...
    'CH_071', 'CH_072', 'CH_073',   'CH_075', 'CH_100', 'CH_101', ...
    'CH_106', 'CH_107', 'CH_117',   'CH_116', 'CH_120', 'CH_125', ...
    'CH_067', 'CH_123', 'CH_109',   'CH_122' };

desired_stimchannels = {};


% Where to look for event codes and TTL signals in the ephys data.

% These structures describe which TTL bit-lines in the recorder and
% stimulator encode which event signals for this dataset.

recbitsignals = struct();
stimbitsignals = struct('rwdB', 'Din_002');

% These structures describe which TTL bit-lines or word data channels
% encode event codes for the recorder and stimulator.
% Note that Open Ephys word data starts with bit 0 and Intan bit lines
% start with bit 1. So Open Ephys code words are shifted by 8 bits and
% Intan code words are shifted by 9 bits to get the same data.
```

```
reccodesignals = struct( ...
  'signameraw', 'rawcodes', 'signamecooked', 'cookedcodes', ...
  'channame', 'DigWordsA_000', 'bitshift', 8 );
stimcodesignals = struct( ...
  'signameraw', 'rawcodes', 'signamecooked', 'cookedcodes', ...
  'channame', 'Din_*', 'bitshift', 9 );


% How to define trials.

% This is the code we want to be at time zero.
% NOTE - We're adding "RwdA" and "RwdB" as codes "TTLRwdA" and "TTLRwdB".
trial_align_evcode = 'StimOn';
%trial_align_evcode = 'TTLRwdA';

% These are codes that carry extra metadata that we want to save; they'll
% show up in "trialinfo" after processing (and in "trl" before that).
trial_metadata_events = ...
  struct( 'trialnum', 'TrialNumber', 'trialindex', 'TrialIndex' );

% This is how much padding we want before 'TrlStart' and after 'TrlEnd'.
padtime = 1.0;


% Narrow-band frequencies to filter out.

% We have power line peaks at 60 Hz and its harmonics, and also often
% have a peak at around 600 Hz and its harmonics.

notch_filter_freqs = [ 60, 120, 180 ];
notch_filter_bandwidth = 2.0;


% Frequency cutoffs for getting the LFP, spike, and rectified signals.

% The LFP signal is low-pass filtered and downsampled. Typical features are
% in the range of 2 Hz to 200 Hz.

lfp_maxfreq = 300;
lfp_samprate = 2000;

% The spike signal is high-pass filtered. Typical features have a time scale
% of 1 ms or less, but there's often a broad tail lasting several ms.

spike_minfreq = 100;

% The rectified signal is a measure of spiking activity. The signal is
% band-pass filtered, then rectified (absolute value), then low-pass filtered
% at a frequency well below the lower corner, then downsampled.
```

```
rect_bandfreqs = [ 1000 3000 ];
rect_lowpassfreq = 500;
rect_samprate = lfp_samprate;


% Nominal frequency for reading gaze data.

% As long as this is higher than the device's sampling rate (300-600 Hz),
% it doesn't really matter what it is.
% The gaze data itself is non-uniformly sampled.

gaze_samprate = lfp_samprate;


% Plotting configuration.

% Number of standard deviations to draw as the confidence interval.
confsigma = 2;


% Debug switches for testing.

debug_skip_gaze_and_frame = true;

debug_use_fewer_chans = true;

debug_use_fewer_trials = true;
%debug_trials_to_use = 30;
debug_trials_to_use = 10;

if debug_use_fewer_chans
  % Take every third channel (hardcoded).
  desired_recchannels = desired_recchannels(1:3:length(desired_recchannels));
end



%
% Paths.

% Adjust these to match your development environment.

addpath('lib-exp-utils-cjt');
addpath('lib-looputil');
addpath('lib-fieldtrip');
addpath('lib-openephys');
addpath('lib-npy-matlab');

% This automatically adds sub-folders.
addPathsExpUtilsCjt;
```

```
addPathsLoopUtil;



%
% Start Field Trip.

% Wrapping this to suppress the annoying banner.
evalc('ft_defaults');

% Suppress spammy Field Trip notifications.
ft_notice('off');
ft_info('off');
ft_warning('off');



%
% Other setup.

% Suppress Matlab warnings (the NPy library generates these).
oldwarnstate = warning('off');

% Limit the number of channels LoopUtil will load into memory at a time.
% 30 ksps double-precision data takes up about 1 GB per channel-hour.
nlFT_setMemChans(8);



%
% Read metadata (paths, headers, and channel lists).

% Get paths to individual devices.

[ folders_openephys folders_intanrec folders_intanstim folders_unity ] = ...
  euUtil_getExperimentFolders(inputfolder);

% FIXME - For now, assume we're using Open Ephys for the recorder.
% FIXME - Assume one recorder dataset and 0 or 1 stimulator datasets.

folder_record = folders_openephys{1};
have_stim = false;
if ~isempty(folders_intanstim)
  folder_stim = folders_intanstim{1};
  have_stim = true;
end
folder_game = folders_unity{1};


% Get headers.
```

```
% NOTE - Field Trip will throw an exception if this fails.
% Add a try/catch block if you want to fail gracefully.
rechdr = ft_read_header( folder_record, 'headerformat', 'nlFT_readHeader' );
if have_stim
  stimhdr = ft_read_header( folder_stim, 'headerformat', 'nlFT_readHeader' );
end


% Read Open Ephys channel mappings and config files, if we can find them.
% This looks for anything with "config" or "mapping" in the filename.
% FIXME - This is upstream from the "structure.oebin" folder! Search the
% whole tree and hope Intan and Unity don't confuse it.
% FIXME - Assume exactly one valid entry. Sometimes we have multiple configs!

chanmap_rec = euUtil_getOpenEphysChannelMap_v5(inputfolder);

% Turn this into a label-based map.
[ chanmap_rec_raw chanmap_rec_cooked ] = ...
  nlFT_getLabelChannelMapFromNumbers( chanmap_rec.oldchan, ...
    rechdr.label, rechdr.label );

% Translate cooked desired channel names into raw desired channel names.
% FIXME - Only doing this for the recorder!

desired_recchannels = nlFT_mapChannelLabels( desired_recchannels, ...
  chanmap_rec_cooked, chanmap_rec_raw );

badmask = strcmp(desired_recchannels, '');
if sum(badmask) > 0
  disp('###  Couldn''t map all requested recorder channels!');
  desired_recchannels = desired_recchannels(~badmask);
end


% Figure out what channels we want.

[ pat_ephys pat_digital pat_stimcurrent pat_stimflags ] = ...
  euFT_getChannelNamePatterns();

rec_channels_ephys = ft_channelselection( pat_ephys, rechdr.label, {} );
rec_channels_digital = ft_channelselection( pat_digital, rechdr.label, {} );

stim_channels_ephys = ft_channelselection( pat_ephys, stimhdr.label, {} );
stim_channels_digital = ft_channelselection( pat_digital, stimhdr.label, {} );
stim_channels_current = ...
  ft_channelselection( pat_stimcurrent, stimhdr.label, {} );
stim_channels_flags = ft_channelselection( pat_stimflags, stimhdr.label, {} );

% Keep desired channels that match actual channels.
```

```
% FIXME - Ignoring stimulation current and flags!
desired_recchannels = ...
  desired_recchannels( ismember(desired_recchannels, rec_channels_ephys) );
desired_stimchannels = ...
  desired_stimchannels( ismember(desired_stimchannels, stim_channels_ephys) );



%
% Read events.

% Use the default settings for this.

% Read USE and SynchBox events. This also fetches the code definitions.
% This returns each device's event tables as structure fields.
% This also gives its own banner, so we don't need to print one.
[ boxevents gameevents evcodedefs ] = euUSE_readAllUSEEvents(folder_game);

% Now that we have the code definitions, read events and codes from the
% recorder and stimulator.

% These each return a table of TTL events, and a structure with tables for
% each extracted signal we asked for.

disp('-- Reading digital events from recorder.');
[ recevents_ttl recevents ] = euUSE_readAllEphysEvents( ...
  folder_record, recbitsignals, reccodesignals, evcodedefs );

if have_stim
  disp('-- Reading digital events from stimulator.');
  [ stimevents_ttl stimevents ] = euUSE_readAllEphysEvents( ...
    folder_stim, stimbitsignals, stimcodesignals, evcodedefs );
end

% Read USE gaze and framedata tables.
% These return concatenated table data from the relevant USE folders.
% These take a while, so stub them out for testing.
gamegazedata = table();
gameframedata = table();
if ~debug_skip_gaze_and_frame
  disp('-- Reading USE gaze data.');
  gamegazedata = euUSE_readRawGazeData(folder_game);
  disp('-- Reading USE frame data.');
  gameframedata = euUSE_readRawFrameData(folder_game);
  disp('-- Finished reading USE gaze and frame data.');
end


% Report what we found from each device.
```

```
helper_reportEvents('.. From SynchBox:', boxevents);
helper_reportEvents('.. From USE:', gameevents);
helper_reportEvents('.. From recorder:', recevents);
helper_reportEvents('.. From stimulator:', stimevents);


%
% Clean up timestamps.

% Subtract the enormous offset from the Unity timestamps.
% Unity timestamps start at 1 Jan 1970 by default.

[ unityreftime gameevents ] = ...
  euUSE_removeLargeTimeOffset( gameevents, 'unityTime' );
% We have a reference time now; pass it as an argument to ensure consistency.
[ unityreftime boxevents ] = ...
  euUSE_removeLargeTimeOffset( boxevents, 'unityTime', unityreftime );


% Add a "timestamp in seconds" column to the ephys signal tables.

recevents = ...
  euFT_addEventTimestamps( recevents, rechdr.Fs, 'sample', 'recTime' );
stimevents = ...
  euFT_addEventTimestamps( stimevents, stimhdr.Fs, 'sample', 'stimTime' );



%
% Do time alignment.

% Default alignment config is fine.
alignconfig = struct();

% Just align using event codes. Falling back to reward pulses takes too long.


disp('.. Propagating recorder timestamps to SynchBox.');

% Use raw code bytes for this, to avoid glitching from missing box codes.
eventtables = { recevents.rawcodes, boxevents.rawcodes };
[ newtables times_recorder_synchbox ] = euUSE_alignTwoDevices( ...
  eventtables, 'recTime', 'synchBoxTime', alignconfig );

boxevents = euAlign_addTimesToAllTables( ...
  boxevents, 'synchBoxTime', 'recTime', times_recorder_synchbox );


disp('.. Propagating recorder timestamps to USE.');
```

```
% Use cooked codes for this, since both sides have a complete event list.
eventtables = { recevents.cookedcodes, gameevents.cookedcodes };
[ newtables times_recorder_game ] = euUSE_alignTwoDevices( ...
  eventtables, 'recTime', 'unityTime', alignconfig );

gameevents = euAlign_addTimesToAllTables( ...
  gameevents, 'unityTime', 'recTime', times_recorder_game );


if have_stim
  disp('.. Propagating recorder timestamps to stimulator.');

  % The old test script aligned using SynchBox TTL signals as a fallback.
  % Since we're only using codes here, we don't have a fallback option. Use
  % event codes or fail.

  eventtables = { recevents.cookedcodes, stimevents.cookedcodes };
  [ newtables times_recorder_stimulator ] = euUSE_alignTwoDevices( ...
    eventtables, 'recTime', 'stimTime', alignconfig );

  stimevents = euAlign_addTimesToAllTables( ...
    stimevents, 'stimTime', 'recTime', times_recorder_stimulator );


  % Propagate stimulator timestamps to the SynchBox, in case we need to
  % use the SynchBox's event records with the stimulator.

  disp('.. Propagating stimulator timestamps to SynchBox.');

  boxevents = euAlign_addTimesToAllTables( ...
    boxevents, 'recTime', 'stimTime', times_recorder_stimulator );
end


if ~debug_skip_gaze_and_frame

  % First, make "eyeTime" and "unityTime" columns.
  % Remember to subtract the offset from Unity timestamps.

  gameframedata.eyeTime = gameframedata.EyetrackerTimeSeconds;
  gameframedata.unityTime = ...
    gameframedata.SystemTimeSeconds - unityreftime;

  gamegazedata.eyeTime = gamegazedata.time_seconds;


  % Get alignment information for Unity and eye-tracker timestamps.
  % This information is already in gameframedata; we just have to extract
  % it.
```

```
  % Timestamps are not guaranteed to be unique, so filter them.
  times_game_eyetracker = euAlign_getUniqueTimestampTuples( ...
    gameframedata, {'unityTime', 'eyeTime'} );


  % Unity timestamps are unique but ET timestamps aren't.
  % Interpolate new ET timestamps from the Unity timestamps.

  disp('.. Cleaning up eye tracker timestamps in frame data.');

  gameframedata = euAlign_addTimesToTable( gameframedata, ...
    'unityTime', 'eyeTime', times_game_eyetracker );


  % Add recorder timestamps to game and frame data tables.
  % To do this, we'll also have to augment gaze data with unity timestamps.

  disp('.. Propagating recorder timestamps to frame data table.');

  gameframedata = euAlign_addTimesToTable( gameframedata, ...
    'unityTime', 'recTime', times_recorder_game );

  disp('.. Propagating Unity and recorder timestamps to gaze data table.');

  gamegazedata = euAlign_addTimesToTable( gamegazedata, ...
    'eyeTime', 'unityTime', times_game_eyetracker );
  gamegazedata = euAlign_addTimesToTable( gamegazedata, ...
    'unityTime', 'recTime', times_recorder_game );

end


disp('.. Finished time alignment.');



%
% Clean up the event tables.

% Propagate any missing events to the recorder and stimulator.

% We have SynchBox events with accurate timestamps, and we've aligned
% the synchbox to the ephys machines with high precision.

% NOTE - This only works if we do have accurate time alignment. If we fell
% back to guessing in the previous step, the events will be at the wrong
% times.

% Copy missing events from the SynchBox to the recorder.
disp('-- Checking for missing recorder events.');
```

```
recevents = euAlign_copyMissingEventTables( ...
  boxevents, recevents, 'recTime', rechdr.Fs );

% Copy missing events from the SynchBox to the stimulator.
if have_stim
  disp('-- Checking for missing stimulator events.');
  stimevents = euAlign_copyMissingEventTables( ...
    boxevents, stimevents, 'stimTime', stimhdr.Fs );
end


% Copy TTL events into the event code tables, if present.

disp('-- Copying TTL events into event code streams.');

if isfield(recevents, 'cookedcodes')

  if isfield(recevents, 'rwdA')
    recevents.cookedcodes = euFT_addTTLEventsAsCodes( ...
      recevents.cookedcodes, recevents.rwdA, ...
      'recTime', 'codeLabel', 'TTLRwdA' );
  end

  if isfield(recevents, 'rwdB')
    recevents.cookedcodes = euFT_addTTLEventsAsCodes( ...
      recevents.cookedcodes, recevents.rwdB, ...
      'recTime', 'codeLabel', 'TTLRwdB' );
  end

end

if have_stim
  if isfield(stimevents, 'cookedcodes')

    if isfield(stimevents, 'rwdA')
      stimevents.cookedcodes = euFT_addTTLEventsAsCodes( ...
        stimevents.cookedcodes, stimevents.rwdA, ...
        'recTime', 'codeLabel', 'TTLRwdA' );
    end

    if isfield(stimevents, 'rwdB')
      stimevents.cookedcodes = euFT_addTTLEventsAsCodes( ...
        stimevents.cookedcodes, stimevents.rwdB, ...
        'recTime', 'codeLabel', 'TTLRwdB' );
    end

  end
end
```

```
%
% Get trial definitions.


disp('-- Segmenting data into trials.');

% Get event code sequences for "valid" trials (ones where "TrialNumber"
% increased afterwards).

[ trialcodes_each trialcodes_concat ] = euUSE_segmentTrialsByCodes( ...
  gameevents.cookedcodes, 'codeLabel', 'codeData', true );


% FIXME - For debugging (faster and less memory), keep only a few trials.

if debug_use_fewer_trials
  trialcount = length(trialcodes_each);
  if trialcount > debug_trials_to_use
    firsttrial = round(0.5 * (trialcount - debug_trials_to_use));
    lasttrial = firsttrial + debug_trials_to_use - 1;
    firsttrial = max(firsttrial, 1);
    lasttrial = min(lasttrial, trialcount);

    trialcodes_each = trialcodes_each(firsttrial:lasttrial);
    trialcodes_concat = vertcat(trialcodes_each{:});
  end
end


% Get trial definitions.
% This replaces ft_definetrial().

[ rectrialdefs rectrialdeftable ] = euFT_defineTrialsUsingCodes( ...
  trialcodes_concat, 'codeLabel', 'recTime', rechdr.Fs, ...
  padtime, padtime, 'TrlStart', 'TrlEnd', trial_align_evcode, ...
  trial_metadata_events, 'codeData' );

if have_stim
  trialcodes_concat = euAlign_addTimesToTable( trialcodes_concat, ...
    'recTime', 'stimTime', times_recorder_stimulator );

  [ stimtrialdefs stimtrialdeftable ] = euFT_defineTrialsUsingCodes( ...
    trialcodes_concat, 'codeLabel', 'stimTime', stimhdr.Fs, ...
    padtime, padtime, 'TrlStart', 'TrlEnd', trial_align_evcode, ...
    trial_metadata_events, 'codeData' );
end


% FIXME - Sanity check.
```

```
if isempty(rectrialdefs)
  error('No valid recorder trial epochs defined!');
end

if have_stim && isempty(stimtrialdefs)
  error('No valid stimulator trial epochs defined!');
end


% NOTE - You'd normally discard known artifact trials here.



%
% Read the ephys data.

% NOTE - We're reading everything into memory at once. This will only work
% if we have few enough channels to fit in memory. To process more data,
% either read it a few trials at a time or a few channels at a time or at
% a lower sampling rate.

% NOTE - For demonstration purposes, I'm just processing recorder series
% here. For stimulator series, use "stimtrialdefs" and "desired_stimchannels".


% First step: Get wideband data into memory and remove any global ramp.

preproc_config_rec = struct( ...
  'headerfile', folder_record, 'datafile', folder_record, ...
  'headerformat', 'nlFT_readHeader', 'dataformat', 'nlFT_readDataDouble', ...
  'trl', rectrialdefs, 'detrend', 'yes', 'feedback', 'text' );

preproc_config_rec.channel = ...
  ft_channelselection( desired_recchannels, rechdr.label, {} );

disp('.. Reading wideband recorder data.');
recdata_wideband = ft_preprocessing( preproc_config_rec );

% NOTE - We need to turn raw channel labels into cooked channel labels here.

newlabels = nlFT_mapChannelLabels( recdata_wideband.label, ...
  chanmap_rec_raw, chanmap_rec_cooked );

badmask = strcmp(newlabels, '');
if sum(badmask) > 0
  disp('###  Couldn''t map all recorder labels!');
  newlabels(badmask) = {'bogus'};
end
```

```
recdata_wideband.label = newlabels;


% NOTE - You'd normally do re-referencing here.


% Second step: Do notch filtering using our own filter, as FT's brick wall
% filter acts up as of 2021.

disp('.. Performing notch filtering (recorder).');
recdata_wideband = euFT_doBrickNotchRemoval( ...
  recdata_wideband, notch_filter_freqs, notch_filter_bandwidth );


% Third step: Get derived signals (LFP, spike, and rectified activity).

disp('.. Getting LFP, spike, and rectified activity signals.');

[ recdata_lfp, recdata_spike, recdata_activity ] = euFT_getDerivedSignals( ...
  recdata_wideband, lfp_maxfreq, lfp_samprate, spike_minfreq, ...
  rect_bandfreqs, rect_lowpassfreq, rect_samprate, false);


% Fourth step: Pull in gaze data as well.

gazedata_ft = struct([]);

if (~debug_skip_gaze_and_frame) && (~isempty(gameframedata))

  disp('.. Reading and resampling gaze data.');

  % We're reading this from the USE FrameData table.
  % The cooked gaze information gives XY positions.
  % The raw gaze information in "gamegazedata" uses three different
  % eye-tracker-specific coordinate systems. We don't want to deal with that.

  % Trick Field Trip into reading nonuniform tabular data as if it was a file.

  gaze_columns_wanted = { 'EyePositionX', 'EyePositionY', ...
    'RelativeEyePositionX', 'RelativeEyePositionY' };
  gazemaxrectime = max(gameframedata.recTime);
  nlFT_initReadTable( gameframedata, gaze_columns_wanted, ...
    'recTime', 0.0, 10.0 + gazemaxrectime, gaze_samprate, gaze_samprate );

  % Adjust trial definition sample numbers.
  % Time 0 is consistent between recorder and gaze, since we're using
  % "recTime" as the timestamp in both. So all we're doing is resampling.

  gazetrialdefs = ...
    euFT_resampleTrialDefs( rectrialdefs, rechdr.Fs, gaze_samprate );
```

```
  % We're not reading from a file, but Field Trip wants it to still
  % exist, so give it a real folder.

  gazehdr = ...
    ft_read_header( folder_game, 'headerFormat', 'nlFT_readTableHeader' );

  preproc_config_gaze = struct( ...
    'headerfile', folder_game, 'datafile', folder_game, ...
    'headerformat', 'nlFT_readTableHeader', ...
    'dataformat', 'nlFT_readTableData', ...
    'trl', gazetrialdefs, 'feedback', 'text' );
  preproc_config_gaze.channel = ...
    ft_channelselection( gaze_columns_wanted, gazehdr.label, {} );

  gazedata_ft = ft_preprocessing( preproc_config_gaze );

end



%
% Plot trial data.

% NOTE - Just plotting recorder, not stimulator, for the demo.

disp('-- Plotting ephys signals.');

euPlot_plotFTTrials( recdata_wideband, rechdr.Fs, ...
  rectrialdefs, rechdr.Fs, recevents, rechdr.Fs, ...
  { 'oneplot', 'perchannel', 'pertrial' }, ...
  'Recorder Trials - Wideband', [ plotdir filesep 'rec-wb' ] );

euPlot_plotFTTrials( recdata_lfp, lfp_samprate, ...
  rectrialdefs, rechdr.Fs, recevents, rechdr.Fs, ...
  { 'oneplot', 'perchannel', 'pertrial' }, ...
  'Recorder Trials - LFP', [ plotdir filesep 'rec-lfp' ] );

euPlot_plotFTTrials( recdata_spike, rechdr.Fs, ...
  rectrialdefs, rechdr.Fs, recevents, rechdr.Fs, ...
  { 'oneplot', 'perchannel', 'pertrial' }, ...
  'Recorder Trials - High-Pass', [ plotdir filesep 'rec-hpf' ] );

euPlot_plotFTTrials( recdata_activity, rect_samprate, ...
  rectrialdefs, rechdr.Fs, recevents, rechdr.Fs, ...
  { 'oneplot', 'perchannel', 'pertrial' }, ...
  'Recorder Trials - Multi-Unit Activity', [ plotdir filesep 'rec-mua' ] );
```

```
if (~debug_skip_gaze_and_frame) && (~isempty(gameframedata))

  disp('-- Plotting gaze.');

  gaze_chans_abs = { 'EyePositionX', 'EyePositionY' };
  gaze_chans_rel = { 'RelativeEyePositionX', 'RelativeEyePositionY' };

  % Per-trial doesn't tell us much when glancing at it, so just do the stack.

  euPlot_plotAuxData( gazedata_ft, gaze_samprate, ...
    gazetrialdefs, gaze_samprate, recevents, rechdr.Fs, ...
    gaze_chans_abs, { 'oneplot' }, ...
    'Gaze - Absolute', [ plotdir filesep 'gaze-abs' ] );

  euPlot_plotAuxData( gazedata_ft, gaze_samprate, ...
    gazetrialdefs, gaze_samprate, recevents, rechdr.Fs, ...
    gaze_chans_rel, { 'oneplot' }, ...
    'Gaze - Relative', [ plotdir filesep 'gaze-rel' ] );

end

disp('-- Finished plotting.');



%
% Do timelock analysis and plot the results.

% NOTE - Just working with the recorder data, not the stimulator data.
% We didn't load and segment the stimulator data for the demo script.


disp('-- Computing time-locked average and variance of ephys signals.')

% For now, just looing at LFP and MUA. Wideband/HPF is less useful.
% It won't be meaningful to compute this for eye movements, I don't think.

% Default configuration is fine.

recavg_activity = ft_timelockanalysis(struct(), recdata_activity);
recavg_lfp = ft_timelockanalysis(struct(), recdata_lfp);


disp('-- Plotting time-locked average data.');

euPlot_plotFTTimelock( recavg_activity, confsigma, ...
  { 'oneplot', 'perchannel' }, ...
  'Recorder Trials - Average Multi-Unit Activity', ...
  [ plotdir filesep 'rec-avg-mua' ] );
```

```
euPlot_plotFTTimelock( recavg_lfp, confsigma, ...
  { 'oneplot', 'perchannel' }, ...
  'Recorder Trials - Average LFP', ...
  [ plotdir filesep 'rec-avg-lfp' ] );

disp('-- Finished plotting.');



%
% Write data to disk.

% FIXME - NYI.



%
% Done.


%
% Helper functions.


% This writes event counts from a specific device to the console.
% Input is a structure containing zero or more tables of events.

function helper_reportEvents(prefix, eventstruct)
  msgtext = prefix;

  evsigs = fieldnames(eventstruct);
  for evidx = 1:length(evsigs)
    thislabel = evsigs{evidx};
    thisdata = eventstruct.(thislabel);
    msgtext = [ msgtext sprintf('  %d %s', height(thisdata), thislabel) ];
  end

  disp(msgtext);
end



%
% This is the end of the file.
```

Part II

Library Functions

Chapter 2

"euAlign" Functions

2.1 euAlign_addTimesToAllTables.m

```
% function newtables = euAlign_addTimesToAllTables( ...
%   oldtables, oldcolumn, newcolumn, corresptimes )
%
% This function augments several tables with an additional timestamp column.
% these new timestamps are derived from an existing timestamp column using a
% correspondence table produced by euAlign_alignTables().
%
% This is a wrapper for euAlign_addTimesToTable().
%
% "oldtables" is a structure with zero or more fields. Each field contains
%   an event table that is to be augmented.
% "oldcolumn" is the name of the existing table column to use to generate
%   timestamps.
% "newcolumn" is the name of the new table column to generate.
% "corresptimes" is a table containing old and new column timestamps at
%   known-corresponding time points, produced by euAlign_alignTableS().
%
% "newtables" is a copy of "oldtables"; all tables in "newtables" have the
%   new timestamp column added. New timestamp values are linearly
%   interpolated between known points of correspondence.
```

2.2 euAlign_addTimesToTable.m

```
% function newtable = euAlign_addTimesToTable( ...
%   oldtable, oldcolumn, newcolumn, corresptimes )
%
% This function augments a table with an additional timestamp column. These
% new timestamps are derived from an existing timestamp column using a
% correspondence table produced by euAlign_alignTables().
```

```
%
% If the new timestamp column already exists or if it can't be generated,
% "newtable" is a copy of "oldtable".
%
% "oldtable" is the table to augment.
% "oldcolumn" is the name of the existing table column to use to generate
%   timestamps.
% "newcolumn" is the name of the new table column to generate.
% "corresptimes" is a table containing old and new column timestamps at
%   known-corresponding time points, produced by euAlign_alignTables().
%
% "newtable" is a copy of "oldtable" with the new column added. New timestamp
%   values are linearly interpolated between known points of correspondence.
```

2.3 euAlign_alignTables.m

```
% function [ newfirsttab, newsecondtab, ...
%   firstmatchmask, secondmatchmask, timecorresp ] = ...
%   euAlign_alignTables( firsttab, secondtab, ...
%     firsttimelabel, secondtimelabel, firstdatalabel, seconddatalabel, ...
%     coarsewindows, medwindows, finewindow, outliersigma, verbosity )
%
% This function time-aligns data tables from two sources, using both the
% two sources' timestamps and the recorded data values. Timestamps are
% expected to be monotonic (tables get sorted to guarantee this).
%
% If data field names are empty strings, alignment is performed based on
% timestamps alone.
%
% NOTE - This takes O(n) memory but O(n^3) time vs the number of events.
% Using data values helps a lot, as it reduces the number of potential
% matches. Using small window sizes also helps a lot.
%
% Timestamps are typically in seconds, but this will tolerate integer
% timestamps, and will preserve data types for interpolated values.
%
% Each input table is augmented with a new column containing aligned
% timestamp values from the other table. A table with known corresponding
% timestamps is also returned, along with vectors indicating which rows in
% the input tables had matching events in the other table.
%
%
% "Coarse" alignment is performed using a sliding window that moves in steps
% equal to the window radius. We pick one representative sample near the
% middle of the window and consider all samples in the window as alignment
% candidates for it.
%
% An alignment using a constant global delay is performed using the first
```

```
% coarse window value, before sliding-window coarse alignment is performed.
%
% "Medium" alignment uses each sample in turn as the center of a sliding
% window. All samples in the window are considered as alignment candidates
% for the central sample.
%
% "Fine" alignment uses each sample in turn as the center of a sliding
% window. Within each window, all samples are considered to be aligned with
% their closest corresponding candidates. A fixed time offset is applied to
% all samples and optimized to minimize cost given these matchings. The
% result is used as a final correction for the central sample's time offset.
%
%
% "firsttab" is a table containing event tuples from the first source.
% "secondtab" is a table containing event tuples from the second source.
% "firsttimelabel" specifies the first source's column containing timestamps.
% "secondtimelabel" specifies the second source's column containing timestamps.
% "firstdatalabel" if non-empty specifies the first source's column containing
%   data samples to compare.
% "seconddatalabel" if non-empty specifies the second source's column
%   containing data samples to compare.
% "coarsewindows" is a vector containing window half-widths for the first
%   pass of sliding-window time shifting. These are applied from widest to
%   narrowest.
% "medwindows" is a vector containing window half-widths for the second pass
%   of sliding-window time shifting. These are applied from widest to
%   narrowest.
% "finewindow" is the window half-width for final non-uniform time shifting.
%   this freezes the event mapping from the previous step and optimizes delay.
% "outliersigma" is used to reject spurious matches. If, within a window, a
%   match's time delay is this many deviations from the mean, it's squashed.
% "verbosity" is 'verbose', 'normal', or 'quiet'.
%
% "newfirsttab" is a copy of "firsttab" with a "secondtimelabel" column added.
% "newsecondtab" is a copy of "secondtab" with a "firsttimelabel" column.
% "firstmatchmask" is a vector indicating which rows in "firsttab" had
%   matching partners in "secondtab".
% "secondmatchmask" is a vector indicating which rows in "secondtab" had
%   matching partners in "firsttab".
% "timecorresp" is a table with "firsttimelabel" and "secondtimelabel"
%   columns, containing tuples with known-good time alignment. This is the
%   "canonical" set of timestamps used to interpolate the time values in
%   "newfirsttab" and "newsecondtab".
```

2.4 euAlign_alignUsingSlidingWindow.m

```
% function [ bestdeltalist bestcostlist ] = ...
%   euAlign_alignUsingSlidingWindow( ...
```

```
%     firsttimes, secondtimes, firstdata, seconddata, ...
%     firstcandidates, windowrad, costmethod )
%
% This performs sliding window time alignment of two event series, optionally
% matching data between series. This uses the squared distance cost function,
% and assumes that one matching pair of events will have ideal time alignment.
% Alignment is performed with windows centered around each "candidate" event.
%
% NOTE - This may take a while! There are O(c*w) tests, and each test
% takes O(w2) time for local optimization. So, total time is O(c*w3).
% For global optimization, each test takes O(n*w) time, for O(c*n*w2) time
% in total.
%
% Timestamps are typically in seconds, but this will tolerate integer
% timestamps.
%
% "firsttimes" is a vector of timestamps for the first set of events being
%   matched. This is expected to be monotonic (sorted in increasing order).
% "secondtimes" is a vector of timestamps for the second set of events.
%   This is expected to be monotonic (sorted in increasing order).
% "firstdata" is a vector containing data values for each event in the first
%   set. Set this to [] to skip data matching.
% "seconddata" is a vector containing data values for each event in the second
%   set. Set this to [] to skip data matching.
% "firstcandidates" is a vector containing indices of elements within
%   "firsttimes" to find optimal time deltas for.
% "windowrad" is the search distance to use when looking for matching
%   elements. This is the sliding window radius.
% "costmethod" is 'global' or 'local'. If it's 'global', all elements of
%   "firsttimes" and "secondtimes" are used when computing the cost function.
%   If it's 'local', only elements within the window range are used.
%
% "bestdeltalist" is a vector containing time deltas such that
%   tfirst + tdelta = tsecond. Each element in "firstcandidates" has a
%   corresponding element in this list (which is NaN if no matching event
%   was found in the second list).
% "bestcostlist" is a vector containing cost function values corresponding to
%   the elements in "bestdeltalist". Deltas are chosen to minimize cost.
```

2.5 euAlign_alignWithFixedMapping.m

```
% function [ bestdeltalist bestcostlist ] = ...
%   euAlign_alignUsingFixedMapping( ...
%     firsttimes, secondtimes, firstdata, seconddata, ...
%     firstcandidates, windowrad )
%
% This performs sliding window time alignment of two event series, optionally
% matching data between series. This uses the squared distance cost function.
```

```
% Alignment is performed with windows centered around each "candidate" event.
% A single time shift is applied within the window and optimized to minimize
% the cost function.
%
% NOTE - Each event within the window in the first list is assumed to map to
% the nearest event within the window in the second list. So, approximate
% alignment must already have been performed.
%
% NOTE - This takes O(c*w2) time, so use a small window size.
%
% Timestamps are typically in seconds, but this will tolerate integer
% timestamps.
%
% "firsttimes" is a vector of timestamps for the first set of events being
%   matched. This is expected to be monotonic (sorted in increasing order).
% "secondtimes" is a vector of timestamps for the second set of events.
%   This is expected to be monotonic (sorted in increasing order).
% "firstdata" is a vector containing data values for each event in the first
%   set. Set this to [] to skip data matching.
% "seconddata" is a vector containing data values for each event in the second
%   set. Set this to [] to skip data matching.
% "firstcandidates" is a vector containing indices of elements within
%   "firsttimes" to find optimal time deltas for.
% "windowrad" is the search distance to use when looking for matching
%   elements. This is the sliding window radius.
%
% "bestdeltalist" is a vector containing time deltas such that
%   tfirst + tdelta = tsecond. Each element in "firstcandidates" has a
%   corresponding element in this list (which is NaN if no matching event
%   was found in the second list).
% "bestcostlist" is a vector containing cost function values corresponding to
%   the elements in "bestdeltalist". Deltas are chosen to minimize cost.
```

2.6 euAlign_copyMissingEventTables.m

```
% function newtarget = ...
%   euAlign_copyMissingEventTables( evsource, oldtarget, timelabel, samprate )
%
% This function augments a structure containing event tables with new event
% tables copied from a different structure. Timestamps from the specified
% column are translated to sample counts in the new event tables.
%
% "evsource" is a structure with zero or more fields. Each field contains
%   an event table that is to be copied. Empty tables are not copied.
% "oldtarget" is a structure with zero or more fields. Each field contains
%   an event table. Event tables that are missing or empty are replaced.
% "timelabel" is the name of the table column in "evsource" to use when
%   generating sample counts in "newtarget".
```

```
% "samprate" is the sampling rate to use when translating timestamps in
%   seconds into sample counts.
%
% "newtarget" is a copy of "oldtarget" with any missing or empty tables
%   overwritten with translated non-empty tables from "evsource".
```

2.7 euAlign_evalAlignCostFunctionSquare.m

```
% function totalcost = euAlign_evalAlignCostFunctionSquare( ...
%   firsttimes, secondtimes, firstdata, seconddata, windowrad )
%
% This evaluates a cost function for an attempted alignment between two
% event lists. Optionally event data codes are presented that also have to
% match. Only events within the window radius of each other can match.
%
% This particular cost function is the sum of the squared distances between
% matching events. This takes O(n*w) time to compute.
%
% "firsttimes" is a vector of timestamps for the first set of events being
%   matched. This is expected to be monotonic (sorted in increasing order).
% "secondtimes" is a vector of timestamps for the second set of events.
%   This is expected to be monotonic (sorted in increasing order).
% "firstdata" is a vector containing data values for each event in the first
%   set. Set this to [] to skip data matching.
% "seconddata" is a vector containing data values for each event in the second
%   set. Set this to [] to skip data matching.
% "windowrad" is the search distance to use when looking for matching
%   elements.
%
% "totalcost" is the value of the cost function (smaller is better).
```

2.8 euAlign_fakeAlignmentWithExtents.m

```
% function timecorresp = euAlign_fakeAlignmentWithExtents( ...
%   firsttimelabel, firsttimeextents, secondtimelabel, secondtimeextents )
%
% This function produces a fake time alignment reference table using the
% extents of two timestamp series. The fake alignment centers the series
% on each other.
%
% This is intended to be used when there isn't enough information to call
% "euAlign_alignTables()".
%
% NOTE - Time ranges shorter than 1 microsecond are assumed to be bogus.
%
% "firsttimelabel" is a column label for timestamps from the first series.
```

```
% "firsttimeseries" contains timestamp values from the first series. This
%   typically just holds the minimum and maximum timestamp value.
% "secondtimelabel" is a column label for timestamps from the second series.
% "secondtimeseries" contains timestamp values from the second series. This
%   typically just holds the minimum and maximum timestamp value.
%
% "timecorresp" is a table with "firsttimelabel" and "secondtimelabel"
%   columns, containing tuples with corresponding timestamps. This may be
%   used to interpolate time values from one time base to the other.
```

2.9 euAlign_findMatchingEvents.m

```
% function [ firstmatches secondmatches ] = euAlign_findMatchingEvents( ...
%   firsttimes, secondtimes, firstdata, seconddata, windowrad )
%
% This performs sliding-window matching between two event series, finding
% corresponding events from the two input series. Matching events are those
% with the smallest squared time distance. This optionally requires matching
% data values between corresponding events.
%
% NOTE - Matches are a 1:1 mapping. Any given first-list event will match at
% most one second-list event, and vice-versa.
%
% Timestamps are typically in seconds, but this will tolerate integer
% timestamps.
%
% NOTE - This takes O(n*w) time, so it should be fairly fast.
%
% "firsttimes" is a vector of timestamps for the first set of events being
%   matched. This is expected to be monotonic (sorted in increasing order).
% "secondtimes" is a vector of timestamps for the second set of events.
%   This is expected to be monotonic (sorted in increasing order).
% "firstdata" is a vector containing data values for each event in the first
%   set. Set this to [] to skip data matching.
% "seconddata" is a vector containing data values for each event in the second
%   set. Set this to [] to skip data matching.
% "windowrad" is the search distance to use when looking for matching
%   elements. This is the sliding window radius.
%
% "firstmatches" is a vector with one element per entry in the first list
%   containing the index of the matching entry in the second list, or NaN
%   if there was no match.
% "secondmatches" is a vector with one element per entry in the second list
%   containing the index of the matching entry in the first list, or NaN if
%   there was no match.
```

2.10 euAlign_getDefaultAlignConfig.m

```
% function newconfig = euAlign_getDefaultAlignConfig( oldconfig )
%
% This function fills in missing time alignment parameters with reasonable
% default values.
%
% These configuration parameters are intended to be used with
% "euAlign_alignTables()".
%
% "oldconfig" is a structure with zero or more of the following fields:
%   "coarsewindows" is a vector with coarse alignment window half-widths.
%   "medwindows" is a vector with medium alignment window half-widths.
%   "finewindow" is the window half-width for final non-uniform alignment.
%   "outliersigma" is the threshold for rejecting spurious matches.
%   "verbosity" is 'verbose', 'normal', or 'quiet'.
%
% "newconfig" is a copy of "oldconfig" with missing fields added.
```

2.11 euAlign_getSlidingWindowIndices.m

```
% function [ firstindices lastindices ] = ...
%   euAlign_getSlidingWindowIndices( firsttimes, secondtimes, windowrad )
%
% This compiles a list of spans within "secondtimes" that are within a
% search range of any given event within "firsttimes". For each time in
% "firsttimes", a span of indices is found such that times in "secondtimes"
% within that span are in the range (t-radius) to (t+radius).
%
% "firsttimes" is a list of event times to use as window centers. This must
%   be sorted in ascending order.
% "secondtimes" is a list of event times to find windows within. This must
%   be sorted in ascending order.
% "windowrad" is the maximum acceptable difference between a window center
%   time from "firstimes" and an event time within "secondtimes".
%
% "firstindices" and "lastindices" are vectors containing the indices of the
%   first and last valid events in "secondtimes", for each time in
%   "firsttimes". For time firsttimes(k), the valid span in secondtimes is
%   from firstindices(k) to lastindices(k).
%
% NOTE - Entries with no matching elements have "NaN" stored. Check for this.
```

2.12 euAlign_getUniqueTimestampTuples.m

```
% function corresptimes = ...
%   euAlign_getUniqueTimestampTuples( evtable, timecolumns )
%
% This function searches a table of events that are labelled with one or more
% different timestamp columns. Timestamps within a column may repeat; for
% these cases, rows with any timestamp that has already been seen are
% discarded.
%
% "evtable" is the data table to read timestamps from.
% "timecolumns" is a cell array containing timestamp column labels.
%
% "corresptimes" is a table containing _only_ timestamp columns, with
%   timestamp values that are guaranteed to be unique.
```

2.13 euAlign_getWindowExemplars.m

```
% function [ mostidxlist medianidxlist leastidxlist ] = ...
%   euAlign_getWindowExemplars( ...
%     firsttimes, secondtimes, firstdata, seconddata, windowrad )
%
% This picks "exemplar" events from an event series, for purposes of alignment
% between two event series.
%
% The "first" event series is segmented into windows whose overlap is less
% than the window radius. Corresponding windows within the "second" event
% series are found. For each such pair of windows, each event within the first
% window is considered, and the number of potential matches it has in the
% second window is computed (typically requiring matching "data" values).
%
% For each "first" event window, the events having the most potential matches,
% least potential matches, and median number of potential matches are chosen.
% The indices of these events within the "first" event list are returned.
%
% NOTE - This may take a while. It's O(n*w).
%
% "firsttimes" is a vector of timestamps for the first set of events being
%   matched. This is expected to be monotonic (sorted in increasing order).
% "secondtimes" is a vector of timestamps for the second set of events.
%   This is expected to be monotonic (sorted in increasing order).
% "firstdata" is a vector containing data values for each event in the first
%   set. Set this to [] to skip data matching.
% "seconddata" is a vector containing data values for each event in the second
%   set. Set this to [] to skip data matching.
% "windowrad" is the window radius (half-width).
%
```

```
% "mostidxlist" is a vector containing indices of elements within
%   "firsttimes" that are window exemplars with large numbers of matches.
% "medianidxlist" is a vector containing indices of elements within
%   "firsttimes" that are window exemplars with typical numbers of matches.
% "leastidxlist" is a vector containing indices of elements within
%   "firsttimes" that are window exemplars with small numbers of matches.
```

2.14 euAlign_squashOutliers.m

```
% function newdata = euAlign_squashOutliers( ...
%   timeseries, olddata, windowrad, outliersigma )
%
% This performs sliding-window outlier rejection. Data elements that are
% more than the specified number of deviations from the mean within the
% window are replaced with NaN in the output.
%
% There must be at least 6 non-NaN data elements in the window for squashing
% to be performed; otherwise all samples are kept.
%
% "timeseries" is a vector of timestamps for the data values being processed.
% "olddata" is a vector containing data values to remove outliers from.
% "windowrad" is the window half-width to use when evaluating statistics.
% "outliersigma" is the rejection threshold, in deviations from the mean.
%
% "newdata" is a copy of "olddata" with outlier values replaced with NaN.
```

Chapter 3

"euFT" Functions

3.1 euFT_addEventTimestamps.m

```
% function newsignals = ...
%   euFT_addEventTimestamps( oldsignals, samprate, samplabel, timelabel )
%
% This function processes a number of event tables or event structure arrays,
% adding a timestamp column or field derived from the "sample" column or
% field.
%
% "oldsignals" is a structure with zero or more fields. Each field contains
%   either a table with event data or a struct array with event data.
% "samprate" is the sampling rate to use when calculating timestamps.
% "samplabel" is the name of the column or field with the sample number. In
%   Field Trip event lists, this is "sample".
% "timelabel" is the name of the new column or field to add.
%
% "newsignals" is a copy of "oldsignals" where each table or struct array
%   is augmented with a column or field containing timestamps in seconds.
```

3.2 euFT_addTTLEventsAsCodes.m

```
% function newtable = ...
%   euFT_addTTLEventsAsCodes( oldtable, ttltable, timecol, codecol, codelabel )
%
% This adds TTL events to an event code event table.
%
% "oldtable" is the event code event table to add events to.
% "ttltable" is the event table containing TTL events.
% "timecol" is the name of the column containing timestamps. This must be
%   present in both tables.
% "codecol" is the name of the column in "oldtable" that contains code
```

```
%   identification labels.
% "codelabel" is the code identification label to use for added TTL events.
%
% "newtable" is a copy of "oldtable" with TTL events added. Columns from
%   "ttltable" that are present in "oldtable" are copied to the new rows;
%   columns in "ttltable" that are not in "oldtable" are discarded. Columns
%   in "oldtable" that are not in "ttltable" are set to NaN or {''} in the
%   new rows.
```

3.3 euFT_assembleEventWords.m

```
% function wordevents = ...
%   euFT_assembleEventWords( bitlabels, bitevents, wordlabel, firstbit )
%
% This assembles events associated with individual bit changes into a
% sequence of events associated with code word changes.
%
% Channel labels are assumed to end in the bit number. The named bit number
% is treated as the least-significant bit in the code word (this is usually
% 0 or 1, depending on channel label conventions).
%
% NOTE - This only works for LoopUtil events! Those have the channel names
% stored in the event records' "type" field.
%
% NOTE - This uses unsigned 64-bit event words.
%
% "bitlabels" is a cell array containing channel labels for individual bits.
% "bitevents" is an event structure array to search for bit-change events in.
% "wordlabel" is the value to store in the derived list's "type" field.
% "firstbit" is the channel bit number corresponding to the least-significant
%   bit in the output word. This is usually 0 or 1.
%
% "wordevents" is an event structure array containing word-change events.
```

3.4 euFT_defineTrialsUsingCodes.m

```
% function trl = euFT_defineTrialsUsingCodes( ...
%   eventtable, labelfield, timefield, samprate, padbefore, padafter, ...
%   startlabel, stoplabel, alignlabel, codemetatosave, codevaluefield )
%
% This function generates a Field Trip "trl" matrix from a USE event code
% table. The first three columns of "trl" are the starting sample, ending
% sample, and trigger offset (per ft_definetrial()). Remaining columns are
% metadata, which ft_preprocess() will move to a "trialinfo" matrix.
%
% The "trltab" table is a table containing the same data as "trl". This
```

```
% may be used to get column labels for "trl" (as metadata would otherwise
% be hard to identify).
%
% NOTE - The first three columns in 'trltab' are named 'sampstart',
% 'sampend', and 'sampoffset'. The original timestamps for the start, stop,
% and alignment events are saved in 'timestart', 'timeend', and 'timetrigger'.
%
% This can tolerate event labels that are numbers or character arrays.
%
% "eventtable" is a table containing event information. This must at minimum
%   contain event labels and timestamps.
% "labelfield" is the name of the event table column holding event labels.
% "timefield" is the name of the event table column holding timestamps.
% "samprate" is the sampling rate to use for converting timestamps into sample
%   indices.
% "padbefore" is the number of seconds to add before the trial-start event.
% "padafter" is the number of seconds to add after the trial-end event.
% "startlabel" is the event label that indicates the start of a trial.
% "stoplabel" is the event label that indicates the end of a trial.
% "alignlabel" is the event label that indicates the trigger to align trials
%    with.
% "codemetatosave" is a structure describing event code value information to
%    be saved as trial metadata. Each field is an output metadata column
%    name, and that field's value is the event label to look for. When that
%    event label is seen, the event value (from "valuefield") is saved as
%    metadata.
% "codevaluefield" is the name of the event table column holding event values.
%    This is only needed if "codemetatosave" is non-empty.
%
% "trl" is a Field Trip trial definition matrix, per ft_definetrial(). This
%   includes additional metadata columns.
% "trltab" is a table containing the same trial definition data as "trl",
%   with column names.
```

3.5 euFT_doBrickNotchRemoval.m

```
% function newdata = ...
%   euFT_doBrickNotchRemoval( olddata, notch_list, notch_bw )
%
% This performs band-stop filtering in the frequency domain by squashing
% frequency components (a "brick wall" filter). This causes ringing near
% large disturbances (a top-hat in the frequency domain gives a sinc
% function impulse response).
%
% NOTE - This uses the LoopUtil brick-wall filter implementation. To use
% Field Trip's implementation, call euFT_getFiltPowerBrick() to get a FT
% filter configuration structure.
%
```

```
% "olddata" is the FT data structure to process.
% "notch_list" is a vector containing notch center frequencies to remove.
% "notch_bw" is the width of the notch. All notches have the same width.
%
% "newdata" is a copy of "olddata" with trial data waveforms filtered.
```

3.6 euFT_doBrickPowerFilter.m

```
% function newdata = ...
%   euFT_doBrickPowerfilter( olddata, notch_freq, notch_modes, notch_bw )
%
% This performs band-stop filtering in the frequency domain by squashing
% frequency components (a "brick wall" filter). This causes ringing near
% large disturbances (a top-hat in the frequency domain gives a sinc
% function impulse response).
%
% NOTE - This uses the LoopUtil brick-wall filter implementation. To use
% Field Trip's implementation, call euFT_getFiltPowerBrick() to get a FT
% filter configuration structure.
%
% "olddata" is the FT data structure to process.
% "notch_freq" is the fundamental frequency of the family of notches.
% "notch_modes" is the number of frequency modes to remove (1 = fundamental,
%   2 = fundamental and first harmonic, etc).
% "notch_bw" is the width of the notch. Harmonics have the same width.
%
% "newdata" is a copy of "olddata" with trial data waveforms filtered.
```

3.7 euFT_getChannelNamePatterns.m

```
% function [ names_ephys names_digital names_stimcurrent names_stimflags ] = ...
%   euFT_getChannelNamePatterns()
%
% This function returns cell arrays of channel name patterns, suitable for
% use with ft_channelselection(). These will identify different types of
% channel in Open Ephys and Intan data read using the LoopUtil library.
%
% "names_ephys" contains patterns for analog ephys recording channels.
% "names_digital" contains patterns for TTL bit-line and word channels.
% "names_stimcurrent" has patterns for Intan stimulation current channels.
% "names_stimflags" hass patterns for Intan stimulation flag status channels.
```

3.8 euFT_getCodeWordEvent.m

```
% function [ evtable have_events ] = euFT_getCodeWordEvent( ...
%   namelut, wordsigname, bitsignames, firstbit, shiftbitsname, ...
%   headerlabels, allevents )
%
% This looks up channel label patterns for a given signal, looks up channels
% that match those patterns, picks appropriate channels, and finds events
% that are from these channels, merges them into event words, and, returns
% the result as a table.
%
% NOTE - This only works for LoopUtil events! Those have the channel names
% stored in the event records' "type" field.
%
% "namelut" is a structure indexed by signal name that has cell arrays of
%   Field Trip channel label specifiers (per ft_channelselection()), and
%   that also has a field storing the number of bits to shift code words.
% "wordsigname" is the class label to look for for whole-word data. This
%   should only match one Field Trip label.
% "bitsignames" is the wildcard class label to look for for single-bit data.
%   These are treated as word bits, and labels are assumed to end in the bit
%   number.
% "firstbit" is the bit number of the least-significant bit in the word. This
%   is usually 0 or 1, depending on channel label conventions.
% "shiftbitsname" is the LUT field to look for for the bit shift. If this
%   isn't found, a bit shift of 0 is assumed.
% "headerlabels" is the "label" cell array from the Field Trip header.
% "allevents" is the event list to search.
%
% "evtable" is a table containing the filtered event list. This may be empty.
% "have_events" is true if at least one matching event was detected.
```

3.9 euFT_getDerivedSignals.m

```
% function [ datalfp dataspike datarect ] = euFT_getDerivedSignals( ...
%   datawide, lfp_corner, lfp_rate, spike_corner, ...
%   rect_band, rect_lowpass, rect_rate, want_quiet )
%
% This performs filtering to turn a wideband signal into an LFP signal
% (low-pass filtered and downsampled), a spike signal (high-pass filtered),
% and a rectified activity signal (band-pass filtered, rectified, low-pass
% filtered, and then downsampled).
%
% This is a wrapper for ft_preprocessing() and ft_resampledata().
%
% "datawide" is the wideband Field Trip data structure.
% "lfp_corner" is the low-pass corner frequency for the LFP signal.
```

```
% "lfp_rate" is the desired sampling rate of the LFP signal.
% "spike_corner" is the high-pass corner frequency for the spike signal.
% "rect_band" [low high] are the band-pass corners for the rectified signal.
% "rect_lowpass" is the low-pass smoothing corner for the rectified signal.
% "rect_rate" is the desired sampling rate of the rectified signal.
% "want_quiet" is an optional argument. If set to true, progress output is
%   suppressed. If omitted, it defaults to true.
%
% "datalfp" is the LFP signal Field Trip data structure.
% "dataspike" is the spike signal Field Trip data structure.
% "datarect" is the rectified activity signal Field Trip data structure.
```

3.10 euFT_getFiltPowerBrick.m

```
% function cfg = euFT_getFiltPowerBrick( powerfreq, modecount )
%
% This generates a Field Trip ft_preprocessing() configuration structure for
% power-line filtering in the frequency domain using a "brick wall" filter
% (squashing unwanted frequency components). This causes ringing near large
% disturbances (a top-hat in the frequency domain gives a sinc function
% impulse response).
%
% FIXME - We're using the undocumented "brickwall" bsfilttype option to get
% this filter.
%
% FIXME - This doesn't seem to be working properly; it boosts amplitude
% and only slightly attenuates unwanted frequencies.
%
% "powerfreq" is the power-line frequency (typically 50 Hz or 60 Hz).
% "modecount" is the number of modes to include (1 = fundamental,
%   2 = fundamental plus first harmonic, etc).
%
% "cfg" is a Field Trip configuration structure for this filter.
```

3.11 euFT_getFiltPowerCosineFit.m

```
% function cfg = euFT_getFiltPowerCosineFit( powerfreq, modecount )
%
% This generates a Field Trip ft_preprocessing() configuration structure for
% power-line filtering in the frequency domain (DFT filter), using a cosine
% fit (subtracting the specified components).
%
% NOTE - This will work best for short trials. For longer trials, we may be
% able to pick up the fact that we're not exactly at the nominal frequency.
%
% "powerfreq" is the power-line frequency (typically 50 Hz or 60 Hz).
```

```
% "modecount" is the number of modes to include (1 = fundamental,
%   2 = fundamental plus first harmonic, etc).
%
% "cfg" is a Field Trip configuration structure for this filter.
```

3.12 euFT_getFiltPowerDFT.m

```
% function cfg = euFT_getFiltPowerDFT( powerfreq, modecount )
%
% This generates a Field Trip ft_preprocessing() configuration structure for
% power-line filtering in the frequency domain (DFT filter), using the
% "dftbandwidth" option to get a band-stop filter with known bandwidth.
%
% NOTE - This is for short signals only (segmented trials)! For anything
% longer than a few seconds, the type of filter this uses consumes a very
% large amount of memory.
%
% "powerfreq" is the power-line frequency (typically 50 Hz or 60 Hz).
% "modecount" is the number of modes to include (1 = fundamental,
%   2 = fundamental plus first harmonic, etc).
%
% "cfg" is a Field Trip configuration structure for this filter.
```

3.13 euFT_getFiltPowerFIR.m

```
% function cfg = euFT_getFiltPowerFIR( powerfreq, modecount, samprate )
%
% This generates a Field Trip ft_preprocessing() configuration structure for
% power-line filtering, using the "bsfreq" option to get a time-domain
% multi-notch band-stop filter.
%
% NOTE - This is intended for long continuous data, where frequency-domain
% filtering might introduce numerical noise.
%
% NOTE - We're using the FIR implementation of this; the IIR implementation
% is unstable and FT flags it as such.
%
% FIXME - The FIR filter takes forever due to needing a very wide filter and
% Matlab doing convolution in the time domain. We also need to use the
% undocumented "bsfiltord" configuration option.
%
% "powerfreq" is the power-line frequency (typically 50 Hz or 60 Hz).
% "modecount" is the number of modes to include (1 = fundamental,
%   2 = fundamental plus first harmonic, etc).
% "samprate" is the sampling rate of the signal being filtered (needed to
%   calculate the number of points needed for the FIR).
```

```
%
% "cfg" is a Field Trip configuration structure for this filter.
```

3.14 euFT_getFiltPowerTW.m

```
% function cfg = euFT_getFiltPowerTW( powerfreq, modecount )
%
% This generates a Field Trip ft_preprocessing() configuration structure for
% power-line filtering, using Thilo's old configuration. For each mode,
% Thilo specified a comb of frequencies to get something close to a
% band-stop filter. Implementation uses the "DFT" filter, which does a
% cosine fit at each requested frequency in the frequency domain.
%
% NOTE - This will only approximate a band-stop filter for short trials with
% relatively low sampling rates, I think. Frequency uncertainty should be
% comparable to the step size (0.1 Hz).
%
% "powerfreq" is the power-line frequency (typically 50 Hz or 60 Hz).
% "modecount" is the number of modes to include (1 = fundamental,
%   2 = fundamental plus first harmonic, etc).
%
% "cfg" is a Field Trip configuration structure for this filter.
```

3.15 euFT_getSingleBitEvent.m

```
% function [ evtable have_events ] = ...
%   euFT_getSingleBitEvent( namelut, thissigname, headerlabels, allevents )
%
% This looks up a channel label pattern for a given signal, looks up channels
% that match that pattern, picks the first such channel, and then finds
% events that are from that channel, returning the result as a table.
%
% NOTE - This only works for LoopUtil events! Those have the channel labels
% stored in the event records' "type" field.
%
% "namelut" is a structure indexed by signal name that has cell arrays of
%   Field Trip channel label specifiers (per ft_channelselection()).
% "thissigname" is the signal label to look for in "namelut".
% "headerlabels" is the "label" cell array from the Field Trip header.
% "allevents" is the event list to search.
%
% "evtable" is a table containing the filtered event list's fields. This
%   table may be empty.
% "have_events" is true if at least one matching event was detected.
```

3.16 euFT_resampleTrialDefs.m

```
% function newtrialdefs = ...
%   euFT_resampleTrialDefs( oldtrialdefs, oldrate, newrate );
%
% This function alters a trial definition table to account for a changed
% sampling rate. Time zero is expected to be consistent between the
% two representations (i.e. just a scaling, without shifting).
%
% "oldtrialdefs" is a matrix or table containing trial definitions, per
%   ft_definetrial().
% "oldrate" is the sampling rate used with "oldtrialdefs".
% "newrate" is the desired new sampling rate.
%
% "newtrialdefs" is a copy of "oldtrialdefs" with the start, end, and
%   offset columns modified to reflect the new sampling rate.
```

Chapter 4

"euPlot" Functions

4.1 euPlot_axesPlotFTTimelock.m

```
% function euPlot_axesPlotFTTimelock( thisax, ...
%   timelockdata_ft, chans_wanted, bandsigma, plot_timerange, plot_yrange, ...
%   legendpos, figtitle )
%
% This plots the mean and variance of one or more channel waveforms in the
% current axes. Events are rendered as cursors behind the waveforms.
%
% "thisax" is the "axes" object to render to.
% "timelockdata_ft" is a Field Trip structure produced by
%   ft_timelockanalysis().
% "chans_wanted" is a cell array with channel names to plot. Pass an empty
%   cell array to plot all channels.
% "bandsigma" is a scalar indicating where to draw confidence intervals.
%   This is a multiplier for the standard deviation.
% "plot_timerange" [ min max ] is the time range (X range) of the plot axes.
%   Pass an empty range for auto-ranging.
% "plot_yrange" [ min max ] is the Y range of the plot axes.
%   Pass an empty range for auto-ranging.
% "legendpos" is a position specifier to pass to the "legend" command, or
%   'off' to disable the plot legend. The legend lists channel labels.
% "figtitle" is the title to apply to the figure. Pass an empty character
%   array to disable the title.
```

4.2 euPlot_axesPlotFTTrials.m

```
% function euPlot_axesPlotFTTrials( thisax, ...
%   wavedata_ft, wavedata_samprate, trialdefs, trialdef_samprate, ...
%   chans_wanted, plot_timerange, plot_yrange, ...
%   events_codes, events_rwdA, events_rwdB, event_samprate, ...
```

```
%   legendpos, figtitle )
%
% This plots a series of stacked trial waveforms in the current axes.
% Events are rendered as cursors behind the waveforms.
%
% "thisax" is the "axes" object to render to.
% "wavedata_ft" is a Field Trip "datatype_raw" structure with the trial
%   data and metadata.
% "wavedata_samprate" is the sampling rate of "wavedata_ft".
% "trialdefs" is the Field Trip trial definition matrix or table that was
%   used to generate the trial data.
% "trialdefs_samprate" is the sampling rate used when generating "trialdefs".
% "chans_wanted" is a cell array with channel names to plot. Pass an empty
%   cell array to plot all channels.
% "trials_wanted" is a vector with trial indices to plot. Pass an empty
%   vector to plot all trials.
% "plot_timerange" [ min max ] is the time range (X range) of the plot axes.
%   Pass an empty range for auto-ranging.
% "plot_yrange" [ min max ] is the Y range of the plot axes.
%   Pass an empty range for auto-ranging.
% "events_codes" is a Field Trip event structure array or table with event
%   code events. This may be empty.
% "events_rwdA" is a Field Trip event structure array or table with reward
%   line A events. This may be empty.
% "events_rwdB" is a Field Trip event structure array or table with reward
%   line B events. This may be empty.
% "events_samprate" is the sampling rate used when reading events.
% "legendpos" is a position specifier to pass to the "legend" command, or
%   'off' to disable the plot legend. The legend lists channel labels.
% "figtitle" is the title to apply to the figure. Pass an empty character
%   array to disable the title.
```

4.3 euPlot_plotAuxData.m

```
% function euPlot_plotAuxData( auxdata_ft, auxsamprate, ...
%   trialdefs, trialsamprate, evlists, evsamprate, ...
%   chans_wanted, plots_wanted, figtitle, obase )
%
% This plots several channels of auxiliary data in strip chart form and
% saves the resulting plots. Plots may have all trials stacked or be plotted
% per trial or both.
%
% NOTE - Time ranges and decorations are hardcoded.
%
% This is a wrapper for euPlot_axesPlotFTTrials().
%
% "auxdata_ft" is a Field Trip "datatype_raw" structure with the trial data
%   and metadata.
```

```
% "auxsamprate" is the sampling rate of "auxdata_ft".
% "trialdefs" is the field trip trial definition matrix or table that was
%   used to generate the trial data.
% "trialsamprate" is the sampling rate used when generating "trialdefs".
% "evlists" is a structure containing event lists or tables, with one event
%   list or table per field. Fields tested are 'cookedcodes', 'rwdA', and
%   'rwdB'.
% "evsamprate" is the sampling rate used when reading events.
% "chans_wanted" is a cell array with channel names to plot. This cannot be
%   empty. Each listed channel gets one strip (subplot) in the strip chart.
% "plots_wanted" is a cell array containing zero or more of 'oneplot' and
%   'pertrial', controlling which plots are produced.
% "figtitle" is the prefix used when generating figure titles.
% "obase" is the prefix used when generating output filenames.
```

4.4 euPlot_plotFTTimelock.m

```
% function euPlot_plotFTTimelock( ...
%   timelockdata_ft, bandsigma, plots_wanted, figtitle, obase )
%
% This plots a series of time-locked average waveforms and saves the
% resulting plots. Plots may have all channels stacked, or be per-channel,
% or a combination of the above.
%
% NOTE - Time ranges and decorations are hardcoded.
%
% This is a wrapper for euPlot_axesPlotFTTimelock().
%
% "timelockdata_ft" is a Field Trip structure produced by
%   ft_timelockanalysis().
% "bandsigma" is a scalar indicating where to draw confidence intervals.
%   This is a multiplier for the standard deviation.
% "plots_wanted" is a cell array containing zero or more of 'oneplot' and
%   'perchannel', controlling which plots are produced.
% "figtitle" is the prefix used when generating figure titles.
% "obase" is the prefix used when generating output filenames.
```

4.5 euPlot_plotFTTrials.m

```
% function euPlot_plotFTTrials( wavedata_ft, wavesamprate, ...
%   trialdefs, trialsamprate, evlists, evsamprate, ...
%   plots_wanted, figtitle, obase )
%
% This plots a series of stacked trial waveforms and saves the resulting
% plots. Plots may have all trials and channels stacked, or have all
% trials stacked and have one plot per channel, or have all channels
```

```
% stacked and have one plot per trial, or a combination of the above.
%
% NOTE - Time ranges and decorations are hardcoded.
%
% This is a wrapper for euPlot_axesPlotFTTrials().
%
% "wavedata_ft" is a Field Trip "datatype_raw" structure with the trial data
%   and metadata.
% "wavesamprate" is the sampling rate of "wavedata_ft".
% "trialdefs" is the field trip trial definition matrix or table that was
%   used to generate the trial data.
% "trialsamprate" is the sampling rate used when generating "trialdefs".
% "evlists" is a structure containing event lists or tables, with one event
%   list or table per field. Fields tested for are 'cookedcodes', 'rwdA',
%   and 'rwdB'.
% "evsamprate" is the sampling rate used when reading events.
% "plots_wanted" is a cell array containing zero or more of 'oneplot',
%   'perchannel', and 'pertrial', controlling which plots are produced.
% "figtitle" is the prefix used when generating figure titles.
% "obase" is the prefix used when generating output filenames.
```

Chapter 5

"euTools" Functions

5.1 euTools_sanityCheckTree.m

```
% function [ reporttext folderdata ] = ...
%   euTools_sanityCheckTree( startdir, config )
%
% This function searches the specified directory tree for Open Ephys and
% Intan ephys data folders, opens the ephys data files, and checks for
% signal anomalies such as drop-outs and artifacts.
%
% NOTE - This tells Field Trip to use the LoopUtil file I/O hooks. So, it'll
% only work on folders that store data in a way that these support.
%
% "startdir" is the root directory of the tree to search.
% "config" is a structure with some or all of the following fields. Missing
%   fields are set to default values.
%   "wantprogress" is true for progress reports and false otherwise.
%   "readposition" is a number between 0 and 1 indicating where to start
%     reading in the ephys data.
%   "readduration" is the number of seconds of ephys data to read.
%   "chanpatterns" is a cell array containing channel label patterns to
%     process. These are passed to ft_channelselection().
%   "notchfreqs" is a list of frequencies to notch-filter.
%   "notchbandwidth" is the notch bandwidth to use when filtering.
%   "quantization_bits" is the minimum acceptable number of bits of dynamic
%     range in the data.
%   "smoothfreq" is the approximate low-pass corner frequency for smoothing
%     before artifact and dropout checking.
%   "dropout_threshold" is the threshold for detecting drop-outs. This is a
%     multiple of the median value, and should be less than 1.
%   "artifact_threshold" is the threshold for detecting artifacts. This is
%     a multiple of the median value, and should be greater than 1.
%   "frac_samples_bad" is the minimum fraction of bad samples in a channel
%     needed for that channel to be flagged as bad.
```

```
%   "lowpasscorner" is the low-pass-filter corner used for generating LFP data.
%   "lowpassrate" is the sampling rate to use when downsampling LFP data.
%   "correlthreshabs" is the absolute r-value threshold for considering
%     channels to be correlated. This should be in the range 0..1.
%   "correlthreshrel" is the relative r-value threshold for considering
%     channels to be correlated. This should be greater than 1.
%
% "reporttext" is a character vector containing a human-readable summary
%   of the sanity check.
% "folderdata" is an array of structures with the following fields:
%   "datadir" is the path containing the ephys data files.
%   "config" is a copy of the configuration structure with missing values
%     set to appropriate defaults.
%   "ftheader" is the Field Trip header for the ephys data.
%   "samprange" [min max] is the range of samples read for the test.
%   "isfloating" is a boolean vector indicating whether channels were members
%     of correlated groups (which usually means floating channels).
%   "isquantized" is a boolean vector indicating whether channel data showed
%     quantization.
%   "hadartifacts" is a boolean vector indicating whether channel data had
%     large amplitude excursions (electrical artifacts).
%   "haddropouts" is a boolean vector indicating whether channel data had
%     intervals with no data (usually a throughput problem).
```

Chapter 6

"euUSE" Notes

6.1 EVCODEDEFS.txt

```
There are two types of event code definition structure: "raw" and "cooked".


The "raw" event code definition structure is the structure returned by
calling "jsondecode" on the contents of the "eventcodes_USE_05.json" file.

This is a structure with one field per event code, with the field name being
the event code's name. Each field contains a structure with the following
fields (upper-case leading):

- "Value" is an integer value in the range 0..65025.
- "Description" is a character array with human-readable text describing this
  event code.

Note that there are often pairs of codes with the name "FooMin" and "FooMax"
describing ranges of code values that are interpreted per their descriptions.


The "cooked" event code definition structure has one field per event code,
with the field name being the event code's name. Each field contains a
structure with the following fields (lower-case):

- "value" is either a scalar integer value or a [min max] pair of integer
  values (describing a ranged code).
- "description" is a cell array containing character arrays with
  human-readable text describing this event code.
- "offset" (optional) is a number to be subtracted from the code value to
  convert it into a processed data value.
- "multiplier" (optional) is a factor by which the code value is to be
  multiplied (after offset subtraction) to convert it into a processed value.
```

```
When turning an event code value into a processed value, the formula used is:
  processed = multiplier * (raw - offset)

Pairs of raw codes of the form "FooMin" and "FooMax" are turned into single
cooked code definitions for the code "Foo". The description strings from
"FooMin" and "FooMax" are usually identical, but both are stored in the
cooked definition as a precaution.



This is the end of the file.
```

6.2 FILES.txt

```
Files that we read from USE, as of 03 Feb 2022:


The "RuntimeData" folder has files of interest in the following subfolders:

- "SerialSent" has messages sent from USE to the SynchBox, with Unity
  timestamps. These are per-trial plus a startup file.
- "SerialRecv" has messages sent from the SynchBox to USE, with both
  Unity and SynchBox timestamps. These are per-trial plus a startup file.
- "GazeData" has eye-tracker data in TSV format. These are per-trial.
- "FrameData" has USE state information, including eye tracker timestamps,
  in TSV format.

NOTE - Trial files are stored as "_N.txt", "_NN.txt", etc; either use the
third-party "sort_nat" to sort them, or sort data based on timestamps.


The "ProcessedData" folder has ".mat" files with data tables:

- "SerialData.mat" has SerialSentData and SerialRecvData tables.
- "RawGazeData" has the gaze data.


The "use/USE_Analysis" scripts on GitHub are used for converting raw into
processed data.

More recent ones are in "use_analysis" on bitbucket (private repository).
```

Chapter 7

"euUSE" Functions

7.1 euUSE_aggregateTrialFiles.m

```
% function tabdata = euUSE_aggregateTrialFiles(filepattern, sortcolumn)
%
% This reads the specified set of tab-delimited text files, aggregating the
% data contained within. The resulting combined table is sorted based on the
% specified column and returned.
%
% This is intended to be used with per-trial files, sorting on the timestamp.
%
% "filepattern" is the path-plus-wildcards file specifier to pass to dir().
% "sortcolumn" is the name of the table column to sort table rows with.
%
% "tabdata" is the resulting sorted merged table.
```

7.2 euUSE_alignTwoDevices.m

```
% function [ alignedevents timecorresp ] = euUSE_alignTwoDevices( ...
%   eventtables, firsttimecol, secondtimecol, alignconfig )
%
% This function performs time alignment between two devices by examining
% correlated event sequences between the devices.
%
% This is a wrapper for "euAlign_alignTables()".
%
% NOTE - If it can't find events in common, it centers the time series on
% each other (using the maximum extents of each source). If it can't even do
% that (for example if one source has no events of any type), it lines them
% up by fiat (declaring missing extents to be 0 to 3600 seconds).
% FIXME - We should really have a return flag to indicate this. Right now
% it just shows up on console output.
```

```
%
% "eventtables" is a Nx2 cell array. Each row contains two tables with
%   events that ostensibly match each other. The first row where both tables
%   are non-empty is used for alignment.
% "firsttimecol" is the name of the timestamp column in tables stored in
%   column 1 of "eventtables".
% "secondtimecol" is the name of the timestamp column in tables stored in
%   column 2 of "eventtables".
% "alignconfig" is a structure with zero or more of the following fields:
%   "coarsewindows" is a vector with coarse alignment window half-widths.
%   "medwindows" is a vector with medium alignment window half-widths.
%   "finewindow" is the window half-width for final non-uniform alignment.
%   "outliersigma" is the threshold for rejecting spurious matches.
%   "verbosity" is 'verbose', 'normal', or 'quiet'.
%   Missing fields will be set to reasonable defaults.
%
% "alignedevents" is a copy of "eventtables" with the tables in column 1
%   augmented with "secondtimecol" timestamps and the tables in column 2
%   augmented with "firsttimecol" timestamps.
% "timecorresp" is a table with "firsttimecol" and "secondtimecol" columns,
%   containing tuples with known-good time alignment. This is the "canonical"
%   set of timestamps used to interpolate time values between tables.
```

7.3 euUSE_cleanEventsTabular.m

```
% function newtable = euUSE_cleanEventsTabular( oldtable, datacol, timecol )
%
% This processes a data table of raw code word events, removing entries that
% have a value of zero (the idle state) and merging events that are one
% sample apart (which happens when event codes change at a sampling boundary).
%
% "oldtable" is a table containing event tuples to process.
% "datacol" is the label of the column that contains data values.
% "timecol" is the label of the column that contains timestamps (in samples).
%
% "newtable" is the revised table.
```

7.4 euUSE_deglitchEvents.m

```
% function [ newvals newtimes ] = euUSE_deglitchEvents( oldvals, oldtimes )
%
% This checks a list of event timestamps for events that are one sample
% apart, and merges them. This happens when event codes change at a sampling
% boundary.
%
% This will also catch the situation where events are reported multiple
```

```
% times with the same timestamp.
%
% "oldvals" is a list of event data samples (integer data values).
% "oldtimes" is a list of event timestamps (in samples).
%
% "newvals" is a revised list of event data samples.
% "newtimes" is a revised list of event timestamps.
```

7.5 euUSE_getEventCodeDefOverrides.m

```
% function defoverrides = euUSE_getEventCodeDefOverrides()
%
% This function provides a default "hints" structure for parsing event code
% definitions. This is mostly used for changing the range of codes that have
% ranged values encoded.
%
% NOTE - This is entirely composed of magic values reflecting our current
% USE configuration, and will have to be manually edited if that changes.
%
% "defoverrides" is a structure suitable for passing to
%   euUSE_parseEventCodeDefs().
```

7.6 euUSE_lookUpEventCode.m

```
% function [ codedata codelabel ] = euUSE_loopUpEventCode( codeword, codedefs)
%
% This function looks up the event code definition for a specified code word.
% The translated event code data value, if applicable, is generated.
%
% "codeword" is the event code word to translate.
% "codedefs" is a "cooked" event code definition structure per
%   "EVCODEDEFS.txt".
%
% "codedata" is a translated data value corresponding to this event code, if
%   translation is appropriate, or a copy of "codeword" if not.
% "codelabel" is the field name of the corresponding entry in "codedefs", if
%   the code was understood. This is usually a human-readable code name. If
%   the code was not recognized, this is an empty character array ('').
```

7.7 euUSE_parseEventCodeDefs.m

```
% function cookeddefs = euUSE_parseEventCodeDefs( rawdefs, overrides )
%
```

```
% This parses raw event code definitions (as given by "jsondecode") and
% builds a structure containing processed event definitions.
%
% "rawdefs" is the decoded JSON event code definition structure.
% "overrides" is a structure containing specific event code fields that
%   overwrite automatically-generated fields in "cookeddefs".
%
% "cookeddefs" is a structure with fields indexed by event code names, each
%   of which contains a structure with the following fields (per
%   EVCODEDEFS.txt):
%   "value" is a scalar or a two-element vector, containing the event code
%     value or range of values ( [ min max ] ) for this code.
%   "description" is a cell array containing human-readable description
%     strings for the event code. There may be multiple strings if the code
%     is defined by multiple entries in "rawdefs" (as with ranged codes).
%   "offset" (optional) is a number to be subtracted from the code value
%     to convert it into a processed data value.
%   "multiplier" (optional) is a factor by which the code value is to be
%     multiplied to convert it into a processed value. This is not
%     automatically generated.
%
% When turning an event code value into a processed value, the formula used
% is:  processed = multiplier * (raw - offset)
%
% The idea is that the automatic parsing generates reasonable guesses for the
% interpretation of "FooMin" and "FooMax" definition pairs, and the
% "overrides" structure can modify these interpretations for known cases
% where automatic guesses aren't correct.
```

7.8 euUSE_parseSerialRecvData.m

```
% function [ evsynchA evsynchB evrwdA evrwdB evcodes ] = ...
%   euUSE_parseSerialRecvData( serialrecvdata, codeformat )
%
% This function parses the "SerialRecvData" table from a "serialData"
% structure, read from the "SerialData.mat" file produced by the USE
% processing scripts. It may alternatively be read by using the
% euUSE_readRawSerialData() function.
%
% This table contains communication received by Unity from the SynchBox,
% which includes Unity and SynchBox timestamps. Timing, reward, and event
% code messages are parsed, and are returned as separate tables. Each table
% contains Unity and SynchBox timestamps (converted to seconds); the reward
% tables also contain reward pulse duration in seconds, and event code
% tables contain the event code value.
%
% Event codes may be in any of several formats; "word" format codes are
% preserved as-is, while "byte" format codes are shortened to 8 bits. The
```

```
% byte may be taken from the most-significant or least-significant 8 bits
% of the code word. For "dupbyte", the most significant and least significant
% 8 bits are expected to contain the same values; any codes that don't are
% rejected.
%
% "serialrecvdata" is the data table containing raw inbound serial data.
% "codeformat" is 'word' (for 16-bit words), 'hibyte' for MS bytes, 'lobyte'
%   for LS bytes, and 'dupbyte' for MS and LS both replicating a byte value.
%
% "evsynchA" and "evsynchB" are tables containing timing A and B events.
%   Table columns are 'unityTime' and 'synchBoxTime'.
% "evrwdA" and "evrwdB" are tables containing reward trigger A and B events.
%   Table columns are 'unityTime', 'synchBoxTime', and 'pulseDuration'.
% "evcodes" is a table containing event code events. Table columns are
%   'unityTime', 'synchBoxTime', and 'codeValue'.
```

7.9 euUSE_parseSerialSentData.m

```
% function [ evrwdA evrwdB evcodes ] = ...
%   euUSE_parseSerialSentData( serialsentdata, codeformat )
%
% This function parses the "SerialSentData" table from a "serialData"
% structure, read from the "SerialData.mat" file produced by the USE
% processing scripts. It may alternatively be read by using the
% euUSE_readRawSerialData() function.
%
% This table contains communication sent from Unity to the SynchBox, which
% includes Unity timestamps (but no synchbox timestamps). Reward and event
% code messages are parsed, and are returned as separate tables. Each table
% contains Unity timestamps (converted to seconds); the reward tables also
% contain reward pulse duration in seconds, and event code tables contain
% the event code value.
%
% Event codes may be in any of several formats; "word" format codes are
% preserved as-is, while "byte" format codes are shortened to 8 bits. The
% byte may be taken from the most-significant or least-significant 8 bits
% of the code word. For "dupbyte", the most significant and least significant
% 8 bits are expected to contain the same values; any codes that don't are
% rejected.
%
% "serialsentdata" is the data table containing raw outbound serial data.
% "codeformat" is 'word' (for 16-bit words), 'hibyte' for MS bytes, 'lobyte'
%   for LS bytes, and 'dupbyte' for MS and LS both replicating a byte value.
%
% "evrwdA" and "evrwdB" are tables containing reward trigger A and B events.
%   Table columns are 'unityTime' and 'pulseDuration'.
% "evcodes" is a table containing event code events. Table columns are
%   'unityTime' and 'codeValue'.
```

7.10 euUSE_readAllEphysEvents.m

```
% function [ ttlevents cookedevents ] = ...
%   euUSE_readAllEphysEvents( ephysfolder, bitsignaldefs, codesignaldefs, ...
%   evcodedefs, codebytes, codeendian )
%
% This function reads TTL events from one ephys device and parses these into
% synch, reward, and event code events.
%
% This is a wrapper for the following functions:
%   ft_read_header()
%   ft_read_event()
%   euFT_getSingleBitEvent()
%   euFT_getCodeWordEvent()
%   euUSE_cleanEventsTabular()
%   euUSE_reassembleEventCodes()
%
% NOTE - Channel specifiers (individual or wildcard-based) have the format
% used by ft_channelselection().
%
% "ephysfolder" is the folder containing "structure.oebin", "info.rhd", or
%   "info.rhs".
% "bitsignaldefs" is a structure indexed by signal name, with each field
%   containing the TTL channel name with that signal's events.
%   If the structure is empty, no single-bit signals are recorded.
% "codesignaldefs" is a structure with the following fields:
%   "signameraw" is the output event code signal name for untranslated bytes.
%   "signamecooked" is the output event code signal name for event codes.
%   "channame" is the TTL channel name to convert. This may be a single
%     channel (for word-based TTL data) or a wildcard expression (for
%     single-bit TTL data).
%   "bitshift" is the number of bits to shift to the right, if reassembling
%     from bits. This is also used to compensate for 1-based numbering (the
%     channel names for bit lines are assumed to start at 0).
%   If there's only one matching channel, it's assumed to contain word data.
%   otherwise it's assumed to contain bit data.
%   If the structure is empty, no event codes are recorded.
% "evcodedefs" is a USE event code definition structure per EVCODEDEFS.txt.
% "codebytes" is the number of bytes used to encode each event code.
%   This defaults to 2 if unspecified.
% "codeendian" is 'big' if the most significant byte is received first or
%   'little' if the least-significant byte is received first.
%   This defaults to 'big' if unspecified.
%
% "ttlevents" is the raw TTL event list returned by ft_read_event().
% "cookedevents" is a structure with one field per signal listed in
%   "bitsignaldefs" and "codesignaldefs". Each field contains a table
%   of matching events whose columns correspond to the event list's fields.
%   These tables may be empty if no events were found.
```

7.11 euUSE_readAllUSEEvents.m

```
% function [ boxevents gameevents evcodedefs ] = ...
%   euUSE_readAllUSEEvents( runtimefolder, codeformat, codebytes, codeendian )
%
% This function reads and parses serial data from a USE "RuntimeData" folder.
% This gives events and timestamps from USE and from the SynchBox.
%
% This is a wrapper for the following functions:
%   euUSE_readRawSerialData()
%   euUSE_parseSerialRecvData()
%   euUSE_parseSerialSentData()
%   euUSE_readEventCodeDefs()
%   euUSE_reassembleEventCodes()
%
% "runtimefolder" is the path to the "RuntimeData" folder.
% "codeformat" is the event code format used by the SynchBox. This is 'word',
%   'hibyte', 'lobyte', or 'dupbyte', per euUSE_parseSerialRecvData().
%   This defaults to 'dupbyte' if unspecified.
% "codebytes" is the number of bytes used to encode each event code.
%   This defaults to 2 if unspecified.
% "codeendian" is 'big' if the most significant byte is received first or
%   'little' if the least-significant byte is received first.
%    This defaults to 'big' if unspecified.
%
% "boxevents" is a structure with fields "synchA", "synchB", "rwdA", "rwdB",
%   "rawcodes", and "cookedcodes". These each contain tables of events.
%   Common columns are 'unityTime' and 'synchBoxTime'. Additional columns are
%   'pulseDuration' (for rewards), 'codeValue' (for raw codes), and
%   'codeWord', 'codeData', and 'codeLabel' (for cooked codes).
% "gameevents" is a structure with fields "rwdA", "rwdB", "rawcodes", and
%   "cookedcodes". These each contain tables of events. All tables contain
%   a 'unityTime' column. Additional columns are 'pulseDuration' (for
%   rewards), 'codeValue' (for raw codes), and 'codeWord', 'codeData', and
%   'codeLabel' (for cooked codes).
% "evcodedefs" is a USE event code definition structure per EVCODEDEFS.txt.
```

7.12 euUSE_readEventCodeDefs.m

```
% function defscooked = euUSE_readEventCodeDefs( runtimedir )
%
% This function reads the "eventcodes_USE_05.json" file and returns a
% "cooked" event code definition structure, per EVCODEDEFS.txt.
%
% "runtimedir" is the "RuntimeData" directory location.
%
% "defscooked" is a "cooked" event code definition structure.
```

7.13 euUSE_readRawFrameData.m

```
% function framedata = euUSE_readRawFrameData( runtimedir )
%
% This function looks for "*_Trial_(number).txt" files in the FrameData folder
% in the specified directory, and converts them into an aggregated Matlab
% table with rows sorted by timestamp.
%
% New timestamp columns ("SystemTimeSeconds" and "EyetrackerTimeSeconds")
% are generated from the respective native timestamp columns.
%
% "runtimedir" is the "RuntimeData" directory location.
%
% "framedata" is an aggregated data table.
```

7.14 euUSE_readRawGazeData.m

```
% function gazedata = euUSE_readRawGazeData( runtimedir )
%
% This function looks for "*_Trial_(number).txt" files in the GazeData folder
% in the specified directory, and converts them into an aggregated Matlab
% table with rows sorted by timestamp.
%
% A new timestamp column ("time_seconds") is generated from the native gaze
% timestamp column.
%
% "runtimedir" is the "RuntimeData" directory location.
%
% "gazedata" is an aggregated data table.
```

7.15 euUSE_readRawSerialData.m

```
% function [ sentdata recvdata ] = euUSE_readRawSerialData( runtimedir )
%
% This function looks for "*_Trial_(number).txt" files in the SerialSent
% and SerialRecv folders in the specified directory, and converts them
% into aggregated Matlab tables with rows sorted by Unity timestamp.
%
% "runtimedir" is the "RuntimeData" directory location.
%
% "sentdata" is aggregated data from trial files in the "SerialSent" folder.
% "recvdata" is aggregated data from trial files in the "SerialRecv" folder.
```

7.16 euUSE_reassembleEventCodes.m

```
% function [ cookedcodes cookedindices ] = euUSE_reassembleEventCodes( ...
%   rawcodes, codedefs, codebytes, codeendian, rawcolumn )
%
% This translates a data table containing raw (byte) event codes into a
% table containing cooked (word) event codes.
%
% This recovers from dropped bytes. Anything unrecognized (due to dropped
% bytes or just not being in the definition file) is tagged with an empty
% character array as the "codeLabel".
%
% "rawcodes" is a table containing a raw code value column and optionally
%   other columns.
% "codedefs" is a structure containing "cooked" event code definitions per
%   "parseEventCodeDefs" and "EVCODEDEFS.txt".
% "codebytes" is the number of bytes per cooked code.
% "codeendian" is 'big' if the most-significant byte is received first
%   or 'little' if the least-significant byte is received first.
% "rawcolumn" is the name of the column to read raw codes from.
%
% "cookedcodes" is a table with the following columns:
%   "codeWord" is the reconstructed event code word value.
%   "codeData" is (code number - offset) * multiplier, per "EVCODEDEFS.txt".
%   "codeLabel" is the corresponding "codedefs" field name for this code.
%     This is usually a human-readable code name.
%   Other columns from "rawcodes" are copied for rows corresponding to the
%   first byte of each cooked code. These are typically timestamps.
% "cookedindices" is a vector with length equal to the number of rows in
%   "cookedcodes", containing indices (row numbers) pointing to the
%   locations of the corresponding first code bytes in "rawcodes".
```

7.17 euUSE_removeLargeTimeOffset.m

```
% function [ reftime newsignals ] = ...
%   euUSE_removeLargeTimeOffset( oldsignals, timecolumn, desiredreftime )
%
% This function subtracts a large time offset from all signal tables in
% the provided structure.
%
% This is intended to be used to modify Unity timestamps, which are relative
% to 1 Jan 1970.
%
% "oldsignals" is a structure with fields that each contain a table of
%   timestamped events.
% "timecolumn" is the name of the table column that contains timestamps.
% "desiredreftime" is a desired time offset to be subtracted from all
```

```
%   timestamps. If this argument is omitted, and arbitrary offset is chosen.
%
% "reftime" is the time value that was subtracted from all timestamps.
% "newsignals" is a copy of "oldsignals" with "reftime" subtracted from all
%   tables' timestamp columns.
```

7.18 euUSE_segmentTrialsByCodes.m

```
% function [ pertrialtabs alltrialtab ] = ...
%   euUSE_segmentTrialsByCodes( rawtab, labelfield, valuefield, discardbad )
%
% This function processes a table containing event code sequences, segmenting
% the sequence into trials and optionally discarding bad trials.
%
% Trials are demarked by 'TrlStart' and 'TrlEnd' codes. "Good" trials are
% those for which 'TrialNumber' incremented.
%
% Additional columns in "rawtab" are copied to the output tables. These are
% typically things like timestamps.
%
% "rawtab" is the table to process. It must contain columns with event code
%   labels and (if discarding bad trials) event code data values.
% "labelfield" is the name of the table column that has code label character
%   arrays.
% "valuefield" is the name of the table column that has code data values.
%   This is ignored if bad trials aren't being filtered.
% "discardbad" is true if bad trials are to be discarded and false otherwise.
%
% "pertrialtabs" is a cell array containing event code sequence tables for
%   each trial.
% "alltrialtab" is an event code sequence table containing all trials
%   (equal to the concatenated contents of "pertrialtabs").
```

7.19 euUSE_translateGazeIntoSignals.m

```
% function dataset = euUSE_translateGazeIntoSignals( ...
%   gazetable, timecol, ignorecols, samprate )
%
% This translates tabular gaze data read from USE into a Field Trip data
% structure containing continuous waveform data.
%
% NOTE - The signal names produced by this will vary depending on the
% eye-tracker used.
%
% NOTE - Do not pick a sampling rate that exactly matches the eye-tracker
% rate. That will almost certainly result in beat frequencies in the data.
```

```
%
% "gazetable" is a table containing raw USE gaze data.
% "timecol" is the label of the table column to read timestamps from.
% "ignorecols" is a cell array containing labels of columns to not copy.
% "samprate" is the sampling rate to use for the output data.
%
% "dataset" is a ft_datatype_raw structure containing gaze data signals.
```

Chapter 8

"euUtil" Functions

8.1 euUtil_getExperimentFolders.m

```
% function [ dirs_opene dirs_intanrec dirs_intanstim dirs_use ] = ...
%   euUtil_getExperimentFolders( topdir )
%
% This function searches a directory tree, looking for subfolders containing
% Open Ephys data (structure.oebin), Intan data (info.rhs/info.rhd), and
% USE data (RuntimeData folder).
%
% "topdir" is the folder to search.
%
% "dirs_opene" is a cell array containing paths to Open Ephys folders.
% "dirs_intanrec" is a cell array containing paths to Intan recorder folders.
% "dirs_intanstim" is a cell array with paths to Intan stimulator folders.
% "dirs_use" is a cell array with paths to USE "RuntimeData" folders.
```

8.2 euUtil_getOpenEphysChannelMap_v5.m

```
% function chanmap = euUtil_getOpenEphysChannelMap_v5( inputfolder )
%
% This searches the specified tree looking for Open Ephys configuration
% files and channel mapping files (anything with "config" or "mapping" in
% the filename). Channel maps are extracted, and the first map found is
% returned. If no maps are found, an empty structure array is returned.
%
% This is a wrapped for "euUtil_getOpenEphysConfigFiles",
% "nlOpenE_parseChannelMapJSON_v5", and "nlOpenE_parseChannelMapXML_v5".
%
% "inputfolder" is the top-level folder to search.
%
% "chanmap" is a structure with the following fields:
```

```
%   "oldchan" is a vector indexed by new channel number containing the old
%     channel number that maps to each new location, or NaN if none does.
%   "oldref" is a vector indexed by new channel number containing the old
%     channel number to be used as a reference for each new location, or
%     NaN if unspecified.
%   "isenabled" is a vector of boolean values indexed by new channel number
%     indicating which new channels are enabled.
```

8.3 euUtil_getOpenEphysConfigFiles.m

```
% function [ configfiles mapfiles ] = euUtil_getOpenEphysConfigFiles( topdir )
%
% This function looks for files with "Config" or "Mapping" in the name (not
% case-sensitive) and reports their full names (including path).
%
% This is intended to be used to find Open Ephys channel mapping and
% configuration files.
%
% "topdir" is the top-level folder to search (usually the Open Ephys folder).
%
% "configfiles" is a cell array containing full filenames that had "config".
% "mapfiles" is a cell array containing full filenames that had "mapping".
```

8.4 euUtil_makePrettyTime.m

```
% function durstring = euUtil_makePrettyTime(dursecs)
%
% This formats a duration (in seconds) in a meaningful human-readable way.
% Examples would be "5.0 ms" or "5d12h".
%
% "dursecs" is a duration in seconds to format. This may be fractional.
%
% "durstring" is a character array containing a terse human-readable
%   summary of the duration.
```

8.5 euUtil_makeSafeString.m

```
% function [ newlabel newtitle ] = euUtil_makeSafeString( oldstring )
%
% This function makes label- and title-safe versions of an input string.
% The label-safe version strips anything that's not alphanumeric.
% The title-safe version replaces stripped characters with spaces.
%
```

```
% This is more aggressive than filename- or fieldname-safe strings; in
% particular, underscores are interpreted as typesetting metacharacters
% in plot labels and titles.
%
% "oldstring" is the string to convert.
%
% "newlabel" is a string with only alphanumeric characters.
% "newtitle" is a string with non-alphanumeric characters replaced with spaces.
```