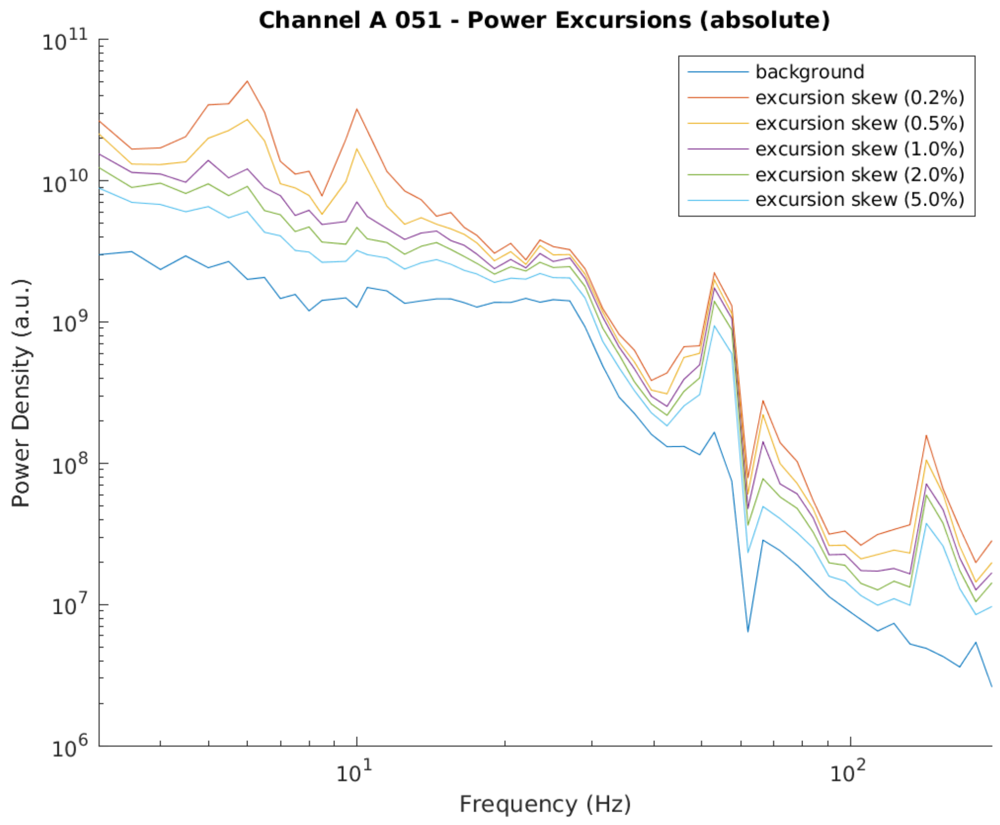
NeuroLoop Utilities – Function Reference

Written by Christopher Thomas – January 12, 2023.



Overview

The NeuroLoop utility library functions are divided into several categories:

- **“Core”** functions (part I) are functions useful in a wide variety of contexts (not vendor- or application-specific).
- **“Abstraction”** functions (part II) are functions used to support specific hardware devices or software suites.
- **“Application”** functions (part III) are functions used by individual application programs that are either designed specifically to interact with that application’s code or that perform tasks specific enough to limit reusability.
- Lastly, part IV contains sample code that may help illustrate the ways these libraries may be used.

“Core” function libraries are as follows:

- **“Processing”** functions (ch. 2) perform signal processing operations on data series such as filtering, artifact rejection, statistical calculations, and so forth.
- **“Utility”** functions (ch. 3) perform miscellaneous operations that are not covered by the other categories.
- **“I/O”** functions (ch. 5) facilitate operations for reading and writing data that aren’t vendor-specific.
- **“Plotting”** functions (ch. 4) render plots of various types to files, figures, or axes. These are included as an aid to rapid prototyping, and are used by the GUI scripts. The output is generally not publication-ready.

“Abstraction” function libraries are as follows:

- **“Field Trip”** functions (ch. 7) facilitate interoperation with Field Trip.
- **“Intan”** functions (ch. 8) facilitate the use of datasets stored in Intan’s format.
- **“Vendor-Supplied Intan”** functions (ch. 9) are functions derived from vendor-supplied code that are in turn wrapped by the functions in ch. 8.
- **“Open Ephys”** functions (ch. 10) facilitate the use of datasets stored in Open Ephys’s format.

"Application" library functions are as follows:

- **"Channel Tool"** functions (ch. 12) perform operations used by the "Channel Tool" utility. The intention is that all operations that are not tied to GUI implementation are packaged as library functions for reuse outside of that application.
- **"Sanity check"** functions (ch. 13) perform operations used by the in-house dataset "sanity checking" utility. These functions encapsulate operations that are not lab-specific.

Relevant notes about data structures and file structures used by the various libraries are described in their associated "Additional Notes" chapters.

Contents

Part I

Core Libraries

Chapter 1

Core Library Structures and Additional Notes

1.1 CHANLIST.txt

```
Channel lists are used for several purposes. They may be used to specify a
set of channels, or they may be used to store per-channel information such
as reference selection or physical source channel in a channel map.



A "project-level channel list" is a structure with one field per folder,
indexed by user-assigned folder label. Each folder's field contains a
"folder-level channel list".



A "folder-level channel list" is a structure with one field per bank, indexed
by bank label. Each bank's field contains a "bank-level channel list".



A "bank-level channel list" is a structure with the following fields:

- "chanlist" is a vector containing integer-valued channel indices.
- "scalarmeta" (optional) is a structure containing metadata that isn't
  stored per-channel.
Other optional fields are vectors or cell arrays with per-channel metadata.

Situation-specific optional fields are described below:

For channel lists specifying data to be read:
- "samprange" (optional) [firstsamp lastsamp] is a two-element vector
  specifying the range of samples to read. Sample index 1 is the first sample.
  If "samprange" isn't given or is empty ([]), all samples are read.

For channel lists derived from metadata:
- "scalarmeta.banktype" is a copy of the signal bank metadata's "banktype"
  field.

For reference selection:
- "reflist" is a cell array containing user-specified labels of references
  to use for each channel, when re-referencing. A label that's an empty
  character array means "don't re-reference".
  (References are typically defined as channel lists stored elsewhere.)

For channel mapping:
- "foldersrc" is a cell array containing the folder labels of the source
  channels for each channel in "chanlist".
- "banksrc" is a cell array containing the bank labels of the source
  channels for each channel in "chanlist".
- "chansrc" is a vector containing channel indices of the source channels
  for each channel in "chanlist".

For per-channel signal processing:
- "resultlist" is a cell array containing the output of the signal processing
  function that was called for each channel (per PROCFUNC.txt).



This is the end of the file.
```

1.2 FOLDERMETA.txt

```
Metadata for data repository folders, ephys devices, signal banks within
ephys devices, and channels within signal banks are stored in a hierarchicial
set of structures. User-defined metadata may also be added to these
structures.



A "project metadata structure" is a structure containing top-level metadata
and metadata for each of the data folders used by the project. Folders
typically correspond to data captured by one specific piece of equipment or
using one specific software tool.

"folders" is a structure with one field per folder, indexed by user-assigned
  folder label, containing folder metadata structures.

User-defined metadata fields may also be added.
```

A "folder metadata structure" is a structure containing folder-level metadata and metadata for each of the signal banks provided by the ephys device or ephys software suite that created the folder. Signal banks may correspond to hardware banks (such as specific headstages or groups of I/O channels) or to virtual banks defined by the ephys software; the metadata representation is the same.

The folder metadata structure has the following fields:

"path" is the filesystem path of this data folder.
"devicetype" is a label assigned by the LoopUtil library identifying the type of device that produced the data (i.e. identifying the helper functions needed to read that data).
"banks" is a structure with one field per signal bank, indexed by signal bank label, containing signal bank metadata structures.
"nativeorder" (optional) is a structure array containing bank and channel numbers in the order in which the underlying device sorted them. These are stored in "bank" and "channel" fields, respectively. This is not guaranteed to contain all types of channel.
"nativemeta" (optional) is an object containing device-specific metadata in its original form. This is typically, but not necessarily, a structure.

Other metadata fields specific to a given device type may also be present. User-defined metadata fields may also be added.

In the event that one physical folder contains data from multiple ephys devices, multiple folder metadata structures are produced. These will have the same "path" value but different folder labels.

A "signal bank metadata structure" is a structure describing multiple channels that are grouped within one real or virtual "bank" by an ephys device or ephys software suite (for example, one headstage's analog recording channels). The following fields are present:

"channels" is a vector containing (integer-valued) channel indices for which data is present. These are not guaranteed to be sorted or to be contiguous (sparse and unsorted channels are okay).
"samprate" is the data sampling rate in samples per second.
"sampcount" is the number of data samples in each channel's recording. This is assumed to be the same for all channels within a bank. For sparse series (event data), event timestamps are assumed to be in the range 1..sampcount.
"banktype" is a label assigned by the LoopUtil library identifying the type of signal contained within this bank. Defined types are:

- 'analog' represents continuous-range time-varying signal data.
- 'integer' represents discrete-range time-varying signal data.
- 'boolean' represents a signal that is either "on" or "off". Changes may be interpreted as events.

```
  - 'flagvector' is integer data that represents multiple 'boolean' signals.
    Changes may be interpreted as events, and flags may be decoded using the
    "flagdefs" structure if present.
  - 'eventwords' is integer data stored as a sparse time series (event data
    rather than continuous data).
  - 'eventbool' is boolean data stored as a sparse time series (event data
    rather than continuous data).
"flagdefs" (optional) is present for banks of type 'flagvector'. It is a
  structure indexed by flag label containing the integer bit-mask values
  that correspond to each flag in the vector.
"nativetimetype" is the Matlab type name of the underlying "native" timestamp
  format. This is typically a sample count stored as a large integer type.
"nativedatatype" is the Matlab type name of the underlying "native" signal
  data.
"nativezerolevel" is the "native" data value corresponding to a signal value
  of zero. This is typically used with unsigned integer native data types.
"nativescale" is a multiplier used to convert "native" data values to
  floating-point data values in suitable units (volts, amperes, etc).
"fpunits" is a human-readable label indicating measurement units after
  conversion to floating point ('uV', 'V', 'uA', 'A', etc). This may be ''
  for data without meaningful units (integer, boolean, flag vector, etc).
"handle" is an opaque object used by the LoopUtil library to manage file
  I/O state. It typically contains file format details, file descriptors for
  open files, and state flags.
"nativemeta" (optional) is an object containing bank-specific metadata in
  the original device-supplied format. This is typically, but not
  necessarily, a structure.

Other metadata fields specific to a given bank type may also be present,
although this is typically stored in the "handle" object instead.
User-defined metadata fields may also be added.



This is the end of the file.
```

1.3 PROCFUNC.txt

```
A processing function handle is called to perform signal processing when
iterating across input channels. The intention is to allow user-defined
signal processing while never having to load more than one input channel
into RAM at any given time.


A processing function has the form:

resultval = procfunc( metadata, folderid, bankid, chanid, ...
  wavedata, timedata, wavenative, timenative )
```

```
"metadata" is a project metadata structure, per FOLDERMETA.txt.
"folderid" is a character array with the user-defined folder label of the
  signal being processed.
"bankid" is a character array with the device-defined bank label of the
  signal being processed.
"chanid" is a scalar containing the channel number of the signal being
  processed.
"wavedata" is a vector containing signal samples being processed. Data values
  are floating-point values (typically volts).
"timedata" is a vector containing timestamps of the signal samples being
  processed. These are floating-point values (typically seconds).
"wavenative" is a vector containing signal samples being processed in "native"
  format (typically 16-bit signed or unsigned integers). Data scale is
  equipment-dependent (per FOLDERMETA.txt).
"timenative" is a vector containing timestamps of the signal samples in
  "native" format (typically 32- or 64-bit signed integer sample counts).

"resultval" is a user-defined object representing the results of processing
  one signal. For top-level processing, this is typically a structure
  containing feature information extracted from the signal waveform. For
  preprocessing, this is typically a filtered copy of the waveform itself.


A typical processing function definition would be as follows. This example
wraps a helper function that is passed additional arguments set at the time
the processing function is defined.

tuning_parameters = (stuff);
other_parameters = (stuff);
procfunc = @( metadata, folderid, bankid, chanid, ...
  wavedata, timedata, wavenative, timenative ) ...
  helper_do_processing( metadata, folderid, bankid, chanid, ...
    wavedata, timedata, tuning_parameters, other_parameters );


This is the end of the file.
```

1.4 ZMODELS.txt

```
An impedance model is a list of category definitions for clustering impedance
values. This is stored as a structure indexed by category label, with each
field containing a structure that defines that category's cluster.


A "box" cluster accepts all data points within a given range of magnitudes
and phase angles.
A "box" cluster definition structure has the following fields:
```

```
"type" is 'box'.
"magrange" [min max] is the range of accepted impedance magnitudes. This is
  typically the logarithm of the actual magnitude.
"phaserange" [min max] is the range of accepted impedance phase angles, in
  radians. A pair of angles defines two arcs - one larger than pi radians,
  and one smaller. The smaller arc is taken to be the accepted range.


An "orthogauss" cluster definition is a bivariate normal distribution with
principal axes parallel to the magnitude and phase axes. The category label
of a sample is the category whose probability density function is highest
for that sample, out to some maximum range (typically 3 sigma).
An "orthogauss" cluster definition structure has the following fields:

"type" is 'orthogauss'.
"magmean" is the magnitude distribution's mean.
"magdev" is the magnitude distribution's standard deviation.
"phasemean" is the phase distribution's circular mean (mean direction).
"phasedev" is the standard deviation of (phase - phase mean). This is
  assumed to be much smaller than 2pi (not needing circular statistics).


FIXME - There's a planned "gauss" model that has proper multivariate
Gaussian distributions with covariance matrices, but impedance data rarely
actually needs that, so it's deferred.



This is the end of the file.
```

Chapter 2

“nlProc” Functions

2.1 nlProc_autoClusterImpedance.m

```
% function zmodels = ...
%   nlProc_autoClusterImpedance( magdata, phasedata, phaseunits )
%
% This attempts to build sensible cluster definitions covering the specified
% impedance magnitude and phase data. Cluster models are described in
% "ZMODELS.txt".
%
% NOTE - Magnitude input is in ohms and phase input may be radians or degrees,
% but model parameters use log10(ohms) and radians exclusively.
%
% "magdata" is a set of impedance magnitude values, in ohms.
% "phasedata" is a corresponding set of impedance phase values.
% "phaseunits" is 'degrees' or 'radians'.
%
% "zmodels" is a structure containing zero or more models of the data, indexed
%   by model type (typically 'box' and 'orthogauss').
%
% A 'box' model is a structure indexed by category label with the following
% fields:
%   "type" is 'box'.
%   "magrange" [min max] is the range of accepted log10(magnitude) values.
%   "phaserange" [min max] is the range of accepted phases in radians.
% A category label is applied to a sample if that sample's magnitude and
% phase are within the specified ranges.
%
% An 'orthogauss' model is a structure indexed by category label with the
% following fields:
%   "type" is 'orthogauss'.
%   "magmean" is the mean of log10(magnitude) for this category.
%   "magdev" is the standard deviation of log10(magnitude) for this category.
%   "phasemean" is the mean direction of phase for this category in radians.
%   "phasedev" is the standard deviation of (phase - mean) for this category.
% The category label of a sample is the category whose probability density
% function is highest for that sample.
```

2.2 nlProc_binTableDataSimple.m

```
% function newtable = ...
%  nlProc_binTableDataSimple( oldtable, bindefs, testorder, newcol )
%
% This applies catagory labels to table data rows by partitioning based on
% values in one or more table columns. Category labels are character arrays.
%
% "oldtable" is the table to apply labels to.
% "bindefs" is a structure indexed by category label. Each field contains
%   a structure array specifying conditions for that category label. A
%   condition structure contains the following fields:
%     "source" is the column to test.
%     "range" [min max] is the range of accepted values.
%     "negate" is true to only accept values _outside_ the range.
%   All conditions must match for the label to be applied. Tests on
%   non-existent source columns automatically fail.
% "testorder" is a cell array of category labels specifying the order in
%   which to test bin definitions. The last successful test determines the
%   label applied. Labels that don’t have bin definitions automatically
%   succeed (this is useful for specifying a default label). Bin definitions
%   not in this list aren’t tested.
% "newcol" is the name of the column to store category labels in.
%
% "newtable" is a copy of "oldtable" with the category label column added.
```

2.3 nlProc_calcSkewPercentile.m

```
% function [ seriesmedian seriesiqr seriesskew rawpercentiles ] = ...
%   nlProc_calcSkewPercentile(dataseries, tailpercent)
%
% This computes the median and the (tailpercent, 100%-tailpercent) tail
% percentiles for the specified series, and evaluates skew by comparing the
% midsummary (average of the tail values) with the median. The result is
% normalized (a skew of +/- 1 is a displacement by +/- the interquartile
% range).
%
% "dataseries" is the sample sequence to process.
% "tailpercent" is an array of tail values to test.
%
% "seriesmedian" is the series median value.
% "seriesiqr" is the series interquartile range.
% "seriesskew" is an array of normalized skew values corresponding to the
%   tail percentages.
% "rawpercentiles" is an array with the actual percentile values used for
%   skew calculations. It contains percentile values corresponding to
%   [ (tailpercent) (median) (100 - tailpercent) (25%) (75%) ].
```

2.4 nlProc_calcSpectrumSkew.m

```
% function [ spectfreqs spectmedian spectiqr spectskew ] = ...
%   nlProc_calcSpectrumSkew( dataseries, samprate, ...
%   freqrange, freqperdecade, wintime, winsteps, tailpercent)
%
% This computes a persistence spectrum for the specified series, and finds
% the median, interquartile range, and the normalized skew for each frequency
% bin. "Skew" is defined per nlProc_calcSkewPercentile().
%
% "dataseries" is the data series to process.
% "samprate" is the sampling rate of the data series.
% "freqrange" [ fmin fmax ] specifies the frequency band to evaluate.
% "freqperdecade" is the number of frequency bins per decade.
% "wintime" is the window duration in seconds to compute the time-windowed
%   Fourier transform with.
% "winsteps" is the number of overlapping steps taken when advancing the time
%   window. The window advances by wintime/winsteps seconds per step.
% "tailpercent" is an array of percentile values that define the tails for
%   skew calculation, per nlProc_calcSkewPercentile().
%
% "spectfreqs" is an array of bin center frequencies.
% "spectmedian" is an array of per-frequency median power values.
% "spectiqr" is an array of per-frequency power interquartile ranges.
% "spectskew" is a cell array, with one cell per "tailpercent" value. Each
%   cell contains an array of per-frequency skew values.
```

2.5 nlProc_computeAverageSignal.m

```
% function avgseries = ...
%   nlProc_computeAverageSignal(metadata, chanlist, memchans, preprocfunc)
%
% This iterates through a list of channels, computing the average signal
% value of all specified channels. The average signal is returned.
%
% NOTE - It is the user's responsibility to ensure that all listed channels
% are time-aligned and have data values that use the same scale. If all
% signals are from the same ephys machine, that's usually handled. Otherwise
% the preprocessing function should handle that.
%
```

```
% "metadata" is the project metadata structure, per FOLDERMETA.txt.
% "chanlist" is a structure listing the channels to be averaged, per
%   CHANLIST.txt.
% "memchans" is the maximum number of channels that may be loaded into
%   memory at the same time.
% "preprocfunc" is a function handle that is called to preprocess each
%   channel's data prior to being averaged, per PROCFUNC.txt. This typically
%   performs artifact removal (filtering happens after re-referencing).
%
% "avgseries" is a data series computed as the average of the input channels.
%   Input series of different lengths are tolerated, but all are assumed to
%   start at the same time and to have the same sampling rate.
```

2.6 nlProc_erodeBooleanCount.m

```
% function newflags = ...
%   nlProc_erodeBooleanCount( oldflags, erodebefore, erodeafter )
%
% This processes a vector of boolean values, eroding "true" flags (extending
% "false" flags) forwards and backwards in time by the specified number of
% samples. Samples up to "erodebefore" at the start of and "erodeafter"
% at the end of sequences of true samples in the original signal are false
% in the returned signal.
%
% Erosion is implemented as dilation of the complement vector with "before"
% and "after" values swapped.
%
% "oldflags" is the boolean vector to process.
% "erodebefore" is the number of samples at the start of a sequence to squash.
% "erodeafter" is the number of samples at the end of a sequence to squash.
%
% "newflags" is the boolean vector with erosion performed.
```

2.7 nlProc_erodeBooleanTime.m

```
% function newflags = ...
%   nlProc_erodeBooleanTime( oldflags, samprate, erodebefore, erodeafter )
%
% This processes a vector of boolean values, eroding "true" flags (extending
% "false" flags) forwards and backwards in time by the specified durations.
% Samples up to "erodebefore" at the start of and "erodeafter" at the end of
% sequences of true samples in the original signal are false in the returned
% signal.
%
% Erosion is implemented as dilation of the complement vector with "before"
% and "after" values swapped.
```

```
%
% "oldflags" is the boolean vector to process.
% "samprate" is the sampling rate of the flag vector.
% "erodebefore" is the duration in seconds at the start of a sequence to
%   squash.
% "erodeafter" is the duration in seconds at the end of a sequence to squash.
%
% "newflags" is the boolean vector with erosion performed.
```

2.8 nlProc_examineLFPSpectrum.m

```
% function [ isgood typelabel fitexponent ...
%   spectpowers spectfreqs fitpowers ] = nlProc_examineLFPSpectrum( ...
%     wavedata, samprate, freqrange, binwidth )
%
% This takes the power spectrum of the specified signal, bins it in the
% log-frequency domain, rejects narrow peaks, and then tries to fit a power
% law curve to it. If this looks like pink noise or red noise, it's a valid
% LFP; if this looks like white noise or like hash, it isn't.
%
% FIXME - In a perfect world we'd evaluate "burstiness" and give different
% labels based on whether a valid LFP was quiet or bursty. That's NYI.
%
% "wavedata" is a vector containing waveform sample data.
% "samprate" is the sampling rate of the waveform data.
% "freqrange" [ min max ] is the range of frequencies to fit over.
% "binwidth" is the relative width of frequency bins. A value of 0.1 would
%   mean a bin width of 2 Hz for a bin with a center frequency of 20 Hz.
%
% "isgood" is true if the spectrum looks like LFP background, false otherwise.
% "typelabel" is a human-readable descriptive label. Typical values are
%   'lfp', 'lfpbad', 'whitenoise', 'powerlaw', and 'hash'.
% "fitexponent" is the exponent of the power-law fit. Pink noise is -1, red
%   noise is -2.
% "spectpowers" are the binned power spectrum powers (linear, not dB).
% "spectfreqs" are the spectrum bin center frequencies in Hz.
% "fitpowers" are the curve-fit power law powers at the bin frequencies.
```

2.9 nlProc_fillNaN.m

```
% function newseries = nlProc_fillNaN( oldseries )
%
% This interpolates NaN segments within the series using linear interpolation,
% and then fills in NaNs at the end of the series by replicating samples.
% This makes the derivative discontinuous when filling endpoints but prevents
% large excursions from curve fit extrapolation.
```

```
%
% "oldseries" is the series containing NaN segments.
%
% "newseries" is the interpolated series without NaN segments.
```

2.10 nlProc_filterBrickBandStop.m

```
% function newwave = nlProc_filterBrickBandStop( oldwave, samprate, bandlist )
%
% 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).
%
% "oldwave" is the signal to filter. This is assumed to be real.
% "samprate" is the sampling rate of "oldwave".
% "bandlist" is a cell array containing [ min max ] tuples indicating
%   frequency ranges to squash.
%
% "newwave" is a filtered version of "oldwave".
```

2.11 nlProc_filterSignal.m

```
% function [ lfpseries spikeseries ] = nlProc_filterSignal( oldseries, ...
%   samprate, lfprate, lowpassfreq, powerfreq, dcfreq )
%
% This applies several filters:
% - A DC removal filter.
% - A notch filter to remove power line noise.
% - A low-pass filter to isolate the local field potential signal.
% The full-rate LFP series is subtracted from the original series to produce
% a high-pass-filtered "spike" series, and a downsampled LFP series is
% also returned.
%
% "oldseries" is the original wideband signal.
% "samprate" is the sampling rate of the wideband signal.
% "lfprate" is the desired sampling rate of the LFP signal. This should
%   cleanly divide "samprate" (samprate = k * lfprate for some integer k).
% "lowpassfreq" is the edge of the pass-band for the LFP signal. This is
%   lower than the filter's corner frequency; it's the 0.2 dB frequency.
% "powerfreq" is an array of values specifying the center frequencies of the
%   power line notch filter. This is typically 60 Hz or 50 Hz (a single
%   value), but may contain multiple values to filter harmonics. An empty
%   array disables this filter.
% "dcfreq" is the edge of the pass-band for the high-pass DC removal filter.
%   Set to 0 to disable this filter. This is higher than the corner frequency;
%   it's the 0.2 dB frequency (ripple is flat above it).
%
% "lfpseries" is the downsampled low-pass-filtered signal.
% "spikeseries" is the full-rate high-pass-filtered signal.
%
% Filters used are IIR, called with "filtfilt" to remove time offset by
% running the filter forwards and backwards in time. The power line filter
% takes about 1/2 second to fully stabilize, and the low-pass LFP filter takes
% about 1/2 period to 1 period to fully stabilize. Edge effects may occur
% within this distance of the start and end of the signal.
% The DC rejection filter also takes at least 1 period to stabilize. Since
% it's applied in both directions, and won't perturb the pass-band, it should
% be well-behaved over the entire signal.
%
% The LFP sampling rate should be at least 10 times "lowpassfreq" to avoid
% aliasing during downsampling. The DC rejection filter pass frequency
% should be no lower than half the lowest frequency of interest, to
% minimize edge effects.
```

2.12 nlProc_findCorrelatedChannels.m

```
% function [ changroups rvalues groupdefs ] = ...
%   nlProc_findCorrelatedChannels( wavedata, thresh_abs, thresh_rel )
%
% This attempts to find sets of strongly-correlated channels in waveform data.
% These may be floating channels coupling identical noise (for high-frequency
% data) or may be measuring from the same environment (for LFP data).
%
% NOTE - Channel correlation time goes up as the square of the number of
% channels!
%
% Correlation is judged using Pearson's Correlation Coefficient.
%
% "wavedata" is an Nchans*Nsamples matrix containing waveform data.
% "thresh_abs" is an absolute threshold. Channel pairs with correlation
%   coefficients above +thresh_abs are assumed to be copies.
%   NOTE - Differential channel pairs with have coefficients below -thresh_abs;
%   this is okay, and gets taken into account for thresh_rel per below.
% "thresh_rel" is a relative threshold. Channel pairs with correlation
%   coefficients above this multiple of a "typical" correlation coefficient
%   value are assumed to be copies. The "typical" value is the median of the
%   absolute value of all correlation coefficients that are below +thresh_abs
%   and above -thresh_abs.
%
% "changroups" is a vector indicating which group each channel is a member of,
%   or NaN if a channel is not a member of a group (not strongly correlated).
% "rvalues" is an Nchans*Nchans matrix containing correlation coefficient
%   values for all channel pairs.
% "groupdefs" is a cell array containing vectors representing groups of
%   mutually correlated channels. Each vector contains channel indices for
%   the members of that group.
```

2.13 nlProc_findSpectrumPeaks.m

```
% function [ peakfreqs peakheights peakwidths binlevels bincenters ] = ...
%   nlProc_findSpectrumPeaks( ...
%     wavedata, samprate, peakwidth, backgroundwidth, peakthresh )
%
% This takes the frequency spectrum of the specified signal, bins it in the
% log-frequency domain, and looks for narrow peaks against the background.
%
% "wavedata" is a vector containing waveform sample data.
% "samprate" is the sampling rate of the waveform data.
% "peakwidth" is the relative width of the fine-resolution frequency bins.
%   A value of 0.1 would mean a bin width of 2 Hz at a frequency of 20 Hz.
% "backgroundwidth" is the ratio between the upper and lower frequencies of
%   the span used to evaluate noise background around any given bin. A value
%   of 1.0 would mean evaluating noise over a one-octave span.
% "peakthresh" is the magnitude threshold for recognizing a peak in the
%   frequency spectrum. This is a multiple of the average local background.
%
% "peakfreqs" is a vector containing peak center frequencies in Hz.
% "peakheights" is a vector containing peak heights relative to the background.
% "peakwidths" is a vector containing relative peak widths (FWHM / frequency).
% "binmags" is a vector containing frequency spectrum bin magnitudes.
% "binfreqs" is a vector containing frequency spectrum bin center frequencies.
```

2.14 nlProc_impedanceCalcDistanceToOrthoGauss.m

```
% function sampdistances = nlProc_impedanceCalcDistanceToOrthoGauss( ...
%   sampmags, sampphases, magmean, magdev, phasemean, phasedev )
%
% Given a set of impedance measurements (or other magnitude/phase data
% points), this computes the distance between each measurement and the
% center of a normal distribution with principal axes parallel to the
% magnitude and phase axes.
%
% Distance is expressed in standard deviations.
%
% "sampmags" is a series of impedance magnitude measurements (typically the
%   logarithm of the actual magnitude).
% "sampphases" is a series of impedance phase measurements, in radians.
% "magmean" is the magnitude distribution's mean.
% "magdev" is the magnitude distribution's standard deviation.
% "phasemean" is the phase distribution's circular mean.
% "phasedev" is the standard deviation of (phase - phase mean). This is
%   assumed to be much smaller than 2pi.
%
% "sampdistances" is a series of scalar values indicating how many standard
%   deviations away from the mean each measurement is. This is the L2 norm
%   of the distances from the magnitude and phase means.
```

2.15 nlProc_impedanceClassifyBox.m

```
% function labels = nlProc_impedanceClassifyBox( ...
%   magdata, phasedata, classdefs, testorder, defaultlabel )
%
% This tests a series of impedance values, applying cluster labels as
% defined by the supplied class definitions. Samples that can't be clustered
% are given a default cluster label.
%
% This function tests against "box" models, as defined in "ZMODELS.txt".
%
% "magdata" is a set of impedance magnitude values. This is typically
%   the logarithm of the actual magnitude.
% "phasedata" is a set of impedance phase values, in radians.
% "classdefs" is a structure indexed by category label, with each field
%   containing a structure that defines the category's cluter, per
%   "ZMODELS.txt". Only "box" definitions are processed by this function.
% "testorder" is a cell array containing category labels, defining the order
%   in which to test for category membership (to disambiguate overlapping
%   categories). The _last_ matching test determines the category label. If
%   this is an empty cell array, an arbitrary order is chosen.
% "defaultlabel" is a character array to be applied as a category label for
%   data points that do not match any cluster definition.
%
% "labels" is a cell array containing cluster labels for each data point.
```

2.16 nlProc_impedanceClassifyOrthoGauss.m

```
% function labels = nlProc_impedanceClassifyOrthoGauss( ...
%   magdata, phasedata, classdefs, maxdistance, defaultlabel )
%
% This tests a series of impedance values, applying cluster labels as
% defined by the supplied class definitions. Samples that can't be clustered
% are given a default cluster label.
%
% This function tests against "orthogauss" models, as defined in
% "ZMODELS.txt".
%
```

```
% "magdata" is a set of impedance magnitude values. This is typically
%   the logarithm of the actual magnitude.
% "phasedata" is a set of impedance phase values, in radians.
% "classdefs" is a structure indexed by category label, with each field
%   containing a structure that defines the category's cluter, per
%   "ZMODELS.txt". Only "orthogauss" definitions are processed by this
%   function.
% "maxdistance" is the maximum distance from a cluster center (in standard
%   deviations) that a sample can have while being a member of that cluster.
% "defaultlabel" is a character array to be applied as a category label for
%   data points that do not match any cluster definition.
%
% "labels" is a cell array containing cluster labels for each data point.
```

2.17 nlProc_impedanceFitOrthoGauss.m

```
% function [ magmean, magdev, phasemean, phasedev ] = ...
%   nlProc_impedanceFitOrthoGauss(membermags, memberphases)
%
% Given a set of impedance measurements (or other magnitude/phase data
% points), this independently estimates mean and deviation for impedance
% magnitude and phase angle.
%
% Magnitude uses the arithmetic mean. Phase uses the circular mean, but
% linear deviation (we're assuming deviation is much smaller than 2pi).
%
% "membermags" is a series of magnitude measurements. These are evaluated
%   on a linear scale; they're typically the logarithm of actual magnitude.
% "memberphases" is a series of phase measurements, in radians.
%
% "magmean" is the arithmetic mean of "membermags".
% "magdev" is the standard deviation of "membermags".
% "phasemean" is the circular mean of "memberphases".
% "phasedev" is the standard deviation of (memberphases - phasemean).
```

2.18 nlProc_interpolateSeries.m

```
% function newdata = nlProc_interpolateSeries( oldtimes, olddata, newtimes )
%
% This performs linear interpolation of a sparsely-defined data series to
% a new set of sparse sample points, handling unusual cases (empty lists,
% NaN entries, etc).
%
% NOTE - To interpolate values off the ends of the source list, the first
% and last points in the source list are used to define a ramp. This gives
% a degraded fit that should still be fairly accurate near the endpoints.
```

```
%
% "oldtimes" is a list of time values for which the data series has known
%   values.
% "olddata" is a list of known data values for these times.
% "newtimes" is a list of time values to produce interpolated data values
%   for.
%
% "newdata" is a list of interpolated data values at the requested times.
```

2.19 nlProc_padBooleanCount.m

```
% function newflags = nlProc_padBooleanCount( oldflags, padbefore, padafter )
%
% This processes a vector of boolean values, extending "true" flags
% forwards and backwards in time by the specified number of samples. Samples
% up to "padbefore" ahead of and "padafter" following true samples in the
% original signal are true in the returned signal.
%
% This is a dilation operation. To perform erosion, perform dilation on the
% complement of a vector (i.e.  newflags = ~ padBooleanCount( ~ oldflags )).
% Remember to swap "before" and "after" for the complement vector.
%
% "oldflags" is the boolean vector to process.
% "padbefore" is the number of samples backwards in time to pad.
% "padafter" is the number of samples forwards in time to pad.
%
% "newflags" is the boolean vector with padding performed.
```

2.20 nlProc_padBooleanTime.m

```
% function newflags = ...
%   nlProc_padBooleanTime( oldflags, samprate, padbefore, padafter )
%
% This processes a vector of boolean values, extending "true" flags
% forwards and backwards in time by the specified durations. Samples
% up to "padbefore" ahead of and "padafter" following true samples in the
% original signal are true in the returned signal.
%
% This is a dilation operation. To perform erosion, perform dilation on the
% complement of a vector (i.e.  newflags = ~ padBooleanTime( ~ oldflags )).
% Remember to swap "before" and "after" for the complement vector.
%
% "oldflags" is the boolean vector to process.
% "samprate" is the sampling rate of the flag vector.
% "padbefore" is the duration in seconds backwards in time to pad.
% "padafter" is the duration in seconds forwards in time to pad.
```

```
%
% "newflags" is the boolean vector with padding performed.
```

2.21 nlProc_removeArtifactsSigma.m

```
% function newseries = nlProc_removeArtifactsSigma( oldseries, ...
%   ampthresh, derivthresh, ampthreshfall, derivthreshfall, ...
%   trimbefore, trimafter, smoothsamps, dcsamps )
%
% This identifies artifacts as excursions in the signal's amplitude or
% derivative, and replaces affected regions with NaN. Excursion thresholds
% are expressed in terms of the standard deviation of the signal or its
% derivative.
%
% "oldseries" is the series to process.
% "ampthresh" is the threshold for flagging amplitude excursion artifacts.
% "derivthresh" is the threshold for flagging derivative excursion artifacts.
% "ampthreshfall" is the turn-off threshold for amplitude artifacts.
% "derivthreshfall" is the turn-off threshold for derivative artifacts.
% "trimbefore" is the number of samples to squash ahead of the artifact.
% "trimafter" is the number of samples to squash after the artifact.
% "smoothsamps" is the size of the smoothing window to apply before taking
%   the derivative, or 0 for no smoothing.
% "dcsamps" is the size of the window for computing local DC average removal
%   ahead of computing signal statistics.
%
% Regions where the amplitude or derivative exceeds the threshold are flagged
% as artifacts. These regions are widened to encompass the region where the
% amplitude or derivative exceeds the "fall" threshold, and then padded by
% the specified number of samples. This is intended to correctly handle
% square-pulse artifacts and fast-step-slow-decay artifacts.
```

2.22 nlProc_removeTimeRanges.m

```
% function newseries = ...
%   nlProc_removeTimeRanges( oldseries, samprate, trimtimes )
%
% This NaNs out specified regions of the input signal.
%
% "oldseries" is the series to process.
% "samprate" is the sampling rate of the input signal.
% "trimtimes" is a cell array containing time spans to NaN out. Time spans
%   have the form "[ time1 time2 ]", where times are in seconds. Negative
%   times are relative to the end of the signal, positive times are relative
%   to the start of the signal (both start at 0 seconds). Use a very small
%   negative value for "-0".
```

```
%
% "newseries" is a modified version of the input series with the specified
%   time ranges set to NaN.
```

2.23 nlProc_rollAndPadCount.m

```
% function newseries = ...
%   nlProc_rollAndPadCount( oldseries, rollsamps, padsamps )
%
% This performs DC and ramp removal, applies a Tukey (cosine) roll-off
% window, and pads the endpoints of the supplied signal.
%
% "oldseries" is the series to process.
% "rollsamps" is the length in samples of the starting and ending roll-offs.
% "padsamps" is the number of starting and ending padding samples to add.
%
% "newseries" is the processed signal.
```

2.24 nlProc_rollAndPadTime.m

```
% function newseries = ...
%   nlProc_rollAndPadTime( oldseries, samprate, rolltime, padtime )
%
% This performs DC and ramp removal, applies a Tukey (cosine) roll-off
% window, and pads the endpoints of the supplied signal.
%
% "oldseries" is the series to process.
% "samprate" is the sampling rate of the input signal.
% "rolltime" is the duration in seconds of the starting and ending roll-offs.
% "padtime" is the duration in seconds of starting and ending padding.
%
% "newseries" is the processed signal.
```

2.25 nlProc_trimEndpointsCount.m

```
% function newseries = ...
%  nlProc_trimEndpointsCount( oldseries, trimstart, trimend )
%
% This crops the specified number of samples from the start and end of the
% supplied signal.
%
% "oldseries" is the series to process.
% "trimstart" is the number of samples to remove from the beginning.
% "trimend" is the number of samples to remove from the end.
%
% "newseries" is a truncated version of the input signal.
```

2.26 nlProc_trimEndpointsTime.m

```
% function newseries = ...
%   nlProc_trimEndpointsTime( oldseries, samprate, trimstart, trimend )
%
% This crops the specified durations from the start and end of the supplied
% signal.
%
% "oldseries" is the series to process.
% "samprate" is the sampling rate of the input signal.
% "trimstart" is the number of seconds to remove from the beginning.
% "trimend" is the number of seconds to remove from the end.
%
% "newseries" is a truncated version of the input signal.
```

Chapter 3

“nlUtil” Functions

3.1 nlUtil_continuousToSparse.m

```
% function [ eventvals, eventtimes ] = ...
%   nlUtil_continuousToSparse( wavedata, timedata )
%
% This converts a continuous discrete-valued waveform into a series of
% nonuniformly-sampled events (representing changes in the waveform’s value).
%
% The waveform value is assumed to be discrete and changes are assumed to be
% infrequent. This will still work if given floating-point data or data that
% changes at every sample, but there’s little point in representing these
% signals as event lists ("eventvals" and "eventtimes" will be copies of
% "wavedata" and "timedata", respectively).
%
% NOTE - There will always be an event at the first timestamp representing
% the initial data value.
%
% "wavedata" is a vector with samples from the continuous signal.
% "timedata" is a vector of timestamps corresponding to these samples.
%
% "eventvals" is a vector containing signal values immediately after changes.
% "eventtimes" is a vector containing the timestamps of these changes.
```

3.2 nlUtil_forceColumn.m

```
% function newseries = nlUtil_forceColumn(oldseries)
%
% This function forces a one-dimensional vector into Nx1 form.
%
% "oldseries" is a 1xN or Nx1 vector.
%
```

```
% "newseries" is the corresponding Nx1 vector.
```

3.3 nlUtil_forceRow.m

```
% function newseries = nlUtil_forceRow(oldseries)
%
% This function forces a one-dimensional vector into 1xN form.
%
% "oldseries" is a 1xN or Nx1 vector.
%
% "newseries" is the corresponding 1xN vector.
```

3.4 nlUtil_sparseToContinuous.m

```
% function wavedata = ...
%   nlUtil_sparseToContinuous( eventtimes, eventvalues, samprange )
%
% This converts a sequence of nonuniformly-sampled events into a continuous
% waveform. Samples are assumed to reflect the first instances of changed
% waveform values; this signal is held constant at the last seen sample value.
%
% Events are sorted prior to processing.
%
% "eventtimes" are the timestamps (sample indices) associated with each event.
% "eventvalues" are the data values associated with each event.
% "samprange" [min max] is the span of sample indices to generate data for.
%
% "wavedata" is a waveform spanning the specified range of sample indices,
%   where the value of any given sample is the value of the most recently-seen
%   event, or zero if there were no prior events.
```

3.5 nlUtil_structToTable.m

```
% function outdata = nlUtil_structToTable(indata, ignorelist)
%
% This converts a structure into a table. Table columns correspond to
% structure fields. Structure vectors forced to Nx1 (column) format.
% Specified structure fields may be skipped.
% NOTE - Data fields must all have the same length!
%
% "indata" is the structure to convert.
% "ignorelist" is a cell array containing structure field names to skip.
%
```

```
% "outdata" is a table with columns containing structure field data.
```

3.6 nlUtil_tableToStruct.m

```
% function outdata = nlUtil_tableToStruct(indata, rowcol)
%
% This converts a table into a structure. Structure fields correspond to
% table columns, and are stored as either 1xN or Nx1 vectors.
%
% "indata" is the table to convert.
% "rowcol" is 'row' to output 1xN vectors and 'col' for Nx1 vectors.
%
% "outdata" is a structure containing table column data.
```

Chapter 4

“nlPlot” Functions

4.1 nlPlot_axesPlotExcursions.m

```
% function nlPlot_axesPlotExcursions( thisax, ...
%   spectfreqs, spectmedian, spectiqr, spectskew, percentlist, ...
%   want_relative, figtitle )
%
% This plots LFP power excursions, either as relative power excess alone or
% against the median power spectrum. See nlChan_applySpectSkewCalc() for
% details of skew calculation and array contents.
% The plot is rendered to the specifed set of figure axes.
%
% "thisax" is the "axes" object to render to.
% "spectfreqs" is an array of frequency bin center frequencies.
% "spectmedian" is an array of per-frequency median power values.
% "spectiqr" is an array of per-frequency power interquartile ranges.
% "spectskew" is a cell array, with one cell per "percentlist" value. Each
%   cell contains an array of per-frequency skew values.
% "percentlist" is an array of percentile values that define the tails for
%   skew calculations, per nlProc_calcSkewPercentile().
% "want_relative" is true to plot relative power excess alone, and false to
%   plot against the median power spectrum.
% "figtitle" is the title to use for the figure, or '' for no title.
```

4.2 nlPlot_axesPlotPersist.m

```
% function nlPlot_plotPersist( thisax, ...
%   persistvals, persistfreqs, persistpowers, want_log, figtitle )
%
% This plots a pre-tabulated persistence spectrum. See "pspectrum()" for
% details of input array structure.
%
```

```
% "thisax" is the "axes" object to render to.
% "thisfig" is the figure to render to (this may be a UI component).
% "persistvals" is the matrix of persistence spectrum fraction values.
% "persistfreqs" is the list of frequencies used for binning.
% "persistpowers" is the list of power magnitudes used for binning.
% "want_log" is true if the frequency axis should be plotted on a log scale
%   (it's computed on a linear scale).
% "figtitle" is the title to use for the figure, or '' for no title.
```

4.3 nlPlot_axesPlotSpikeHist.m

```
% function nlPlot_plotSpikeHist( thisax, ...
%   bincounts, binedges, percentamps, percentpers, figtitle )
%
% This plots a pre-tabulated histogram of normalized spike waveform
% amplitude. For channels with real spikes, tails are asymmetrical.
%
% "thisax" is the "axes" object to render to.
% "bincounts" is an array containing bin count values, per histogram().
% "binedges" is an array containing the histogram bin edges, per histogram().
% "percentamps" is an array of normalized amplitudes corresponding to desired
%   tail percentiles to highlight. Entries 1..N are tail percentile amplitudes,
%   entry N+1 is the median, and entries N+2..2N+1 are (100%-tail) amplitudes.
% "percentpers" is an array naming desired tail percentiles to highlight.
% "figtitle" is the title to use for the figure, or '' for no title.
```

4.4 nlPlot_getColorLUTPeriodic.m

```
% function collut = nlPlot_getColorLUTPeriodic()
%
% This function returns a cell array of color triplets. Colors are chosen
% so that successive colors are similar and so that the list may be iterated
% through repeatedly.
%
% "collut" is a cell array containing color triplets.
```

4.5 nlPlot_getColorPalette.m

```
% function cols = nlPlot_getColorPalette()
%
% This function returns a structure containing color triplets indexed by
% color name.
%
```

```
% Colors suppled are "blu", "brn", "yel", "mag", "grn", "cyn", and "red".
% These are mostly cribbed from get(cga,'colororder'), with tweaks.
%
% "cols" is a structure containing color triplets.
```

4.6 nlPlot_getColorSpread.m

```
% function colorlist = nlPlot_getColorSpread(origcol, count, anglespan)
%
% This function takes a starting color and turns it into a spectrum of
% nearby colors, by walking around the color wheel starting with the original
% color.
%
% The resulting sequence is returned as a cell array of color vectors.
%
% "origcol" [ r g b ] is the starting color.
% "count" is the number of colors to return.
% "anglespan" (degrees) is the distance to walk along the color wheel.
%
% "colorlist" is a cell array of the resulting [r g b] color vectors.
```

4.7 nlPlot_plotExcursions.m

```
% function nlPlot_plotExcursions( thisfig, oname, ...
%   spectfreqs, spectmedian, spectiqr, spectskew, percentlist, ...
%   want_relative, figtitle )
%
% This plots LFP power excursions, either as relative power excess alone or
% against the median power spectrum. See nlChan_applySpectSkewCalc() for
% details of skew calculation and array contents.
%
% "thisfig" is the figure to render to (this may be a UI component).
% "oname" is the filename to save to, or '' to not save.
% "spectfreqs" is an array of frequency bin center frequencies.
% "spectmedian" is an array of per-frequency median power values.
% "spectiqr" is an array of per-frequency power interquartile ranges.
% "spectskew" is a cell array, with one cell per "percentlist" value. Each
%   cell contains an array of per-frequency skew values.
% "percentlist" is an array of percentile values that define the tails for
%   skew calculations, per nlProc_calcSkewPercentile().
% "want_relative" is true to plot relative power excess alone, and false to
%   plot against the median power spectrum.
% "figtitle" is the title to use for the figure, or '' for no title.
```

4.8 nlPlot_plotPersist.m

```
% function nlPlot_plotPersist( thisfig, oname, ...
%   persistvals, persistfreqs, persistpowers, want_log, figtitle )
%
% This plots a pre-tabulated persistence spectrum. See "pspectrum()" for
% details of input array structure.
%
% "thisfig" is the figure to render to (this may be a UI component).
% "oname" is the filename to save to, or '' to not save.
% "persistvals" is the matrix of persistence spectrum fraction values.
% "persistfreqs" is the list of frequencies used for binning.
% "persistpowers" is the list of power magnitudes used for binning.
% "want_log" is true if the frequency axis should be plotted on a log scale
%   (it's computed on a linear scale).
% "figtitle" is the title to use for the figure, or '' for no title.
```

4.9 nlPlot_plotSpikeHist.m

```
% function nlPlot_plotSpikeHist( thisfig, oname, ...
%   bincounts, binedges, percentamps, percentpers, figtitle )
%
% This plots a pre-tabulated histogram of normalized spike waveform
% amplitude. For channels with real spikes, tails are asymmetrical.
%
% "thisfig" is the figure to render to (this may be a UI component).
% "oname" is the filename to save to, or '' to not save.
% "bincounts" is an array containing bin count values, per histogram().
% "binedges" is an array containing the histogram bin edges, per histogram().
% "percentamps" is an array of normalized amplitudes corresponding to desired
%   tail percentiles to highlight. Entries 1..N are tail percentile amplitudes,
%   entry N+1 is the median, and entries N+2..2N+1 are (100%-tail) amplitudes.
% "percentpers" is an array naming desired tail percentiles to highlight.
% "figtitle" is the title to use for the figure, or '' for no title.
```

Chapter 5

“nlIO” Functions

5.1 nlIO_filterChanList.m

```
% function newlist = nlIO_filterChanList( oldlist, filterrules )
%
% This function filters a channel list, keeping or discarding entries
% according to user-specified filter rules.
%
% "oldlist" is a channel list to filter, per "CHANLIST.txt".
% "filterrules" is a structure with zero or more of the following fields:
%   "keepfolders" is a cell array containing regex patterns that folder labels
%     must match.
%   "omitfolders" is a cell array containing regex patterns that folder labels
%     must not match.
%   "keepbanks" is a cell array containing regex patterns that bank labels
%     must match.
%   "omitbanks" is a cell array containing regex patterns that bank labels
%     must not match.
%   "keepchans" is a vector containing channel indices that should be kept.
%     Any channels not in this list are discarded.
%   "omitchans" is a vector containing channel indices that must be discarded.
%   "keepbanktypes" is a cell array containing bank type identifiers that
%     should be matched. Any bank type identifiers not in this list are
%     discarded.
%   "omitbanktypes" is a cell array containing bank type identifiers that
%     must not be matched.
%
% "newlist" is a subset of "oldlist" containing entries that pass all rules.
```

5.2 nlIO_formatMemberList.m

```
% function memberstring = ...
%   nlIO_formatMemberList( candidates, format, memberflags )
%
% This formats a list of matching candidates in human-readable form, in a
% manner similar to page numbering (e.g. "5-6, 7, 12, 13-15"). Member
% entries that are contiguous in the candidate list are reported as ranges
% rather than as individuals.
%
% The candidate list is assumed to already be sorted in a sensible order.
%
% "candidates" is a vector or cell array containing member IDs or labels.
% "format" is a "sprintf" conversion format for turning a candidate ID or
%   label into appropriate human-readable output.
% "memberflags" is a logical vector of the same size as "candidates" that is
%   "true" for candidates that are to be reported and "false" otherwise.
%
% "memberstring" is a human-readable string summarizing the list of
%   candidates for which "memberflags" is true.
```

5.3 nlIO_formatTableData.m

```
% function newtable = nlIO_formatTableData( oldtable, formatlut )
%
% This function converts the specified data columns in oldtable into strings,
% using "sprintf" with the format specified for that column's entry in the
% lookup table.
%
% "oldtable" is the table to convert.
% "formatlut" is a "containers.Map" object mapping table column names to
%   sprintf format character arrays.
%
% "newtable" is a copy of "oldtable" with the specified columns converted.
```

5.4 nlIO_getChanListFromMetadata.m

```
% function chanlist = nlIO_getChanListFromMetadata(metadata)
%
% This generates a channel list describing all channels defined in the
% specified metadata structure. The "banktype" field is copied to the
% channel list scalar metadata structure to facilitate list filtering.
%
% "metadata" is a project metadata structure, per "FOLDERMETA.txt".
%
% "chanlist" is a channel list, per "CHANLIST.txt".
```

5.5 nlIO_iterateChannels.m

```
% function resultvals = ....
%   nlIO_iterateChannels(metadata, chanlist, memchans, procfunc)
%
% This iterates through a set of channels, loading each channel's waveform
% data in sequence and calling a processing function with that data.
% Processing output is aggregated and returned.
%
% This is implemented such that only a few channels are loaded at a time.
%
% Channel time series are stored as sample numbers (not times). Analog data
% is converted to microvolts. TTL data is converted to boolean.
%
% "metadata" is a project metadata structure, per FOLDERMETA.txt.
% "chanlist" is a structure listing channels to process, per CHANLIST.txt.
% "memchans" is the maximum number of channels that may be loaded into
%   memory at the same time.
% "procfunc" is a function handle used to transform channel waveform data
%   into "result" data, per PROCFUNC.txt.
%
% "resultvals" is a channel list structure that has bank-level channel lists
%   augmented with a "resultlist" field, per CHANLIST.txt. The "resultlist"
%   field is a cell array containing per-channel output from "procfunc".
```

5.6 nlIO_quoteTableStrings.m

```
% function newtable = nlIO_quoteTableStrings(oldtable)
%
% This function returns a copy of the input table with all string cell values
% and all column names in quotes.
%
% This is a workaround for an issue with "writetable". If "writetable" is
% called to write CSV with 'QuoteStrings' set to true, only cell values are
% quoted, not column names. If column names are manually quoted, they get
% triple quotes. The solution is to set 'QuoteStrings' false and manually
% quote all strings and all column names, which this function does.
```

5.7 nlIO_readAndBinImpedance.m

```
% function ztable = nlIO_readAndBinImpedance( fnamelist, ...
%   chancolumn, magcolumn, phasecolumn, phaseunits, bindefs, testorder )
%
% This reads one or more CSV files containing channel impedance measurements.
% Impedance measurements are averaged across files, and channels are tagged
% with type labels based on user-specified criteria (typical types are
% high-impedance, low-impedance, grounded, and floating).
%
% For automated clustering, supply an empty cell array for "testorder"
% (the contents of "bindefs" are ignored in this situation).
%
% When multiple measurements for a given channel ID label are present,
% the magnitude is averaged using the geometric mean (to tolerate large
% differences in magnitude) and the phase angle is averaged using
% circular statistics (mean direction).
%
% NOTE - Phase units are not modified, but we need to know what the units
% are in order to compute the mean direction when averaging phase samples.
%
% NOTE - Phase is wrapped to +/- 180 deg (+/- pi radians).
%
% NOTE - This needs Matlab R2019b or later for 'PreserveVariableNames'.
% It'll still work in older versions but will alter column names to be
% Matlab-safe (use matlab.lang.makeValidName() to duplicate this).
%
% "fnamelist" is a cell array containing the names of files to read. These
%   are expected to be CSV files.
% "chancolumn" is the table column to read channel ID labels from.
% "magcolumn" is the table column to read impedance magnitude from.
% "phasecolumn" is the table column to read impedance phase from.
% "phaseunits" is 'degrees' or 'radians'.
% "bindefs" is a category definition structure per "nlProc_binTableDataSimple".
% "testorder" is the order in which to test category definitions. The first
%   label is the default label, and the _last_ label with a successful test
%   is applied. If this is empty, it forces automatic cluster detection.
%
% "ztable" is a table containing the following columns:
%   "label" is a copy of the "chancolumn" input column.
%   "magnitude" is a copy of the "magcolumn" input column.
%   "phase" is a copy of the "phasecolumn" input column.
%   "type" is the category label for each channel.
```

5.8 nlIO_readBinaryFile.m

```
% function [is_ok sampdata] = nlIO_readBinaryFile(fname, dtype, samprange)
%
% This attempts to read a packed array of the specified data type from the
% specified file.
%
% NOTE - The data is returned as-is, _not_ promoted to double.
%
% "fname" is the name of the file to read from.
% "dtype" is a string identifying the Matlab data type (e.g. 'uint32').
% "samprange" [first last] is the range of data samples (not bytes) to read,
%   specified with Matlab's conventions (the first sample is sample 1, not 0).
%   Specify an empty sample range ([]) to read all data in the file.
%
% "is_ok" is set to true if the operation succeeds and false otherwise.
% "sampdata" is an array containing the sample values, in native format.
```

5.9 nlIO_readFolderMetadata.m

```
% function [ isok newmeta ] = ...
%   nlIO_readFolderMetadata( oldmeta, newlabel, indir, devicetype )
%
% This probes the specified directory, looking for data and metadata files
% from the specified type of ephys machine or software suite. If the type
% is given as 'auto', this searches for types that it knows how to identify.
%
% Metadata structure format is defined in "FOLDERMETA.txt".
%
% "oldmeta" is a project metadata structure to add to. Pass an empty structure
%   to create a new project metadata structure.
% "newlabel" is used as a folder label for adding this folder's metadata to
%   the project metadata structure.
% "indir" is the directory to search.
% "devicetype" is a character array specifying the type of architecture to
%   look for. Known types are 'intan' and 'openephys'. Use 'auto' for
%   automatic detection.
%
% "isok" is true if folder metadata was successfully read and false otherwise.
% "newmeta" is a copy of "oldmeta" with new folder metadata added. If no new
%   metadata was found, a copy of "oldmeta" is still returned. If metadata
%   from multiple devices is found, multiple folder metadata structures are
%   added. "makeUniqueStrings" is called to avoid folder label conflicts.
```

5.10 nlIO_searchForDir.m

```
% function dirs_found = nlIO_searchForDir( startdir, targetname )
%
% This searches a directory tree for folders that match the specified name.
% This wraps "dir", so the folder name can contain wildcards.
%
% "startdir" is the top-level folder to look in.
% "targetname" is the folder name to match. This is passed to "dir", so it
%   can contain wildcards.
%
% "dirs_found" is a cell array containing paths to folders that match the
%   target name.
```

5.11 nlIO_searchForFile.m

```
% function [ fnames_found paths_found ] = ...
%   nlIO_searchForFile( startdir, targetname )
%
% This searches a directory tree for files that match the specified name.
% This wraps "dir", so filenames can contain wildcards.
%
% "startdir" is the top-level folder to look in.
% "targetname" is the file name to match. This is passed to "dir", so it can
%   contain wildcards.
%
% "fnames_found" is a cell array containing the names of files found, without
%   paths.
% "paths_found" is a cell array containing the paths of files found, without
%   filenames.
```

5.12 nlIO_subtractFromChanList.m

```
% function newlist = nlIO_subtractFromChanList(oldlist, removelist)
%
% This function removes the specified members from a channel list, if the
% members are present.
%
% "oldlist" is the channel list (per "CHANLIST.txt").
% "removelist" is a channel list containing members to be removed.
%
% "newlist" is a copy of "oldlist" that does not contain any members that
%   were listed in "removelist".
```

5.13 nlIO_writeBinaryFile.m

```
% function is_ok sampdata = nlIO_writeBinaryFile(fname, sampdata, dtype)
%
% This attempts to write the specified sample data as a packed array of the
% specified data type. Per fwrite(), data is rounded and saturated if
% appropriate.
%
% "fname" is the name of the file to write to.
% "sampdata" is an array containing the sample values.
% "dtype" is a string identifying the Matlab data type (e.g. 'uint32').
%
% "is_ok" is set to true if the operation succeeds and false otherwise.
```

Part II

Abstraction Libraries

Chapter 6

Hardware Library Structures and Additional Notes

6.1 FOLDERMETA_INTAN.txt

```
Intan-specific folder metadata format is as follows. This is intended as a
reference for maintaining code; nothing outside of this set of functions
should need to look at Intan-specific metadata.

The sole exception is knowing what to look for in the "devicetype" field.



In "folder metadata":

- "devicetype" is 'intan'.
- "nativemeta" is the output of nlIntan_readMetadata().
- Banks that may exist are as follows:
  - "AmpA".."AmpH" contain ephys channel signals.
  - "AuxA".."AuxH" contain on-chip aux analog input signals.
  - "DcA".."DcH" contain low-gain DC-coupled ephys channel signals.
  - "Ain" and "Aout" contain BNC analog inputs and outputs, respectiely.
  - "Din" and "Dout" contain TTL inputs and outputs, respectively.
  - "StimA".."StimH" contain stimulation drive currents.
  - "FlagsA".."FlagsH" contain encoded stimulation-related flags.
  - "VddA".."VddH" contain voltage supply signals.



In "signal bank metadata":

- "banktype" is:
  - 'boolean' for "Din" and "Dout", if they're stored per-channel.
  - 'integer' for "Din" and "Dout" otherwise.
```

  - ’flagvector’ for "FlagsA".."FlagsH".
  - ’analog’ for other banks.
- Ephys recording channels have "fpunits" of ’uV’; stimulation channels have "fpunits" of ’uA’. Other analog channels have "fpunits" of ’V’. Channels that aren’t analog have "fpunits" of ’’.
- "handle" is a structure with the following fields:

  - "format" is ’onefileperchan’, ’neuroscope’, or ’monolithic’.
  - "special" is ’’, ’stimflags’ or ’stimcurrent’.

  For "monolithic":
    FIXME - Monolithic NYI.

  For "neuroscope":
    FIXME - Neuroscope NYI.

  For "onefileperchan":
  - "chanfilechans" is a vector containing channel indices in the same order as "chanfilenames".
  - "chanfilenames" is a vector containing channel data filenames in the same order as "chanfilechans".
  - "timefile" is the file to read sample indices from.

This is the end of the file.

6.2 FOLDERMETA_OPENEPHYS.txt

Open Ephys-specific folder metadata format is as follows. This is intended as a reference for maintaining code; nothing outside of this set of functions should need to look at Open Ephys-specific metadata.

The sole exception is knowing what to look for in the "devicetype" field.

FIXME - This may need to be updated if support for one file per channel format and for spike data are added.

In "folder metadata":

- "devicetype" is ’openephys’.
- "firsttime" is the smallest timestamp value seen across all banks, in native format. The I/O functions subtract this from all per-bank timestamps.

In "signal bank metadata":

- "nativemeta" is the "header" field from the output of
  "load_open_ephys_binary()".
- "banktype" is:
  - 'analog' for continuous data.
  - 'eventwords' for the read-as-words alias of event data.
  - 'eventbool' for the read-as-bits alias of event data.
- "firsttime" is the smallest timestamp value seen in this bank, in native
  format.

This is the end of the file.

6.3 FT_EVENTS.txt

Field Trip events are constructed using channels that had type "eventbool"
or "eventwords" (i.e. channels that are stored as sparse rather than
continuous time series).

Field Trip's event record fields that we set are "sample" (event location in
samples), "value" (event value in the native channel type), and "type" (set
to the channel type: "eventbool" or "eventwords"). Other fields are set to
the empty value ([]).

Field Trip does not appear to support event filtering based on source (the
event's channel). To do this, use the "nlFT_selectChannels()" function before
calling "ft_read_events()". There should be exactly one channel selected;
results from selecting multiple channels are undefined.

(This is the end of the file.)

6.4 FT_ITERFUNC.txt

An iteration processing function handle is called to perform signal
processing when iterating across trials and channels within a Field Trip
data structure. The intention is to simplify processing of Field Trip data
using non-FT functions.

An iteration processing function has the form:

```
[ waveresult otherresult ] = ...
```

```
  iterfunc( wavedata, timedata, samprate, trialidx, chanidx, chanlabel )
```

"wavedata" is a vector containing the waveform to be processed (from the
  Field Trip data's "trial" field).
"timedata" is a vector containing sample times (from the Field Trip data's
  "time" field).
"samprate" is the sampling rate (from the Field Trip data's "fsample" field).
"trialidx" is the trial number.
"chanidx" is the channel number.
"chanlabel" is the corresponding channel label (from the Field Trip data's
  "label" field).

"waveresult" is a vector containing modified waveform data. This is
  typically used to build a modified version of the "trial" cell array.
"otherresult" is an arbitrary data type containing any other information
  that the user wishes to associate with this input data/trial/channel.

A typical iteration processing functiton definition would be as follows.
This example wraps a helper function that is passed additional arguments
set at the time the processing function is defined.

```
tuning_parameters = (stuff);
other_parameters = (stuff);
iterfunc = @( wavedata, timedata, samprate, trialidx, chanidx, chanlabel ) ...
  helper_do_iteration_processing( wavedata, timedata, samprate, ...
    trialidx, chanidx, chanlabel, tuning_parameters, other_parameters );
```

This is the end of the file.

6.5 FT_ITERFUNC_BATCHED.txt

A batched iteration processing function handle is called to perform signal
processing when iterating across a very large FT dataset in batches of
channels or batches of trials (to avoid loading the entire dataset into
memory).

The intention is to abstract away the channel and trial batching logic.

A batched iteration processing function has the form:

```
[ ftdata_new auxdata ] = ...
  iterfunc_batched( ftdata_old, chanindices_orig, trialindices_orig )
```

"ftdata_old" is a "ft_datatype_raw" structure containing this batch's data.
"chanindices_orig" is a vector with one entry per channel in ftdata_old
  containing each channel's corresponding index in the larger dataset

(prior to batching).
"trialindices_orig" is a vector with one entry per trial in ftdata_old
containing each trial's corresponding index in the larger dataset
(prior to batching).

"ftdata_new" is a modified version of ftdata_old (after any iterator
data processing is performed), or struct([]) to omit consturction.
"auxdata" is a cell array indexed by {trial, channel} containing arbitrary
user-defined data that the user wishes to associate with each trial/channel.

A typical batched iteration processing function definition would be as
follows. This example wraps a helper function that is passed additional
arguments set at the time the processing function is defined, and also
wraps nlFT_iterateAcrossData() for per-trial/per-channel processing.

```
tuning_parameters = (stuff);
other_parameters = (stuff);
processing_config = (stuff);

iterfunc_batched = @( ftdata_old, chanindices_orig, trialindices_orig ) = ...
  helper_do_batch_iteration( ftdata_old, ...
    chanindices_orig, trialindices_orig ...
    processing_config, tuning_parameters, other_parameters );


function [ ftdata_new auxdata ] = helper_do_batch_iteration( ...
  ftdata_old, chanindices_orig, trialindices_orig, ...
  proc_config, tuning_params, other_params )

  ftdata_new = ft_preprocessing(proc_config, ftdata_old);

  iterfunc_single = @( wavedata, timedata, samprate, ...
    trialidx, chanidx, chanlabel ) ...
    helper_do_channel_iteration( wavedata, timedata, samprate, ...
      trialidx, chanidx, chanlabel, chanindices_orig, trialindices_orig, ...
      tuning_params, other_params );

  [ newtrials auxdata ] = ...
    nlFT_iterateAcrossData( ftdata_new, iterfunc_single );
  ftdata_new.trial = newtrials;

end
```

This is the end of the file.

6.6 INTAN_FILES.txt

```
Data is saved in monolithic format ("traditional Intan"), one-file-per-type
format (NeuroScope-compatible), or one-file-per-channel format.



"Traditional Intan" format stores data as follows:

- Data is stored as a self-contained ".rhd" file (for the recording
  controller) or ".rhs" file (for the stimulate-and-record controller).

- A session may be saved as multiple independent files (with a new file
  started every N minutes).

- All ".rhd" and ".rhs" files begin with a header containing metadata.

- Blocks of data follow the header.

- For file formats other than "traditional Intan", an "info.rhd" or
  "info.rhs" file exists with a header and no blocks of data.

See Intan's file format documentation for stucture details for the header
and the data blocks.



"One-file-per-type" format (NeuroScope-compatible) stores signals in the
following files:

- All files are saved at the full sampling rate for ease of alignment, even
  if the underlying signals are sampled at lower rates (as with auxiliary
  and power supply signals).

- "time.dat" is a series of (signed int32) sample indices.

- "amplifier.dat" stores ephys channel signals (signed int16, 0.195 uV/LSB).

- "auxiliary.dat" (RHD only) stores on-chip aux analog input signals
  (unsigned uint16, 37.4 uV/LSB)

- "dcamplifier.dat" (RHS only) stores low-gain DC-coupled ephys channel
  signals (unsigned uint16, 19.23 mV/LSB, zero level 512).

- "analogin.dat" stores BNC analog input signals (unsigned uint16,
  0.3125 mV/LSB for RHS/RHD, 50.354 uV/LSB mode 0, 0.15259 mV/LSB mode 1,
  zero level 32768).

- "analogout.dat" (RHS only) stores BNC analog output signals (unsigned
  uint16, 0.3125 mV/LSB, zero level 32768).

- "digitalin.dat" stores TTL input signals as a packed unsigned 16-bit word.

- "digitalout.dat" stores TTL output signals as a packed unsigned 16-bit word.

- "stim.dat" (RHS only) stores stimulation current and flags as encoded
  unsigned 16-bit values. The least-significant 8 bits are the current
  magnitude (units per the header), the 9th bit is the sign, the 14th bit is
  amplifier settle, the 15th bit is the charge recovery, and the 16th bit is
  compliance limit. Note that bit numbering starts at 1, not 0.

- "supply.dat" (RHD only) stores on-chip supply voltage sensor signals
  (unsigned uint16, 74.8 uV/LSB)



"One-file-per-type" format (NeuroScope-compatible) stores signals in the
following files:

- File format is per "one-file-per-type", except with only one channel per
  file, unless otherwise indicated.

- All files are saved at the full sampling rate for ease of alignment, even
  if the underlying signals are sampled at lower rates (as with auxiliary
  and power supply signals).

- Note that as of 2021 the stim/record controller (RHS) only has four banks
  ("A".."D") and a maximum of 32 channels per bank ("000".."031").

- "time.dat" is a series of (signed int32) sample indices.

- "amp-X-NNN.dat" files store ephys channel signals for bank X ("A".."H")
  and channel NNN ("000".."127").

- "aux-X-AUXN.dat" files (RHD only) store on-chip aux analog input signals
  for bank X ("A".."H") and channel N ("1".."6"). Channels 1..3 are for the
  first connected chip and channels 4..6 are for the second connected chip
  on that bank.

- "dc-X-NNN.dat" files (RHS only) store low-gain DC-coupled ephys channel
  signals for bank X ("A".."H") and channel NNN ("000".."127").

- "board-ADC-NN.dat" files store BNC analog input signals for
  channel NN ("00".."07").
- "board-ANALOG-IN-NN.dat" is an alternate set of filenames for this.
- FIXME - Saved as 1..8, not 0..7!

- "board-ANALOG-OUT-N" files store BNC analog output signals for
  channel N ("0".."7").
```

```
- FIXME - Saved as 1..8, not 0..7!

- "board-DIN-NN.dat" files store TTL input signals for channel NN
  ("00".."15"). Samples are still unsigned uint16 values but only have
  values of 0 or 1.
- "board-DIGITAL-IN-NN.dat" is an alternate set of filenames for this.
- FIXME - Saved as 1..16, not 0..15!

- "board-DOUT-NN.dat" files store TTL output signals for channel NN
  ("00".."15"). Samples are still unsigned uint16 values but only have
  values of 0 or 1.
- "board-DIGITAL-OUT-NN.dat" is an alternate set of filenames for this.
- FIXME - Saved as 1..16, not 0..15!

- "stim-X-NNN.dat" files store encoded stimulation current and flags for
  bank X ("A".."H") and channel NNN ("000".."127").

- "vdd-X-VDDN.dat" files (RHD only) store on-chip supply voltage sensor
  signals for bank X ("A".."H"). "VDD1" is for the first connected chip and
  "VDD2" is for the second connected chip on that bank.



This is the end of the file.
```

6.7 INTAN_METADATA.txt

```
An Intan metadata structure is a structure containing header information
supplied by RHD and RHS files, in a format consistent with that provided by
"read_Intan_RHD2000_file.m" and "read_Intan_RHS2000_file.m".

The metadata structure has the following fields:

"filename" is the file that was read (including path).
"path" is the folder path containing data files.
"devtype" is 'RHD' or 'RHS'.
"version_major" is the major version number of the data file.
"version_minor" is the minor version number of the data file.
"num_samples_per_data_block" is the number of samples per data block, for
  monolithic data.
"frequency_parameters" is a structure with the following fields:
  "amplifier_sample_rate" is the ephys channel sampling rate.
  "aux_input_sample_rate" is the sampling rate of the on-chip auxiliary
    analog inputs.
  "supply_voltage_sample_rate" is the sampling rate of the chip supply
    voltage measurement.
  "board_adc_sample_rate" is the sampling rate of the RHD controller's
    BNC analog inputs.
  "board_dig_in_sample_rate" is the sampling rate of the RHD controller's
    TTL inputs.
  "desired_dsp_cutoff_frequency" is the user-specified cutoff for the
    on-chip first-order digital high-pass filter.
  "actual_dsp_cutoff_frequency" is the implemented cutoff for the
    on-chip first-order digital high-pass filter.
  "dsp_enabled" is nonzero if the on-chip digital filter is active.
  "desired_lower_bandwidth" is the user-specified lower bandwidth for
    the ephys amplifiers. This is a first-order high-pass filter.
  "actual_lower_bandwidth" is the implemented lower bandwidth for the
    ephys amplifiers. This is a first-order high-pass filter.
  "desired_lower_settle_bandwidth" (RHS-only) is the user-specified lower
    bandwidth for the ephys amplifiers when recovering from stimulation.
  "actual_lower_settle_bandwidth" (RHS-only) is the implemented lower
    bandwidth for the ephys amplifiers when recovering from stimulation.
  "desired_upper_bandwidth" is the user-specified upper bandwidth for
    the ephys amplifiers. This is a third-order low-pass filter.
  "actual_upper_bandwidth" is the implemented upper bandwidth for the
    ephys amplifiers. This is a third-order low-pass filter.
  "notch_filter_frequency" is the center frequency of the power line
    frequency rejection filter, or 0 for no notch filtering. This is
    a software-implemented biquad filter.
  "desired_impedance_test_frequency" is the user-specified frequency
    used for measuring channel impedance.
  "actual_impedance_test_frequency" is the implemented frequency used
    for measuring channel impedance.
"stim_parameters" (RHS only) is a structure with the following fields:
  "stim_step_size" is the current scale used for stimulation (amps/LSB).
  "charge_recovery_current_limit" is the current driven when restoring the
    specified voltage during charge recovery (in amperes).
  "charge_recovery_target_voltage" is the voltage to settle stimulation
    channels to after stimulation, if performing charge recovery (in volts).
  "amp_settle_mode" indicates which method was used for fast-settling
    after stimulation. 0 uses bandwidth-switching (recommended) and 1 resets
    the on-chip amplifiers.
  "charge_recovery_mode" indicates which method of charge recovery was used.
    0 uses current-limited charge recovery and 1 uses a resistive switch.
"notes" is a structure with the following fields:
  "note1" is a character array containing the first notes string.
  "note2" is a character array containing the second notes string.
  "note3" is a character array containing the third notes string.
"num_temp_sensor_channels" is the number of temperature sensor channels.
"dc_amp_data_saved" is nonzero if low-gain DC-coupled amplifier signals were
  saved for each channel (suitable for monitoring stimulation artifacts).
"board_mode" indicates board configuration. This mostly affects BNC
  analog input scale. Mode 0 is 50.354 uV/LSB, mode 1 is 152.59 uV/LSB,
  and mode 13 is 312.5 uV/LSB. These correspond to ranges of 0..3.3V,
  +/-5V, and +/-10.24V.
"voltage_parameters" is a structure with the following fields:
  "amplifier_scale" is the ephys voltage scale in V/LSB.
  "aux_scale" is the chip auxiliary input scale in V/LSB.
  "dcamp_scale" is the RHS low-gain amplifier scale in V/LSB.
  "dcamp_zerolevel" is the RHS low-gain amplifier output with 0V input.
  "board_analog_scale" is the BNC analog input and output scale in V/LSB.
  "board_analog_zerolevel" is the BNC analog I/O value with a 0V signal level.
  "supply_scale" is the RHD supply voltage scale in V/LSB.
  "temperature_scale" is the RHD temperature sensor scale in degC/LSB.
"reference_channel" is a character array containing the name of the
  channel used for re-referencing the input, or an empty string if no
  re-referencing was performed.
"amplifier_channels" is a structure array with zero or more entries
  containing ephys channel metadata.
"spike_triggers" is a structure array with the same number of entries
  as "amplifier_channels", containing spike scope trigger settings for
  each ephys channel.
"aux_input_channels" is a structure array with zero or more entries
  containing ephys chip auxiliary analog input channel metadata.
"supply_voltage_channels" is a structure array with zero or more entries
  containing ephys chip supply voltage monitor channel metadata.
"board_adc_channels" is a structure array with zero or more entries
  containing controller BNC analog input channel metadata.
"board_dig_in_channels" is a structure array with zero or more entries
  containing controller TTL input channel metadata.
"board_dig_out_channels" is a structure array with zero or more entries
  containing controller TTL output channel metadata.
"num_data_blocks" is the number of blocks of monolithic data stored. This is
  zero for NeuroScope or per-channel data.
"header_bytes" is the number of bytes to skip before reading the first
  block of monolithic data.
"bytes_per_block" is the number of bytes per monolithic data block.


Channel metadata structures contain the following fields:

"native_channel_name" is the device-specific channel name (e.g. 'A-015').
"custom_channel_name" is a user-defined channel name (e.g. 'Probe 15').
"native_order" is the device-assigned channel number.
"custom_order" is a user-assigned channel number.
"board_stream" is the device-assigned internal data stream number.
"chip_channel" is the on-chip channel number.
"port_name" is a human-readable name for this channel's port (e.g. 'Port A').
"port_prefix" is a device-assigned label for this channel's port (e.g. 'A').
"port_number" is a device-assigned port index (ports are numbered 1..N).
"electrode_impedance_magnitude", if applicable, is the measured
  impedance magnitude of this channel (in ohms).
"electrode_impedance_phase", if applicable, is the measured impedance
  phase angle of this channel (in degrees).


Spike scope trigger setting structures contain the following fields:
```

"voltage_trigger_mode" is 0 for TTL triggered and 1 for threshold-triggered.
"voltage_threshold" is the trigger threshold in uV.
"digital_trigger_channel" is the TTL trigger input number (0-15).
"digital_edge_polarity" is 0 for falling-edge triggered and 1 for rising-edge.

This is the end of the file.

6.8 OPENEPHYS_CHANMAP.txt

This file documents Matlab structures used to represent configurations of
the ChannelMappingNode plugin in Open Ephys.

In Open Ephys v0.5.x, this also includes referencing information. In Open
Ephys v0.6.x, re-referencing is moved to its own plugin.

An Open Ephys v5 channel map is represented as 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.
```

(This is the end of the file.)

6.9 OPENEPHYS_DATA.txt

Miscellaneous notes about how Open Ephys stores things (in the Womelsdorf Lab
setup) and about Open Ephys's I/O functions.

Open Ephys's I/O functions: https://github.com/open-ephys/analysis-tools/

Open Ephys has two formats:
- "Open Ephys" format, with one file per channel. This is deprecated.

- "Binary" format, with one file for all continuous data.

"Open Ephys" format is structured as follows:

- Each session just produces one directory.
- There's a "settings.xml" file in the session directory.
- There's a "Continuous_Data.openephys" file with metadata.
- There are "(label).continuous" files with the continuous data itself.
- There are a small number of ".events" files with monolithic event data.

"Binary" format is structured as follows:

- Each session gets a deep directory tree.
- There's a "settings.xml" file in a session's root directory.
- For each experiment and recording node, there's a "structure.oebin" file that describes what that node recorded.
- There are directory trees storing continuous data, event data, and spike data from devices attached to the recording node. Any given device produces a small number of monolithic data files (data, timestamps, event states, etc).

Open Ephys has several I/O functions, with a few peculiarities:

- "list_open_ephys_binary" is pointed at a directory containing a ".oebin" file. It returns a cell array containing the names of subfolders which have the type of data requested. Zero, one, or multiple folders may exist.
- "load_open_ephys_binary" loads folder number N of a requested type from the list produced by "list_open_ephys_binary". This returns header information with metadata, and either data or a memory-mapped file handle. You want to memory-map this data; it's usually too big for RAM.

- "get_session_info" is pointed at a directory containing "settings.xml". This is supposed to read configuration metadata, but in practice it only does this for sessions stored in the older "Open Ephys" format. So, you're flying blind with ".oebin" binary format.
- "load_open_ephys_data" and its variants are intended to read data stored in the older "Open Ephys" format. I haven't tested these, since we only use the new format.

Event data (from "binary" format) is structured as follows:

- Event lists are stored as .npy tables in subfolders indicating the event type which are in turn in subfolders indicating the devices of origin.
- TTL events have the following fields:
  - Timestamps (in samples).
  - ChannelIndex (channel number that changed)

```
  - Data (+chan for rising edges or -chan for falling)
  - FullWords (Nx2 uint8 matrix with word bytes; 1 = least significant)
- This data is redundant; ChannelIndex == abs(Data), and FullWords can be
  reconstructed from data and vice versa.
- Multiple bits changing at the same time results in multiple events with
  the same timestamp.

- FIXME - No information about what's in text events, as these read as empty.


(This is the end of the file.)
```

6.10 OPENEPHYS_TTLWORDS.txt

```
Something very important to keep in mind about Open Ephys's TTL events:

Individual TTL bit lines are always correct and consistent, but the word data
associated with these events is _not_ always consistent.


The problem happens because Open Ephys generates an event every time a bit
changes, so if multiple bits change during one sample, multiple events with
the same timestamp are generated. The "FullWords" vector is updated after
each event, one bit at a time - so events with the same timestamp will have
"FullWords" vectors with different content. The events are not necessarily
returned in the same order that they were processed in, so there's no easy
way to tell which "FullWords" value is the right one in a set of events
that have the same timestamp.


The right thing to do is to either build your own word data and ignore
the words provided by OpenEphys, or else to use additional knowledge about
the experiment scenario to disambiguate these cases.


(This is the end of the file.)
```

Chapter 7

"nlFT" Functions

7.1 nlFT_compareChannelMaps.m

```
% function [ firstok secondok logtext ] = nlFT_compareChannelMaps( ...
%   firstsrc, firstdst, secondsrc, seconddst )
%
% This compares two label-based channel maps and flags discrepancies.
%
% "firstsrc" is a cell array of "before mapping" labels for the first map.
% "firstdst" is a cell array of "after mapping" labels for the first map.
% "secondsrc" is a cell array of "before mapping" labels for the second map.
% "seconddst" is a cell array of "after mapping" labels for the second map.
%
% "firstok" is a vector of the same size as "firstsrc" containing "true" for
%   entries that are consistent between the first and second maps.
% "secondok" is a vector of the same size as "secondsrc" containing "true"
%   for entries that are consistent between the first and second maps.
% "logtext" is a character vector containing a human-readable summary report.
```

7.2 nlFT_compressFTEvents.m

```
% function [ evtable newlut ] = nlFT_compressFTEvents(evstructarray, typelut)
%
% This compresses a Field Trip event list by converting type labels into
% numbers, removing "offset" and "duration" if empty, and storing the result
% as a table. MatLab structures have more overhead and MatLab cell arrays
% have a _lot_ of overhead, so the result takes up much less memory.
%
% This is intended to be used with event lists generated by the LoopUtil
% library, which store TTL state changes as events with "type" holding the
% channel label.
%
```

```
% FIXME - Fallback for numeric rather than character "type" data is to copy
% the type field without translation and return an empty LUT.
%
% "evstructarray" is the Field Trip event list (structure array).
% "typelut" is a cell array containing values expected in the "type" field.
%   For event lists generated by the LoopUtil library, this should be the
%   "label" field from the Field Trip header (channel label list).
%
% "evtable" is a table containing data from the event list.
% "newlut" is a copy of "typelut" with additional entries for any "type"
%   values that weren't recognized.
```

7.3 nlFT_evalChannelMapFit.m

```
% function logtext = nlFT_evalChannelMapFit( ...
%   srclabels, dstlabels, data_before, data_after, rmatrix, pmatrix )
%
% This evaluates how well a supplied channel mapping matches correlations
% between channels in "before" and "after" datasets.
%
% The datasets must have the same sampling rate, the same number of trials,
% and the same number of samples in corresponding trials.
%
% NOTE - Computing correlation values is slow! Time goes up as the square of
% the number of channels.
%
% "srclabels" is a cell array containing channel labels from "data_before".
% "dstlabels" is a cell array containing corresponding channel labels from
%   "data_after".
% "data_before" is a Field Trip dataset prior to channel mapping.
% "data_after" is a Field Trip dataset after channel mapping.
% "rmatrix" is the matrix of R-values returned by nlFT_getChannelCorrelMatrix.
%   If this argument is omitted, the R-value matrix is recomputed.
% "pmatrix" is the matrix of P-values returned by nlFT_getChannelCorrelMatrix.
%   If this argument is omitted, the P-value matrix is recomputed.
%
% "logtext" is a character vector containing a human-readable summary report.
```

7.4 nlFT_findChannelIndices.m

```
% function newindices = nlFT_findChannelIndices( ftheader, chanlabels )
%
% This returns a vector of Field Trip channel indices corresponding to the
% specified list of Field Trip channel labels.
%
% NOTE - If channel labels aren't unique, a matching channel label is chosen
% arbitrarily. If channel labels aren't found in the FT header, an index of
% NaN is returned for that channel.
```

7.5 nlFT_getChannelCorrelMatrix.m

```
% function [ rmatrix pmatrix ] = ...
%   nlFT_getChannelCorrelMatrix( data_before, data_after, wantprogress )
%
% This checks for correlation between channels in the "before" and "after"
% datasets, returning the correlation coefficients and null hypothesis
% P-values for each (before,after) pair (per the "corrcoef" function).
%
% The datasets must have the same sampling rate, the same number of trials,
% and the same number of samples in corresponding trials. Trials are
% concatenated to compute correlation statistics.
%
% NOTE - This is slow! Time goes up as the square of the number of channels.
% A progress banner is printed.
%
% NOTE - Matrix values are NaN for cases where an input had zero variance.
%
% "data_before" is a Field Trip dataset prior to channel mapping.
% "data_after" is a Field Trip dataset after channel mapping.
% "wantprogress" is true to display a progress banner, false otherwise.
%   If omitted, it defaults to true.
%
% "rmatrix" is a matrix indexed by (chanbefore,chanafter) containing
%   correlation coefficient values.
% "pmatrix" is a matrix indexed by (chanbefore,chanafter) containing P-values
%   for the null hypothesis (that the channels are not correlated).
```

7.6 nlFT_getEstimatedChannelMapping.m

```
% function [ srclabels, dstlabels ] = nlFT_getEstimatedChannelMapping( ...
%   data_before, data_after, rmatrix, pmatrix )
%
% This attempts to infer a channel mapping from "data_before" to "data_after"
% by looking at correlations between individual channels in each data set.
%
% The datasets must have the same sampling rate, the same number of trials,
% and the same number of samples in corresponding trials.
%
% Not all channels are guaranteed to be mapped. Channels that couldn't be
% mapped are omitted from the output. Order of labels in the output is not
% guaranteed.
%
```

```
% NOTE - Computing correlation values is slow! Time goes up as the square of
% the number of channels.
%
% "data_before" is a Field Trip dataset prior to channel mapping.
% "data_after" is a Field Trip dataset after channel mapping.
% "rmatrix" is the matrix of R-values returned by nlFT_getChannelCorrelMatrix.
%   If this argument is omitted, the R-value matrix is recomputed.
% "pmatrix" is the matrix of P-values returned by nlFT_getChannelCorrelMatrix.
%   If this argument is omitted, the P-value matrix is recomputed.
%
% "srclabels" is a cell array containing channel labels from "data_before".
% "dstlabels" is a cell array containing corresponding channel labels from
%   "data_after".
```

7.7 nlFT_getLabelChannelMapFromNumbers.m

```
% function [ maplabelsraw maplabelscooked ] = ...
%   nlFT_getLabelChannelMapFromNumbers( ...
%     chanmap, idxlabelsraw, idxlabelscooked )
%
% This function translates a channel mapping table defined using channel
% indices into a channel mapping table defined using labels.
%
% "chanmap" is a vector indexed by cooked channel number containing the raw
%   channel number that maps to each cooked location, or NaN if none does.
%   the first channel index is 1, per Matlab conventions.
% "idxlabelsraw" is a cell array containing all raw channel names.
% "idxlabelscooked" is a cell array containing all cooked channel names.
%
% "maplabelsraw" is a cell array containing raw channel names that correspond
%   to the names in "maplabelscooked".
% "maplabelscooked" is a cell array containing cooked channel names that
%   correspond to the names in "maplabelsraw".
```

7.8 nlFT_getMemChans.m

```
% function memchans = nlFT_getMemChans()
%
% This queries the maximum number of data channels that can be loaded into
% memory at one time (the "memchans" argument for NeuroLoop iterating
% functions).
%
% The higher this is, the faster data is read, due to not having to repeatedly
% scan over data files that store matrix data. The downside is that memory
% requirements can get big very quickly (typically 1 gigabyte per
% channel-hour of data).
```

```
%
% "memchans" is the current maximum number of memory-resident channels.
%
% FIXME - This stores state as global variables. This was the least-ugly
% way of passing tuning parameters to low-level reading functions.
```

7.9 nlFT_getWantedChannels.m

```
% function [ typeswanted nameswanted bankswanted ] = nlFT_getWantedChannels()
%
% This reports the type, name, and bank patterns set by the most recent
% call to nlFT_selectChannels().
%
% Types must match exactly. Individual channel and bank names are valid
% regex patterns. Empty lists of patterns or labels always match, indicating
% absence of filtering on that particular criterion.
%
% "typeswanted" is a cell array containing a list of labels. The channel
%   type (from LoopUtil's "banktype" or Field Trip's "chantype") must be in
%   the list for the channel to be processed.
% "nameswanted" is a cell array containing a list of regex patterns. The
%   channel name (from Field Trip's "label") must match one of the regexes
%   in the list for the channel to be processed.
% "bankswanted" is a cell array containing a list of regex patterns. The
%   bank name (from LoopUtil; used as the prefix for channel names) must
%   match one of the regexes in the list for the channel to be processed.
%
% FIXME - This stores state as global variables. This was the least-ugly way
% of implementing channel and bank filtering without modifying Field Trip.
```

7.10 nlFT_initReadTable.m

```
% function nlFT_initReadTable( datatable, chancolumns, timecolumn, ...
%   timefirst, timelast, samplespertimeunit, samprate )
%
% This stores a table and metadata to be read using nlFT_readTableHeader()
% and nlFT_readTableData(). The idea is to be able to give Field Trip a way
% to read non-uniformly-sampled tabular data as waveform data.
%
% FIXME - Support for nlFT_readTableEvents NYI.
%
% To release memory allocated for the copy of the table, call this with an
% empty table.
%
% "datatable" is the table to read.
% "chancolumns" is a cell array containing column labels of data columns.
% "timecolumn" is the label of the timestamp column.
% "timefirst" is the timestamp value corresponding to sample 1.
% "timelast" is the timestamp value corresponding to the last waveform sample.
% "samplespertimeunit" is the number of samples corresponding to a timestamp
%   change of 1.0 timestamp units. For seconds, this is equal to "samprate".
% "samprate" is the number of samples per second.
%
% FIXME - This stores state as global variables, including a copy of the
% table. This was the least-ugly way to store persistent state.
```

7.11 nlFT_iterateAcrossData.m

```
% function [ newtrials auxdata ] = nlFT_iterateAcrossData( ftdata, iterfunc )
%
% This iterates across the trials and channels in a Field Trip dataset,
% applying a processing function to each trial and channel's data. Processing
% output is aggregated and returned.
%
% "ftdata" is an "ft_datatype_raw" data structure.
% "iterfunc" is a function handle used to transform channel waveform data
%   into "result" data, per FT_ITERFUNC.txt.
%
% "newtrials" is a processed copy of the "trial" field, containing modified
%   per-trial and per-channel waveform data.
% "auxdata" is a cell array indexed by {trial,channel} containing auxiliary
%   data returned by the iteration processing function.
```

7.12 nlFT_iterateAcrossFolderBatching.m

```
% function auxdata = nlFT_iterateAcrossFolderBatching( ...
%   config_load, iterfunc_batched, chanbatchsize, trialbatchsize, verbosity )
%
% This iterates across a Field Trip dataset, loading a few channels at a
% time and/or a few trials at a time and applying a processing function to
% each channel subset. Processing output is aggregated and returned.
%
% The idea is to be able to process a dataset much larger than can fit in
% memory. At 30 ksps the footprint is typically about 1 GB per channel-hour.
%
% This calls an iteration function handle of the type described by
% FT_ITERFUNC_BATCHED.txt.
%
% NOTE - If the iteration function returns anything for "ftdata_new", make
% sure the aggregated result is small enough to fit in memory!
%
% "config_load" is a Field Trip configuration structure to be passed to
%   ft_preprocessing() to load the data. The "channel" field is split into
%   batches when iterating.
% "iterfunc_batched" is a function handle used to transform channel waveform
%   data into "result" data, per FT_ITERFUNC_BATCHED.txt.
% "chanbatchsize" is the number of channels to process at a time. Set this to
%   inf to process all channels at once.
% "trialbatchsize" is the number of trials to process at a time. Set this to
%   inf to process all trials at once.
% "verbosity" is an optional argument. It can be set to 'none' (no console
%   output), 'terse' (reporting the number of channels processed), or 'full'
%   (reporting the names of channels processed). The default is 'none'.
%
% "ftdata" is a "ft_datatype_raw" structure built by aggregating the Field
%   Trip data structures ("ftdata_new") returned by the iteration processing
%   function. These are assumed to be compatible (same sampling rate, etc),
%   and the aggregated result must fit in memory. If the iteration function
%   returns struct([]) as ftdata_new, "ftdata" is also set to struct([]).
% "auxdata" is a cell array indexed by {trial,channel} containing the
%   auxiliary data ("auxdata") returned by the iteration processing
%   function.
```

7.13 nlFT_makeFTName.m

```
% function newlabel = nlFT_makeFTName( banklabel, channum )
%
% This turns a NeuroLoop bank label and channel number into a Field Trip
% channel label.
%
% "banklabel" is the NeuroLoop bank label (a valid field name character array).
% "channum" is the NeuroLoop channel number (an arbitrary nonnegative integer).
%
% "newlabel" is a character array containing the corresponding Field Trip
%   channel label.
```

7.14 nlFT_makeLabelsFromNumbers.m

```
% function chanlabels = nlFT_makeLabelsFromNumbers( bankname, channums )
%
% This function calls "nlFT_makeFTName" to convert channel numbers into
% labels for a list of channel numbers.
%
% "bankname" is the bank name to use when building channel labels.
% "channums" is a vector containing channel numbers.
%
% "chanlabels" is a Nx1 cell array containing channel labels.
```

7.15 nlFT_mapChannelLabels.m

```
% function newlabels = nlFT_mapChannelLabels( oldlabels, lutold, lutnew )
%
% This function translates channel labels according to a lookup table.
% This is intended to be used for channel mapping.
%
% "oldlabels" is a cell array containing labels to translate.
% "lutold" is a cell array containing old labels that correspond to the
%   labels in "lutnew".
% "lutnew" is a cell array containing new labels that correspond to the
%   labels in "lutold".
%
% "newlabels" is a cell array of translated labels. Old labels are turned
%   into corresponding new labels. Labels that can't be translated are
%   replaced with ''.
```

7.16 nlFT_mergeEvents.m

```
% function eventsmerged = nlFT_mergeEvents( alleventmeta )
%
% This serializes the multiple event lists produced by nlFT_readAllEvents(),
% merging them into a single Field Trip event list.
%
% This discards the event metadata, and replaces the "type" field of each
% event with the Field Trip channel label for the event's source.
%
% "alleventmeta" is a vector of structures containing event channel metadata
%   and Field Trip event lists for each channel, per nlFT_readAllEvents().
%
% "eventsmerged" is a Field Trip event list struct vector containing all
%   events. Stored fields are:
%   "sample" - Event onset time in samples.
%   "value" - Event value (boolean or integer code).
%   "type" - Character array with the event source's channel label.
```

7.17 nlFT_parseChannelsIntoBanks.m

```
% function bankdata = nlFT_parseChannelsIntoBanks( chanlabels, wavedata )
%
% This tries to identify channels' banks by parsing channel labels, and
% optionally divides one trial's data into per-bank data. This will only
% work with channel labels generated by the LoopUtil library (i.e. with
% data read using the LoopUtil Field Trip I/O functions).
%
```

```
% "chanlabels" is a cell array containing channel names. This is normally
%   taken from header.label or rawdata.label.
% "wavedata" is a Nchans*Nsamples matrix containing waveform data. This is
%   normally taken from rawdata.trial{k}. An empty matrix skips data copying.
%
% "bankdata" is a structure indexed by LoopUtil bank identifier. Each field
%   contains a bank data structure with a "label" field containing a cell
%   array of FT channel labels, a "channum" field containing a vector of
%   LoopUtil channel indices, and optionally a "wavedata" field containing
%   a Nchans*Nsamples matrix of waveform data.
```

7.18 nlFT_parseFTName.m

```
% function [ bankid chanid ] = nlFT_parseFTName( chanlabel )
%
% This parses a name generated by nlFT_makeFTName, turning the compound name
% back into bank and channel IDs.
```

7.19 nlFT_pruneFTEvents.m

```
% function newlist = nlFT_pruneFTEvents(oldlist)
%
% This traverses a list of events stored in Field Trip format, removing
% event records which, for a given "type", have the same "value" as the
% previously-seen event of that "type".
%
% This is intended to be used with event lists generated by the LoopUtil
% library, which store TTL state changes as events with "type" holding the
% channel label.
%
% "oldlist" is the Field Trip event list to process.
%
% "newlist" is the pruned list.
```

7.20 nlFT_readAllEvents.m

```
% function alleventmeta = nlFT_readAllEvents( indir, wantpromote )
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% calls nlFT_readEvents() for any channels of type "eventbool" or
% "eventwords" that pass the nlFT_selectChannels() criteria. The results for
% each channel are returned, with metadata.
%
```

```
% NOTE - Field Trip channel number is sensitive to filtering. Use
% nlFT_findChannelIndices() to modify ftchanidx if filtering changes.
%
% "indir" is the directory to process.
% "wantpromote" is true if continuous data ("integer", "bool", and
%   "flagvector" types) is to be converted into sparse data, and false if not.
%
% "alleventmeta" is a vector of structures with the following fields:
%   "ftchanlabel" is the Field Trip channel label for this channel.
%   "ftchanidx" is the Field Trip channel number for this channel.
%   "ftchantype" is the Field Trip channel type for this channel.
%   "nlbankid" is the NeuroLoop bank name for this channel.
%   "nlchanid" is the NeuroLoop channel number for this channel.
%   "ftevents" is the Field Trip event list struct array for this channel.
```

7.21 nlFT_readDataDouble.m

```
% function data = ...
%   nlFT_readDataDouble( indir, header, firstsample, lastsample, chanidxlist )
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% reads all appropriate signal data into a Field Trip data matrix.
%
% This is intended to be called by ft_read_data() via the "dataformat"
% argument.
%
% The "Double" version of this function promotes data to double-precision
% floating-point.
%
% NOTE - This returns monolithic data (a 2D matrix), not epoched data.
% NOTE - This requires all selected banks to have the same sampling rate and
% number of samples!
%
% This calls nlFT_testWantChannel() and nlFT_testWantBank() and only reads
% channels that are wanted. By default all channels and banks are wanted;
% use nlFT_selectChannels() to change this.
%
% If directory probing fails, and error is thrown.
%
% "indir" is the directory to process.
% "header" is the Field Trip header associated with this directory.
% "firstsample" is the index of the first sample to read.
% "lastsample" is the index of the last sample to read.
% "chanidxlist" is a vector containing Field Trip channel indices to read.
%
% "data" is the resulting 2D data matrix.
```

7.22 nlFT_readDataNative.m

```
% function data = ...
%   nlFT_readDataNative( indir, header, firstsample, lastsample, chanidxlist )
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% reads all appropriate signal data into a Field Trip data matrix.
%
% This is intended to be called by ft_read_data() via the "dataformat"
% argument.
%
% The "Native" version of this function keeps data as its native type.
%
% NOTE - This returns monolithic data (a 2D matrix), not epoched data.
% NOTE - This requires all selected banks to have the same sampling rate and
% number of samples!
% NOTE - This requires all selected banks to have compatible types! The first
% channel processed determines the returned type. For best results, make sure
% all selected banks have the _same_ type.
%
% This calls nlFT_testWantChannel() and nlFT_testWantBank() and only reads
% channels that are wanted. By default all channels and banks are wanted;
% use nlFT_selectChannels() to change this.
%
% If directory probing fails, and error is thrown.
%
% "indir" is the directory to process.
% "header" is the Field Trip header associated with this directory.
% "firstsample" is the index of the first sample to read.
% "lastsample" is the index of the last sample to read.
% "chanidxlist" is a vector containing Field Trip channel indices to read.
%
% "data" is the resulting 2D data matrix.
```

7.23 nlFT_readData_helper.m

```
% function data = nlFT_readData_helper( indir, wantnative, ...
%   header, firstsample, lastsample, chanidxlist )
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% reads all appropriate signal data into a Field Trip data matrix.
%
% This may either promote data to double or keep it as its native type,
% depending on "wantnative".
%
% NOTE - This returns monolithic data (a 2D matrix), not epoched data.
% NOTE - This requires all selected banks to have the same sampling rate and
% number of samples!
%
% This calls nlFT_testWantChannel() and nlFT_testWantBank() and only saves
% channels that are wanted. By default all channels and banks are wanted;
% use nlFT_selectChannels() to change this.
%
% If directory probing fails, an error is thrown.
%
% "indir" is the directory to process.
% "wantnative" is true to store native-format data and false to promote to
%   double-precision floating-point.
% "header" is the Field Trip header associated with this directory.
% "firstsample" is the index of the first sample to read (starting at 1).
% "lastsample" is the index of the last sample to read.
% "chanidxlist" is a vector containing Field Trip channel indices to read.
%
% "data" is the resulting 2D data matrix.
```

7.24 nlFT_readEvents.m

```
% function eventlist = nlFT_readEvents( indir, header )
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% reads events from all sparse (event-type) channels found. The channel name
% for each event is stored in the event's "type" field.
%
% This is intended to be called by ft_read_event() via the "eventformat"
% argument.
%
% NOTE - Field Trip expects this to return the header, rather than an event
% list, if it's called with just one argument ("indir").
%
% NOTE - Timestamps are guaranteed to be in order, but there are no order
% guarantees for events with the same timestamp (such as simultaneous events
% from different channels).
%
% "indir" is the directory to process.
% "header" is the Field Trip header associated with this directory.
%
% "eventlist" is a vector of field trip event records with the "sample",
%   "value", and "type" fields filled in. The "type" field contains the
%   label of the channel that sourced the event.
```

7.25 nlFT_readEventsContinuous.m

```
% function eventlist = nlFT_readEventsContinuous( indir, header )
```

```
%
% This probes the specified directory using nlIO_readFolderMetadata(), and
% reads events from all sparse (event-type) or discrete-valued continuous
% (boolean/integer/flagvector) channels found. The channel name for each
% event is stored in the event's "type" field.
%
% This is intended to be called by ft_read_event() via the "eventformat"
% argument.
%
% NOTE - Field Trip expects this to return the header, rather than an event
% list, if it's called with just one argument ("indir").
%
% NOTE - Timestamps are guaranteed to be in order, but there are no order
% guarantees for events with the same timestamp (such as simultaneous events
% from different channels).
%
% "indir" is the directory to process.
% "header" is the Field Trip header associated with this directory.
%
% "eventlist" is a vector of field trip event records with the "sample",
%   "value", and "type" fields filled in. The "type" field contains the
%   label of the channel that sourced the event.
```

7.26 nlFT_readHeader.m

```
% function header = nlFT_readHeader(indir)
%
% This probes the specified directory using nlIO_readFolderMetadata(),
% and translates the folder's metadata into a Field Trip header.
%
% This is intended to be called by ft_read_header() via the "headerformat"
% argument.
%
% This calls nlFT_testWantChannel() and nlFT_testWantBank() and only saves
% channels that are wanted. By default all channels and banks are wanted;
% use nlFT_selectChannels() to change this.
%
% NOTE - The following nonstandard fields are added to the header. All of
% these are cell arrays with per-channel metadata copied from the LoopUtil
% bank metadata (per FOLDERMETA.txt).
%   "channativezero" is the native data value representing a signal value of
%     zero.
%   "channativescale" is a multiplier used to convert "native" data values to
%     double-precision floating-point data values in appropriate units.
%   "chanflagbits" is a structure indexed by flag label containing the
%     integer bit-mask values that correspond to each flag, for "flagvector"
%     data. This is an empty structure for other data.
%
```

```
% If probing fails, an error is thrown.
%
% "indir" is the directory to process.
%
% "header" is the resulting Field Trip header.
```

7.27 nlFT_readTableData.m

```
% function data = ...
%   nlFT_readTableData( fname, header, firstsample, lastsample, chanidxlist )
%
% This constructs Field Trip waveform data based on information previously
% supplied to nlFT_initReadTable(). The idea is to be able to give Field
% Trip a way to read non-uniformly-sampled tabular data as waveform data.
%
% This is intended to be called by ft_read_data() via the "dataformat"
% argument.
%
% NOTE - This returns monolithic data (a 2D matrix), not epoched data.
%
% "fname" is the filename passed to ft_read_data(). This is ignored.
% "header" is the Field Trip header returned by nlFT_readTableHeader().
% "firstsample" is the index of the first sample to read.
% "lastsample" is the index of the last sample to read.
% "chanidxlist" is a vector containing Field Trip channel indices to read.
%
% "data" is the resulting 2D data matrix.
```

7.28 nlFT_readTableHeader.m

```
% function header = nlFT_readTableHeader(fname)
%
% This constructs a Field Trip header based on information previously
% supplied to nlFT_initReadTable(). The idea is to be able to give Field
% Trip a way to read non-uniformly-sampled tabular data as waveform data.
%
% This is intended to be called by ft_read_header() via the "headerformat"
% argument.
%
% "fname" is the filename passed to ft_read_header(). This is ignored.
```

7.29 nlFT_selectAllChannels.m

```
% function nlFT_selectAllChannels()
%
% This clears LoopUtil's channel filtering.
% This is a wrapper for nlFT_selectChannels().
```

7.30 nlFT_selectChannels.m

```
% function nlFT_selectChannels( typeswanted, nameswanted, bankswanted )
%
% This sets conditions that chanels must match in order to be processed.
% Channels must have a desired type, have a name that matches a desired
% regex, or be part of a bank whose label matches a desired regex.
%
% Individual channel and bank names are valid regex patterns.
%
% Empty lists always match (so pass "{}" as a condition to disable that test).
%
% "typeswanted" is a cell array containing a list of labels. The channel
%   type (from LoopUtil's "banktype" or Field Trip's "chantype") must be in
%   the list for the channel to be processed.
% "nameswanted" is a cell array containing a list of regex patterns. The
%   channel name (from Field Trip's "label") must match one of the regexes
%   in the list for the channel to be processed.
% "bankswanted" is a cell array containing a list of regex patterns. The
%   bank name (from LoopUtil; used as the prefix for channel names) must
%   match one of the regexes in the list for the channel to be processed.
%
% FIXME - This stores state as global variables. This was the least-ugly way
% of implementing channel and bank filtering without modifying Field Trip.
```

7.31 nlFT_selectOneFTChannel.m

```
% function nlFT_selectOneFTChannel( chanlabel )
%
% This sets up LoopUtil's channel filtering to pass one specific channel
% using Field Trip's channel label.
%
% This is a wrapper for nlFT_selectChannels().
%
% "chanlabel" is the Field Trip channel label for the desired channel.
```

7.32 nlFT_setMemChans.m

```
% function nlFT_setMemChans(newcount)
%
% This sets the maximum number of data channels that can be loaded into
% memory at one time (the "memchans" argument for NeuroLoop iterating
% functions).
%
% The higher this is, the faster data is read, due to not having to repeatedly
% scan over data files that store matrix data. The downside is that memory
% requirements can get big very quickly (typically 1 gigabyte per
% channel-hour of data).
%
% "newcount" is the new maximum number of memory-resident channels.
%
% FIXME - This stores state as global variables. This was the least-ugly
% way of passing tuning parameters to low-level reading functions.
```

7.33 nlFT_testWantBank.m

```
% function iswanted = nlFT_testWantBank(bankname, banktype)
%
% This tests to see if a given bank is desired, per nlFT_selectChannels().
% A bank is desired if its type and bank name are both acceptable.
%
% "bankname" is the bank name to test.
% "banktype" is the bank type label to test.
%
% FIXME - This stores state as global variables. This was the least-ugly way
% of implementing channel and bank filtering without modifying Field Trip.
```

7.34 nlFT_testWantChannel.m

```
% function iswanted = nlFT_testWantChannel(channelname)
%
% This tests to see if a given channel name is in the list of desired channel
% names, per nlFT_selectChannels().
%
% "channelname" is the channel name to test.
%
% FIXME - This stores state as global variables. This was the least-ugly way
% of implementing channel and bank filtering without modifying Field Trip.
```

7.35 nlFT_uncompressFTEvents.m

```
% function evstructarray = nlFT_uncompressFTEvents(evtable, typelut)
%
% This de-compresses a compressed Field Trip event list that was produced by
% nlFT_compressFTEvents(). Table columns are converted back into structure
% fields, the "type" column is converted back to cell data, and empty
% "offset" and "duration" fields are created if not already present.
%
% If an empty LUT is given, the "type" column is copied as-is rather than
% translated. Otherwise all compressed "type" values must be valid indices
% into the lookup table.
%
% "evtable" is a table containing compressed event data.
% "typelut" is a cell array containing values to store in the "type" field
%   in the reconstructed structure array.
%
% "evstructarray" is a Field Trip event list (structure array).
```

Chapter 8

"nlIntan" Functions

8.1 nlIntan_iterateFolderChannels.m

```
% function folderresults = nlIntan_iterateFolderChannels( ...
%   foldermetadata, folderchanlist, memchans, procfunc, procmeta, procfid )
%
% This processes a folder containing Intan-format data, iterating through
% a list of channels, loading each channel's waveform data in sequence and
% calling a processing function with that data. Processing output is
% aggregated and returned.
%
% This is implemented such that only a few channels are loaded at a time.
%
% "Native" channel time series are stored as sample numbers (not times).
% "Cooked" channel time series are in seconds. Cooked analog data is
% converted to the units specified in the bank metadata (volts or microvolts).
% Cooked TTL data is converted to boolean.
%
% "foldermetadata" is a folder-level metadata structure, per FOLDERMETA.txt.
% "folderchanlist" is a structure listing channels to process; it is a
%   folder-level channel list per CHANLIST.txt.
% "memchans" is the maximum number of channels that may be loaded into
%   memory at the same time.
% "procfunc" is a function handle used to transform channel waveform data
%   into "result" data, per PROCFUNC.txt.
% "procmeta" is the object to pass as the "metadata" argument of "procfunc".
% "procfid" is the label to pass as the "folderid" argument of "procfunc".
%
% "folderresults" is a folder-level channel list structure that has
%   bank-level channel lists augmented with a "resultlist" field, per
%   CHANLIST.txt. The "resultlist" field is a cell array containing
%   per-channel output from "procfunc".
```

8.2 nlIntan_probeFolder.m

```
% function foldermeta = nlIntan_probeFolder( indir )
%
% This checks for the existence of Intan-format data files in the specified
% folder, and constructs a folder metadata structure if data is found.
%
% If no data is found, an empty structure is returned.
%
% "indir" is the directory to search.
%
% "foldermeta" is a folder metadata structure, per FOLDERMETA.txt.
```

Chapter 9

"vIntan" Functions

9.1 vIntan_readHeader.m

```
% function metadata = vIntan_readHeader( fname )
%
% This reads the metadata header from Intan ".rhd" and ".rhs" files.
% If reading fails, an empty structure array is returned.
%
% This file is derived from code supplied by Intan Technologies (used and
% re-licensed with permission).
%
% "fname" is the name of the file to read from, including path.
%
% "metadata" is a structure containing header data, per "INTAN_METADATA.txt".
```

Chapter 10

“nlOpenE” Functions

10.1 nlOpenE_assembleWords.m

```
% function wordvals = nlOpenE_assembleWords(bytevals, wordtype)
%
% This function assembles the bytes stored in an event list's "FullWords"
% array into word values. This tolerates a list with zero events.
%
% "bytevals" is a copy of the FullWords array (Nevents x Nbytes uint8 LE).
% "wordtype" is a character array containing the name of the type to promote
%   to (typically 'uint16', 'uint32', or 'uint64').
%
% "wordvals" is a vector containing assembled word values.
```

10.2 nlOpenE_iterateFolderChannels.m

```
% function folderresults = nlOpenE_iterateFolderChannels( ...
%   foldermetadata, folderchanlist, memchans, procfunc, procmeta, procfid )
%
% This processes a folder containing Open Ephys format data, iterating
% through a list of channels, loading each channel's waveform data in
% sequence and calling a processing function with that data. Processing
% output is aggregated and returned.
%
% This is implemented such that only a few channels are loaded at a time.
%
% "Native" channel time series are stored as sample numbers (not times).
% "Cooked" channel time series are in seconds. Cooked analog data is
% converted to the units specified in the bank metadata (volts or microvolts).
% Cooked TTL data is converted to boolean.
%
% "foldermetadata" is a folder-level metadata structure, per FOLDERMETA.txt.
% "folderchanlist" is a structure listing channels to process; it is a
%   folder-level channel list per CHANLIST.txt.
% "memchans" is the maximum number of channels that may be loaded into
%   memory at the same time.
% "procfunc" is a function handle used to transform channel waveform data
%   into "result" data, per PROCFUNC.txt.
% "procmeta" is the object to pass as the "metadata" argument of "procfunc".
% "procfid" is the label to pass as the "folderid" argument of "procfunc".
%
% "folderresults" is a folder-level channel list structure that has
%   bank-level channel lists augmented with a "resultlist" field, per
%   CHANLIST.txt. The "resultlist" field is a cell array containing
%   per-channel output from "procfunc".
```

10.3 nlOpenE_parseChannelMapGeneric_v5.m

```
% function thismap = ...
%   nlOpenE_parseChannelMapGeneric_v5( chanlist, reflist, reflut, enlist )
%
% This parses an array of source channel indices indices, an array of
% reference set indices, a reference lookup list, and an array of "enabled"
% flags, and assembles a structure describing this channel mapping (per
% "OPENEPHYS_CHANMAP.txt").
%
% NOTE - Reference banks start at 1, not 0. Convert before calling.
%
% This is intended to be called by other nlOpenE_parseChannelMap functions.
%
% "chanlist" is a vector indexed by new channel number containing the old
%   channel number that maps to each new location, or NaN if none does.
% "reflist" is a vector indexed by new channel number containing the
%   reference bank number to be used with each new channel. This may be [].
% "reflut" is a vector indexed by reference bank number listing the old
%   channel number to be used as a reference for each reference bank.
%   This may be [].
% "enlist" is a vector of boolean values indexed by new channel number
%   indicating which new channels are enabled. This may be [].
%
% "thismap" 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.
```

10.4 nlOpenE_parseChannelMapJSON_v5.m

```
% function maplist = nlOpenE_parseChannelMapJSONv5( jsonstruct )
%
% This parses a structure containing decoded JSON describing the
% configuration of an Open Ephys version 0.5.x channel map node.
%
% NOTE - Open Ephys labels streams in the channel map starting at 0.
% Matlab will index them in the struct array starting at 1.
%
% "jsonstruct" is a structure containing JSON data, from "jsondecode()".
%
% "maplist" is a structure array with one entry per mapping table found in
%   the original structure. The fields (per "OPENEPHYS_CHANMAP.txt") are:
%   "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.
```

10.5 nlOpenE_parseChannelMapXML_v5.m

```
% function maplist = nlOpenE_parseChannelMapXML_v5( xmlstruct )
%
% This parses a structure containing a decoded Open Ephys version 0.5.x
% configuration file, and collects any channel mapping information it can
% find.
%
% NOTE - We're listing channel maps in the order that we find them, not
% sorted by stream number.
%
% "xmlstruct" is a structure containing XML configuration data, as read by
%   "readstruct()".
%
% "maplist" is a structure array with one entry per mapping table found in
%   the configuration file. The fields (per "OPENEPHYS_CHANMAP.txt") are:
%   "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.
```

10.6 nlOpenE_probeFolder.m

```
% function foldermeta = nlOpenE_probeFolder( indir )
%
% This checks for the existence of Open Ephys format data files in the
% specified folder, and constructs a folder metadata structure if data is
% found.
%
% If no data is found, an empty structure is returned.
%
% "indir" is the directory to search.
%
% "foldermeta" is a folder metadata structure, per FOLDERMETA.txt.
```

Part III

Application Libraries

Chapter 11

Application Library Structures and Additional Notes

11.1 CHANREC.txt

```
A "channel record" is a structure with the following fields:

"folder" is the folder ID of the channel.
"bank" is the bank ID of the channel.
"chan" is the channel number of the channel.
"result" is the processing result associated with this channel.

The intention is that a vector of channel records may be returned that
describes a ranked list of channels along with statistical analysis results
associated with each channel.
```

11.2 TUNING.txt

```
Tuning parameter structures are used by several processing functions. These
structures are defined as follows.


An "artifact rejection tuning parameter" structure is used by
nlChan_applyArtifactReject(), and a default version is provided by
nlChan(getArtifactDefaults(). It has the following fields:

- "trimstart" is the number of seconds to remove at the start of the signal.
- "trimend" is the number of seconds to remove at the end of the signal.
- "ampthresh" is the high threshold for amplitude-based artifact detection.
- "amphalo" is the low threshold for amplitude-based artifact detection.
- "diffthresh" is the high threshold for derivative-based artifact detection.
```

- "diffhalo" is the low threshold for derivative-based artifact detection.
- "timehalosecs" is the additional time in seconds to squash around artifacts.
- "smoothsecs" is the time window size to use for smoothing the derivative.
- "dcsecs" is the time window size to use for DC removal.

See documentation for nlProc_removeArtifactsSigma() for further details of
the artifact rejection algorithm.

A "filter tuning parameter" structure is used by nlChan_applyFiltering(), and
a default version is provided by nlChan_getFilterDefaults(). It has the
following fields:

- "lfprate" is the sampling rate to use when generating the low-pass-filtered
  signal (LFP signal).
- "lfpcorner" is the sampling rate to use when splitting the signal into
  a low-pass-filtered signal (LFP signal) and high-pass-filtered signal
  (spike signal).
- "powerfreq" is a scalar or vector containing power line notch filter values
  to apply. This is typically the power line frequency and its harmonics.
- "dcfreq" is the corner frequency to use for the DC removal filter.

See documentation for nlProc_filterSignal() for further details of signal
filtering.

A "percentile tuning parameter" structure is used by nlChan_processChannel()
and by the "Channel Analysis Tool". It has the following fields:

- "burstrange" is a vector containing percentile values to use when looking
  for burst activity. This should be sorted in ascending order.
- "burstselectidx" is the index of the entry in "burstrange" to default to.
- "spikerange" is a vector containing percentile values to use when looking
  for spike activity. This should be sorted in ascending order.
- "spikeselectidx" is the index of the entry in "spikerange" to default to.

See documentation for nlProc_calcSkewPercentile() and documentation for
nlProc_calcSpectrumSkew() for further details of outlier identification for
activity detection.

A "spectrum tuning parameter" structure is used by nlChan_processChannel().
It has the following fields:

- "freqlow" is the minimum frequency to tabulate information for.
- "freqhigh" is the maximum frequency to tabulate information for.
- "freqsperdecade" specifies the spacing of frequency bins.

```
- "winsecs" is the time window size to use, in seconds.
- "winsteps" is the number of overlapping steps taken when advancing the
  time window. It advances "winsecs/winsteps" seconds per step.

See documentation for nlProc_calcSpectrumSkew() for further details of
outlier identification in time-frequency spectrograms.



This is the end of the file.
```

Chapter 12

“nlChan” Functions

12.1 nlChan_applyArtifactReject.m

```
% function [ newdata fracbad ] = nlChan_applyArtifactReject( ...
%   wavedata, samprate, tuningparams, keepnan )
%
% This performs truncation and artifact rejection, optionally followed by
% interpolation in the former artifact regions.
%
% "wavedata" is the waveform to process.
% "refdata" is a reference to subtract from the waveform, or [] for no
%   reference. The reference should already be truncated and have artifacts
%   removed, but should retain NaN values to avoid introducing new artifacts.
% "samprate" is the sampling rate.
% "tuningparams" is a structure containing tuning parameters for artifact
%   rejection.
% "keepnan" is true if NaN values are to remain and false if interpolation
%   is to be performed to remove them.
%
% "newdata" is the series after artifact removal.
% "fracbad" is the fraction of samples discarded as artifacts (0..1).
```

12.2 nlChan_applyFiltering.m

```
% function [ lfpseries spikeseries ] = nlChan_applyFiltering( ...
%   wavedata, samprate, tuningparams );
%
% This performs filtering to suppress power line noise, zero-average the
% signal, and to split the signal into LFP and spike components.
%
% Power line noise filtering and DC removal filtering can be suppressed by
% setting their respective filter frequencies to 0 Hz.
```

```
%
% "wavedata" is the waveform to process.
% "samprate" is the sampling rate.
% "tuningparams" is a structure containing tuning parameters for filtering.
%
% "newdata" is the series after filtering.
```

12.3 nlChan_applySpectSkewCalc.m

```
% function [ spectfreqs spectmedian spectiqr spectskew ] = ...
%   nlChan_applySpectSkewCalc( wavedata, samprate, tuningparams, perclist )
%
% This calls nlProc_calcSpectrumSkew() to compute a persistence spectrum for
% the specified series and to compute statistics and skew for each frequency
% bin.
%
% "wavedata" is the waveform to process.
% "samprate" is the sampling rate.
% "tuningparams" is a structure containing tuning parameters for persistence
%   spectrum generation.
% "perclist" is an array of percentile values that define the tails for
%   skew calculation, per nlProc_calcSkewPercentile().
%
% "spectfreqs" is an array of bin center frequencies.
% "spectmedian" is an array of per-frequency median power values.
% "spectiqr" is an array of per-frequency power interquartile ranges.
% "spectskew" is a cell array, with one cell per "perclist" value. Each cell
%   contains an array of per-frequency skew values.
```

12.4 nlChan_getArtifactDefaults.m

```
% function paramstruct = nlChan_getArtifactDefaults()
%
% This returns a structure containing reasonable default tuning parameters
% for artifact rejection.
%
% Parameters that will most often be varied are "ampthresh", "diffthresh",
% "trimstart", and "trimend".
```

12.5 nlChan_getFilterDefaults.m

```
% function paramstruct = nlChan_getFilterDefaults()
%
```

```
% This returns a structure containing reasonable default tuning parameters
% for signal filtering.
%
% Parameters that will most often be varied are "powerfreq" and "lfprate".
```

12.6 nlChan_getPercentDefaults.m

```
% function paramstruct = nlChan_getPercentDefaults()
%
% This returns a structure containing reasonable default tuning parameters
% for spike and burst identification via percentile binning.
%
% Parameters that will most often be varied are "burstselectidx" and
% "spikeselectidx".
```

12.7 nlChan_getSpectrumDefaults.m

```
% function paramstruct = nlChan_getSpectrumDefaults()
%
% This returns a structure containing reasonable default tuning parameters
% for persistence spectrum generation.
```

12.8 nlChan_processChannel.m

```
% function resultstats = nlChan_processChannel( wavedata, samprate, ...
%   refdata, tuningart, tuningfilt, tuningspect, tuningperc )
%
% This accepts a wideband waveform, performs filtering to split it into
% spike and LFP signals, and calculates various statistics for each of these
% signals.
%
% This is intended to be called by nlChan_iterateChannels() via a wrapper.
%
% "wavedata" is the waveform to process.
% "samprate" is the sampling rate.
% "refdata" is the reference waveform. This should already be truncated and
%   have artifacts removed, but should retain NaN values to avoid introducing
%   new artifacts. If refdata is [], no re-referencing is performed.
% "tuningart" is a structure containing tuning parameters for artifact removal.
% "tuningfilt" is a structure containing tuning parameters for filtering.
% "tuningspect" is a structure containing tuning parameters for persistence
%   spectrum generation.
% "tuningperc" is a structure containing tuning parameters for spike and
%   burst identification via percentile binning.
%
% "resultstats" is a structure containing the following fields:
%   "spikemedian", "spikeiqr", "spikeskew", and "spikepercentvals" are the
%     corresponding fields returned by nlProc_calcSkewPercentile() using the
%     high-pass-filtered spike signal.
%   "spikebincounts" and "spikebinedges" are the the corresponding fields
%     returned by histcounts() using a normalized version of the spike signal.
%   "spectfreqs", "spectmedian", "spectiqr", and "spectskew" are the
%     corresponding fields returned by nlChan_applySpectSkewCalc() using the
%     low-pass-filtered LFP signal.
%   "persistvals", "persistfreqs", and "persistpowers" are the corresponding
%     fields returned by pspectrum() using the LFP signal.
```

12.9 nlChan_rankChannels.m

```
% function [ bestlist typbest typmiddle typworst ] = ...
%   nlChan_rankChannels( chanresults, maxperbank, typfrac, scorefunc )
%
% This process evaluates a result list returned by nlIO_iterateChannels().
% Channels receive a score, and a list of channel records is compiled that is
% sorted by that score. Channel records for "typical" best, worst, and
% middle-scoring entries are selected, and these plus a trimmed sorted list
% are returned.
%
% Channel records have the format given in "CHANREC.txt".
%
% The sorted list is trimmed to include at most a certain number of channels
% per bank.
%
% NOTE - The "typical" records are not necessarily in the trimmed result list.
%
% "chanresults" is a channel list per "CHANLIST.txt" that has been augmented
%   with "resultlist" fields by per-channel signal processing.
% "maxperbank" is the maximum number of channels per bank in the returned list.
% "typfrac" is the percentile for finding "typical" good and bad records.
%   This is a number between 0 and 50 (typically 5, 10, or 25).
% "scorefunc" is a function handle that is called for each channel. It has
%   the form:
%     scoreval = scorefunc(resultval)
%   The "resultval" argument is a channel's "resultlist" value, per
%     nlIO_iterateChannels().
%   Higher scores are better, for purposes of this function. A score of NaN
%   squashes a result (removing it from the result list).
%
% "bestlist" is a subset of the sorted channel record list containing the
%   highest-scoring entries subject to the constraints described above.
% "typbest" is the channel record for the top Nth percentile channel.
% "typmiddle" is the channel record for the median channel.
% "typworst" is the channel record for the bottom Nth percentile channel.
```

Chapter 13

"nlCheck" Functions

13.1 nlCheck_getFTSignalBits.m

```
% function chanbits = nlCheck_getFTSignalBits( ftdata )
%
% This computes the number of bits of dynamic range in each channel and trial
% in a Field Trip dataset.
%
% The number will only be meaningful with integer data (i.e. with a minimum
% step size of 1.0).
%
% This will usually be called with single-trial continuous data to provide a
% sanity check of recording settings.
%
% "ftdata" is a ft_datatype_raw structure containing ephys data.
%
% "chanbits" is a cell array with one cell per trial, each containing a
%   Nchans x 1 floating-point vector with the number of bits needed to
%   represent each channel's data in that trial.
```

13.2 nlCheck_getSignalBits.m

```
% function signalbits = nlCheck_getSignalBits( wavedata )
%
% This computes the number of bits of dynamic range in a signal.
% The number will only be meaningful with integer data (i.e. with a minimum
% step size of 1.0).
%
% "wavedata" is a vector containing waveform data samples.
%
% "signalbits" is a floating-point value containing the number of bits
%   needed to represent the waveform data.
```

13.3 nlCheck_testDropoutsArtifacts.m

```
% function [ dropoutfrac artifactfrac ] = nlCheck_testDropoutsArtifacts( ...
%   wavedata, smoothsamps, dropout_threshold, artifact_threshold )
%
% This function computes a rectified version of the input signal, smooths it,
% finds the median amplitude, and looks for samples that are above some
% multiple of the median amplitude or below some fraction of the median
% amplitude.
%
% Samples above the high threshold are assumed to be artifacts, and samples
% below the low threshold are assumed to be drop-outs.
%
% "wavedata" is a vector containing waveform data samples.
% "smoothsamps" is the smoothing window size, in samples. This is
%   approximately one period at the low-pass filter's cutoff frequency.
% "dropout_threshold" is a multiplier determining the lower threshold to test.
%   This should be less than one.
% "artifact_threshold" is a multiplier determining the upper threshold to
%   test. This should be greater than one.
%
% "dropoutfrac" is the fraction of samples that were below the lower
%   threshold.
% "artifactfrac" is the fraction of samples that were above the upper
%   threshold.
```

13.4 nlCheck_testFTDropoutsArtifacts.m

```
% function [ dropoutfrac artifactfrac ] = nlCheck_testFTDropoutsArtifacts( ...
%   ftdata, smoothfreq, dropout_threshold, artifact_threshold )
%
% This function calls nlCheck_testDropoutsArtifacts() to compute the fraction
% of samples that are dropouts and the fraction of samples that are artifacts
% in each channel and trial in a Field Trip dataset.
%
% This is done by rectifying and smoothing the original signal and then
% thresholding the resulting signal against multiples of the median signal.
% This should be treated as an estimate only; for more accurate artifact and
% dropout detection, use more sophisticated algorithms.
%
% "ftdata" is a ft_datatype_raw structure containing ephys data.
% "smoothfreq" is the low-pass filter corner frequency to use for smoothing
%   the data prior to detection. Artifacts and dropouts shorter than 1/2pi*f
%   will be attenuated.
% "dropout_threshold" is a multiplier determining the lower threshold to test.
%   This should be less than one.
% "artifact_threshold" is a multiplier determining the upper threshold to
%   test. This should be greater than one.
%
% "dropoutfrac" is a cell array with one cell per trial, each containing a
%   Nchans x 1 floating-point vector with fraction of samples that were below
%   the lower threshold.
% "artifactfrac" is a cell array with one cell per trial, each containing a
%   Nchans x 1 floating-point vector with fraction of samples that were above
%   the upper threshold.
```

Part IV

Sample Code

Chapter 14

Sample Code

14.1 do_config_frey_silicon.m

```
% NeuroLoop Project - Test program configuration file - Frey silicon dataset
% Written by Christopher Thomas.

%
%
% Configuration.


%
% Switches.

want_unity = false;
want_trim = false;

% Recording channels can be referenced to common average, or to the ground
% channel average, or to the lumped signal and ground average, or to nothing
% (cable reference only).

frey_rec_ref = 'none';
%frey_rec_ref = 'refavg';


%
% Signal processing configuration.

% Make a guess at when driving happens.
% Ali's data drove during the first 4 minutes and last 2 minutes.
% Motor driving gives a peak from 10-12 Hz.

trimtimes = [];  % No trimming.
if want_trim
```

```
  % Trim the first 6 and last 3 minutes.
  trimtimes = [ 360 180 ];
end



%
% Dataset configuration.

% Folders to probe.
oeprefix = ...
  'datasets/20211111-frey-silicon/021-11-11_12-08-33/Record Node 101/';
folderlist = struct( ...
  'FreyRec1', [ oeprefix 'experiment1/recording1' ], ...
  'FreyRec2', [ oeprefix 'experiment2/recording1' ], ...
  'FreyStim', 'datasets/20211111-frey-silicon/stim_211111_121220' );

if want_unity
  folderlist.FreyUnity = ...
'datasets/20211111-frey-silicon/Session3__11_11_2021__12_12_49/RuntimeData';
end



% Data channels.
% The Reider recording has channels 14, 45, 47, 49, and 51 in bank A.
% Documentation for the test says A45 was CD, A47 was CD, elec11 was ACC.
% Cable reference was connected to the guide tube; no channel reference.
% Channels 14, 49, and 51 were grounded and used for artifact monitoring.

tungsten_chans_rec = [ 45 47 ];
tungsten_chans_recgnd = [ 14 49 51 ];
tungsten_chans_stim = [ 11 ];

% NOTE - We don't need to list the ground channels here.
% Reference-building and signal processing are done in separate passes.
chanlist = struct( ...
  'FreyRec', struct( ...
    'ampA', struct( 'chanlist', tungsten_chans_rec ), ...
    'Din', struct( 'chanlist', [1:16] ), ...
    'Dout', struct( 'chanlist', [1:16] ) ), ...
  'FreyStim', struct( ...
    'ampC', struct( 'chanlist', tungsten_chans_stim ), ...
    'Din', struct( 'chanlist', [1:16] ), ...
    'Dout', struct( 'chanlist', [1:16] ) ) );


% Referencing.
% Define average references for signals and for ground channels.
% FIXME - No reference for the stimulator!
% Hope that the cable reference is good enough.

refdefs = struct( ...
  'refrecord', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', tungsten_chans_rec ) ) ), ...
  'refgnd', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', tungsten_chans_recgnd ) ) ), ...
  'refall', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', [ tungsten_chans_rec tungsten_chans_recgnd ] ) ) ) ...
  );


% Our selected reference may be 'none'; if that's the case, don't store one.

if isfield(refdefs, frey_rec_ref)
  reflist = cell(size( chanlist.FreyRec.ampA.chanlist ));
  reflist(:) = { frey_rec_ref };
  chanlist.FreyRec.ampA.reflist = reflist;
end

% NOTE - We have no useful reference for the stimulation controller.



%
% Analysis tuning parameters.

% Start with the default parameters and modify as needed.

% Get default algorithm parameters.
tuningart = nlChan_getArtifactDefaults();
tuningfilt = nlChan_getFilterDefaults();
tuningspect = nlChan_getSpectrumDefaults();
tuningperc = nlChan_getPercentDefaults();


% Adjust the trimming endpoints.
if (0 < length(trimtimes))
  tuningart.trimstart = trimtimes(1);
  tuningart.trimend = trimtimes(2);
end


% Add notch filtering for power line harmonics.
%tuningfilt.powerfreq = [ 60 120 180 ];


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

14.2 do_config_frey_tungsten.m

```
% NeuroLoop Project - Test program configuration file - Frey tungsten dataset
% Written by Christopher Thomas.

%
%
% Configuration.


%
% Switches.

want_unity = false;
want_trim = false;

% Recording channels can be referenced to common average, or to the ground
% channel average, or to the lumped signal and ground average, or to nothing
% (cable reference only).

%frey_rec_ref = 'none';
%frey_rec_ref = 'refrecord';
frey_rec_ref = 'refgnd';
%frey_rec_ref = 'refall';


%
% Signal processing configuration.

% Make a guess at when driving happens.
% Ali's data drove during the first 4 minutes and last 2 minutes.
% Motor driving gives a peak from 10-12 Hz.

trimtimes = [];  % No trimming.
if want_trim
  % Trim the first 6 and last 3 minutes.
  trimtimes = [ 360 180 ];
end



%
% Dataset configuration.

% Folders to probe.
folderlist = struct( ...
  'FreyRec', 'datasets/20211112-frey-tungsten/record_211112_112922', ...
  'FreyStim', 'datasets/20211112-frey-tungsten/stim_211112_112924' );
```

```
if want_unity
  folderlist.FreyUnity = ...
'datasets/20211112-frey-tungsten/Session4__12_11_2021__11_29_57/RuntimeData/';
end


% Data channels.
% The Reider recording has channels 14, 45, 47, 49, and 51 in bank A.
% Documentation for the test says A45 was CD, A47 was CD, elec11 was ACC.
% Cable reference was connected to the guide tube; no channel reference.
% Channels 14, 49, and 51 were grounded and used for artifact monitoring.

tungsten_chans_rec = [ 45 47 ];
tungsten_chans_recgnd = [ 14 49 51 ];
tungsten_chans_stim = [ 11 ];

% NOTE - We don't need to list the ground channels here.
% Reference-building and signal processing are done in separate passes.
chanlist = struct( ...
  'FreyRec', struct( ...
    'ampA', struct( 'chanlist', tungsten_chans_rec ), ...
    'Din', struct( 'chanlist', [1:16] ), ...
    'Dout', struct( 'chanlist', [1:16] ) ), ...
  'FreyStim', struct( ...
    'ampC', struct( 'chanlist', tungsten_chans_stim ), ...
    'Din', struct( 'chanlist', [1:16] ), ...
    'Dout', struct( 'chanlist', [1:16] ) ) );


% Referencing.
% Define average references for signals and for ground channels.
% FIXME - No reference for the stimulator!
% Hope that the cable reference is good enough.

refdefs = struct( ...
  'refrecord', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', tungsten_chans_rec ) ) ), ...
  'refgnd', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', tungsten_chans_recgnd ) ) ), ...
  'refall', struct( 'FreyRec', struct( 'ampA', ...
    struct( 'chanlist', [ tungsten_chans_rec tungsten_chans_recgnd ] ) ) ) ...
  );


% Our selected reference may be 'none'; if that's the case, don't store one.

if isfield(refdefs, frey_rec_ref)
  reflist = cell(size( chanlist.FreyRec.ampA.chanlist ));
  reflist(:) = { frey_rec_ref };
  chanlist.FreyRec.ampA.reflist = reflist;
```

```
end

% NOTE - We have no useful reference for the stimulation controller.



%
% Analysis tuning parameters.

% Start with the default parameters and modify as needed.

% Get default algorithm parameters.
tuningart = nlChan_getArtifactDefaults();
tuningfilt = nlChan_getFilterDefaults();
tuningspect = nlChan_getSpectrumDefaults();
tuningperc = nlChan_getPercentDefaults();


% Adjust the trimming endpoints.
if (0 < length(trimtimes))
  tuningart.trimstart = trimtimes(1);
  tuningart.trimend = trimtimes(2);
end


% Add notch filtering for power line harmonics.
%tuningfilt.powerfreq = [ 60 120 180 ];


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

14.3 do_config_reider.m

```
% NeuroLoop Project - Test program configuration file - Reider dataset
% Written by Christopher Thomas.

%
%
% Configuration.


%
% Switches.

% Use common-average referencing instead of using the reference channel.
want_common_ref = false;
```

```
%
% Signal processing configuration.

% Ali said that driving happens in the first 4 minutes and last 2 minutes.
% Trim the first 6 and last 3, just in case.
% Actual times are shorter than Ali said for this dataset.

trimtimes = [ 360 180 ];
%trimtimes = [];  % No trimming.

% Motor driving gives a peak from 10-12 Hz.
% We're getting a broad 40-50 Hz peak that looks suspicious, but it's not
% due to the notch filter (removing that filter doesn't change the peak).



%
% Dataset configuration.

% Folders to probe.
folderlist = struct( 'reider', 'datasets/reider-20200611-02' );


% Data channels.
% The Reider recording has channels 14, 45, 47, 49, and 51 in bank A.
% Ignore channel 47; it's the reference channel.

reider_chans = [ 14 45 49 51 ];
chanlist = struct( 'reider', struct( 'ampA', ...
  struct( 'chanlist', reider_chans ) ) );

% Referencing.
% Define a common average reference and a single channel reference.

refdefs = struct( 'single', ...
  struct( 'reider', struct( 'ampA', struct( 'chanlist', 47 ) ) ), ...
  'common', ...
  struct( 'reider', struct( 'ampA', struct( 'chanlist', [0:63] ) ) ) ...
  );

desired_ref = 'single';
if want_common_ref
  desired_ref = 'common';
end

% Add reference information to the channel list.
reflist = cell(size(reider_chans));
```

```
reflist(:) = { desired_ref };
chanlist.reider.ampA.reflist = reflist;



%
% Analysis tuning parameters.

% Start with the default parameters and modify as needed.

% Get default algorithm parameters.
tuningart = nlChan_getArtifactDefaults();
tuningfilt = nlChan_getFilterDefaults();
tuningspect = nlChan_getSpectrumDefaults();
tuningperc = nlChan_getPercentDefaults();


% Adjust the trimming endpoints.
if (0 < length(trimtimes))
  tuningart.trimstart = trimtimes(1);
  tuningart.trimend = trimtimes(2);
end


% Add notch filtering for power line harmonics.
%tuningfilt.powerfreq = [ 60 120 180 ];


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

14.4 do_test.m

```
% NeuroLoop Project - Test program
% Written by Christopher Thomas.

%
% Library paths.

addPathsLoopUtil;



%
%
% Configuration.
```

```
%
% Behavior switches.


% Signal processing switches.

configflags = struct();

configflags.want_rereference = true;
configflags.want_filter = true;
configflags.want_artifactreject = true;
configflags.want_chanspikestats = true;
configflags.want_chanburststats = true;

% Plotting switches.

want_stats_chantool = false;
want_plots_chantool = true;


% Plot directory.

global plotdir;
plotdir = 'plots';


%
% Various tuning parameters.
% These may be overridden by dataset-specific configuration.

% Number of channels to load into memory concurrently.
memchans = 1;


%
% Dataset-specific configuration.

% This is expected to provide the following variables:
%
% - "folderlist" is a structure indexed by user-defined folder labels that
%   contains folder paths (i.e. a map from folder labels to path strings).
% - "chanlist" is a list of channels to process, per CHANLIST.txt. This
%   is optionally augmented with a "reflist" field specifying a reference
%   label to use with each channel.
% - "refdefs" is a structure indexed by user-defined reference labels that
%   contains channel lists (per CHANLIST.txt) defining signals to be used
%   to create references (averaged if multiple signals).
% - "tuningart" is an artifact-rejection tuning parameter structure, per
%   TUNING.txt.
% - "tuningfilt" is a filtering tuning parameter structure, per TUNING.txt.
% - "tuningperc" is a percentile statistics tuning parameter structure,
%   per TUNING.txt.
% - "tuningspect" is a time-frequency spectrogram tuning parameter structure,
%   per TUNING.txt.


% Set up defaults.

folderlist = struct();
chanlist = struct();
refdefs = struct();

tuningart = nlChan_getArtifactDefaults();
tuningfilt = nlChan_getFilterDefaults();
tuningspect = nlChan_getSpectrumDefaults();
tuningperc = nlChan_getPercentDefaults();


% Load dataset-specific changes.

do_config_reider;
%do_config_frey_tungsten;
%do_config_frey_silicon;



%
%
% Main Program


%
% Read metadata.

is_ok = true;
metadata = struct();
foldernames = fieldnames(folderlist);

disp('-- Reading metadata.');

for fidx = 1:length(foldernames)
  thisname = foldernames{fidx};
  [ folder_ok metadata ] = nlIO_readFolderMetadata( metadata, ...
    thisname, folderlist.(thisname), 'auto' );

  if ~folder_ok
    disp(sprintf( '... Failed to read from "%s"; bailing out.', ...
      folderlist.(thisname) ));
  end
  is_ok = is_ok && folder_ok;
end


%
% Build references.

% There are few enough of these that they can be stored in RAM without issue.

refseries = struct();

if is_ok && configflags.want_rereference
  disp('-- Building references.');

  % FIXME - Performance diagnostics.
  tic;

  % FIXME - Compute the reference values without any signal preprocessing.
  refpreprocfunc =  @(metadata, folderid, bankid, chanid, ...
    wavedata, timedata, wavenative, timenative) wavedata;

  refnames = fieldnames(refdefs);
  for ridx = 1:length(refnames)
    thisname = refnames{ridx};
    thisdef = refdefs.(thisname);
    refseries.(thisname) = nlProc_computeAverageSignal( ...
      metadata, thisdef, memchans, refpreprocfunc );
  end

  % FIXME - Performance diagnostics.
  disp(sprintf( '-- Reference compilation time:   %.1f s', toc ));
end


%
% Iterate channel by channel.

if is_ok

  disp('-- Processing signal data.');

  % FIXME - Performance diagnostics.
  global calctime;
  calctime = 0;

  % Reference selection is stored in chanlist (reflist field).
  % References themselves also need to be passed (refseries struct).

  % NOTE - Our helper function expects sample counts, not times, so use
  % "timenative".
  % NOTE - Our power scale expects uV rather than V, so multiply by 1e+6.
  sigprocfunc = @(metadata, folderid, bankid, chanid, ...
    wavedata, timedata, wavenative, timenative) ...
    helper_processChannel( metadata, folderid, bankid, chanid, ...
      wavedata * 1e+6, timenative, chanlist, refseries, configflags, ...
      tuningart, tuningfilt, tuningspect, tuningperc );

  resultlist = ...
    nlIO_iterateChannels( metadata, chanlist, memchans, sigprocfunc );

  % FIXME - Performance diagnostics.
  disp(sprintf( '-- Processing time:   %.1f s', calctime ));

end


%
% Generate statistics summaries and plots.


if is_ok
  % FIXME - Performance diagnostics.

  global renderspiketime;
  global renderpersisttime;
  global renderexcursiontime;

  renderspiketime = 0;
  renderpersisttime = 0;
  renderexcursiontime = 0;

  if want_stats_chantool
    disp('== Statistics:');
  end

  folderlist = fieldnames(resultlist);
  for fidx = 1:length(folderlist)
    folderid = folderlist{fidx};
    thisfolder = resultlist.(folderid);

    banklist = fieldnames(thisfolder);
    for bidx = 1:length(banklist)
      bankid = banklist{bidx};
      thisbank = thisfolder.(bankid);

      chanlist = thisbank.chanlist;
      for cidx = 1:length(chanlist)
        chanid = chanlist(cidx);
        thisresult = thisbank.resultlist{cidx};
        helper_plotChannel( thisresult, folderid, bankid, chanid, ...
          want_stats_chantool, want_plots_chantool, tuningperc );
      end
    end
  end

  if want_stats_chantool
    disp('== End of statistics.');
  end

  % FIXME - Performance diagnostics.
  if want_plots_chantool
    disp('-- Graphing time:');
    disp(sprintf( ...
      '%.1f s spike hist  %.1f s persist  %.1f s excursions', ...
      renderspiketime, renderpersisttime, renderexcursiontime ));
  end
end


disp('-- Finished.');


%
% Done.


%
%
% Private helper functions.


% This does per-channel signal processing.
% NOTE - For now, it's very much like what nlChan_processChannel() does.
% NOTE - We computed references without preprocessing (or truncating), so
% we need to do reference subtraction before those steps here too.

function resultval = helper_processChannel( ...
  metadata, folderid, bankid, chanid, ...
  wavedata, timedata, chanlist, refseries, configflags, ...
  tuningart, tuningfilt, tuningspect, tuningperc )

  % Reference selection is stored in chanlist (reflist field).
  % Reference waveforms themselves are in the refseries structure.


  % FIXME - Behavior switch. We can wrap nlChan_XX functions, or avoid them.
  want_nlchan = false;
```

```
% Initialize the result to an empty structure.
% Different processing operations add different fields to it.

resultval = struct();


% Sanity check; make sure we have this folder/bank/channel.
% Copy metadata if we do.

is_ok = false;
if isfield(metadata.folders, folderid)
  foldermeta = metadata.folders.(folderid);
  if isfield(foldermeta.banks, bankid)
    bankmeta = foldermeta.banks.(bankid);
    if ismember(chanid, bankmeta.channels)
      is_ok = true;
    end
  end
end


% Process this channel.

if is_ok

  % Banner.
  disp(sprintf( '... Processing %s-%s-%03d...', folderid, bankid, chanid ));


  % FIXME - Performance diagnostics.
  global calctime;
  tic;


  % Get this bank's sampling rate.

  samprate = bankmeta.samprate;


  % Get time trimming information.

  want_trim = false;
  if (tuningart.trimstart > 0) || (tuningart.trimend > 0)
    want_trim = true;
  end


  % Subtract the reference.
  % NOTE - We computed references without trimming or preprocessing, so
  % we need to subtract them before doing that here.
```

```
if configflags.want_rereference
  % This tolerates missing references.
  % FIXME - Blithely assume that this channel is in the channel list.

  reflabel = '';
  if isfield(chanlist.(folderid).(bankid), 'reflist')
    bankchanlist = chanlist.(folderid).(bankid);
    % We _should_ have one match. We _may_ have zero or more matches.
    refidx = (bankchanlist.chanlist == chanid);
    reflabel = bankchanlist.reflist(refidx);
    if ~isempty(reflabel)
      reflabel = reflabel{1};
    end
  end

  if ~isempty(reflabel)
    if isfield(refseries, reflabel)
      wavedata = wavedata - refseries.(reflabel);
    end
  end
end


% Truncate the time and data series.

if want_trim
  timedata = nlProc_trimEndpointsTime( ...
    timedata, samprate, tuningart.trimstart, tuningart.trimend );
  wavedata = nlProc_trimEndpointsTime( ...
    wavedata, samprate, tuningart.trimstart, tuningart.trimend );
end

% Squash the trim times after doing so, as otherwise the nlChan_XX
% artifact rejection function will do it again.
tuningart.trimstart = 0;
tuningart.trimend = 0;


% Perform artifact rejection.
% We've already subtracted the reference, so don't pass a reference here.

if configflags.want_artifactreject
  if want_nlchan
    % Use nlChan_applyArtifactReject().
    % NOTE - This does trimming as well, so squash trim times if using it.
    [ wavedata fracbad ] = nlChan_applyArtifactReject( ...
      wavedata, [], samprate, tuningart, false );
  else
    % Doing this directly (without nlChan_XX functions).
    wavedata = nlProc_removeArtifactsSigma( wavedata, ...
      tuningart.ampthresh, tuningart.diffthresh, ...
      tuningart.amphalo, tuningart.diffhalo, ...
      round(tuningart.timehalosecs * samprate), ...
      round(tuningart.timehalosecs * samprate), ...
      round(tuningart.smoothsecs * samprate), ...
      round(tuningart.dcsecs * samprate) );

    fracbad = sum(isnan(wavedata)) / length(wavedata);
    wavedata = nlProc_fillNaN(wavedata);
  end

  % FIXME - Diagnostics. Complain if this channel looks bad.
  if fracbad > 0.1
    disp(sprintf( '... NOTE - %d%% bad samples in %s-%s-%03d.', ...
      round(100*fracbad), folderid, bankid, chanid ));
  end
end


% Filter the signal to get LFP and spike waveforms.

if configflags.want_filter
  if want_nlchan
    % Use nlChan_applyFiltering().
    [ lfpseries spikeseries ] = ...
      nlChan_applyFiltering( wavedata, samprate, tuningfilt );
  else
    % Doing this directly (without nlChan_XX functions).
    [ lfpseries spikeseries ] = ...
      nlProc_filterSignal( wavedata, samprate, ...
        tuningfilt.lfprate, tuningfilt.lfpcorner, ...
        tuningfilt.powerfreq, tuningfilt.dcfreq );
  end

  % Now that we have LFP and spike waveforms, compute channel stats.

  % Compute and save per-channel spike statistics.
  if configflags.want_chanspikestats
    [ spikemedian spikeiqr spikeskew spikepercentvals ] = ...
      nlProc_calcSkewPercentile(spikeseries, tuningperc.spikerange);

    spikebinedges = -20:0.5:20;
    [ spikebincounts spikebinedges ] = ...
      histcounts(spikeseries / spikeiqr, spikebinedges);

    resultval.spikemedian = spikemedian;
    resultval.spikeiqr = spikeiqr;
    resultval.spikeskew = spikeskew;
```

```
        resultval.spikepercentvals = spikepercentvals;

        resultval.spikebincounts = spikebincounts;
        resultval.spikebinedges = spikebinedges;
      end

      % Compute and save per-channel burst statistics.
      % This includes a persistence spectrum.
      if configflags.want_chanburststats
        lfprate = tuningfilt.lfprate;

        if want_nlchan
          [ spectfreqs spectmedian spectiqr spectskew ] = ...
            nlChan_applySpectSkewCalc( lfpseries, lfprate, ...
              tuningspect, tuningperc.burstrange );
        else
          % Doing this directly (without nlChan_XX functions).
          [ spectfreqs spectmedian spectiqr spectskew ] = ...
            nlProc_calcSpectrumSkew( lfpseries, lfprate, ...
              [ tuningspect.freqlow tuningspect.freqhigh ], ...
              tuningspect.freqsperdecade, ...
              tuningspect.winsecs, tuningspect.winsteps, ...
              tuningperc.burstrange );
        end

        % There is no nlChan_XX wrapper for this.
        [ persistvals persistfreqs persistpowers ] = ...
          pspectrum( lfpseries, lfprate, 'persistence', ...
            'Leakage', 0.75, ...
            'FrequencyLimits', [tuningspect.freqlow tuningspect.freqhigh], ...
            'TimeResolution', tuningspect.winsecs );

        resultval.spectfreqs = spectfreqs;
        resultval.spectmedian = spectmedian;
        resultval.spectiqr = spectiqr;
        resultval.spectskew = spectskew;

        resultval.persistvals = persistvals;
        resultval.persistfreqs = persistfreqs;
        resultval.persistpowers = persistpowers;
      end
    end


    % FIXME - Performance diagnostics.
    calctime = calctime + toc;
  end

end
```

```
% This reports statistics and generates plots for one channel's data.

function helper_plotChannel( thisdata, thisfolder, thisbank, thischan, ...
  want_stats, want_plots, tuningperc )

  % FIXME - Plot directory.

  global plotdir;

  % FIXME - Performance diagnostics.

  global renderspiketime;
  global renderpersisttime;
  global renderexcursiontime;


  %
  % Statistics reporting.
  % NOTE - Some or all of these may be missing.

  if want_stats

    disp(sprintf( '-- Statistics for %s-%s-%03d...', ...
      thisfolder, thisbank, thischan ));

    spikerange = tuningperc.spikerange;
    burstrange = tuningperc.burstrange;

    if isfield(thisdata, 'spikeskew')
      spikeskew = thisdata.spikeskew;

      for pidx = 1:length(spikerange)
        disp(sprintf( 'Spike skew %.3f %%:   %.2f', ...
          spikerange(pidx), spikeskew(pidx) ));
      end
    end

    if isfield(thisdata, 'spectskew')
      spectskew = thisdata.spectskew;

      for pidx = 1:length(tuningperc.burstrange)
        thisspectskew = spectskew{pidx};
        disp(sprintf( 'Burst skew %.2f %%:   %.2f - %.2f', ...
          burstrange(pidx), min(thisspectskew), max(thisspectskew) ));
      end
    end

  end
  %
  % Statistics plots and time/frequency plots.
  % NOTE - Some or all of these may be missing.

  if want_plots

    disp(sprintf( ...
      '-- Plotting histograms and spectrograms for %s-%s-%03d...', ...
      thisfolder, thisbank, thischan ));


    thisfig = figure();


    casetitle = sprintf('Channel %s %s %03d', thisfolder, thisbank, thischan);
    caselabel = sprintf('%s-%s-%03d', thisfolder, thisbank, thischan);


    % FIXME - We no longer have access to the raw waveform here.
%    helper_plotSignal( samprate, lfprate, ...
%      dataseries, interpseries, lfpseries, spikeseries, ...
%      casetitle, caselabel, sprintf('%s/input', plotdir) );


    if isfield(thisdata, 'spikeiqr')
      tic;

      spikeiqr = thisdata.spikeiqr;

      % The bin edges are multiples of the IQR. Normalize the percentage
      % values to match this.

      nlPlot_plotSpikeHist( thisfig, ...
        sprintf('%s/spikes-%s-hist.png', plotdir, caselabel), ...
        thisdata.spikebincounts, thisdata.spikebinedges, ...
        thisdata.spikepercentvals / spikeiqr, tuningperc.spikerange, ...
        sprintf('%s - HPF Amplitude', casetitle) );

      renderspiketime = renderspiketime + toc;
    end


    if isfield(thisdata, 'persistvals')
      tic;

      want_log = true;

      nlPlot_plotPersist( thisfig, ...
        sprintf('%s/spect-persist-%s.png', plotdir, caselabel), ...
        thisdata.persistvals, thisdata.persistfreqs, thisdata.persistpowers, ...
        want_log, sprintf('%s - Persistence Spectrum', casetitle) );

      renderpersisttime = renderpersisttime + toc;
    end


    if isfield(thisdata, 'spectskew')
      tic;

      % Make one call to plot relative and one to plot absolute.

      nlPlot_plotExcursions( thisfig, ...
        sprintf('%s/spect-burst-rel-%s.png', plotdir, caselabel), ...
        thisdata.spectfreqs, thisdata.spectmedian, ...
        thisdata.spectiqr, thisdata.spectskew, ...
        tuningperc.burstrange, ...
        true, sprintf('%s - Power Excursions (relative)', casetitle) );

      nlPlot_plotExcursions( thisfig, ...
        sprintf('%s/spect-burst-abs-%s.png', plotdir, caselabel), ...
        thisdata.spectfreqs, thisdata.spectmedian, ...
        thisdata.spectiqr, thisdata.spectskew, ...
        tuningperc.burstrange, ...
        false, sprintf('%s - Power Excursions (absolute)', casetitle) );

      renderexcursiontime = renderexcursiontime + toc;
    end


    close(thisfig);

  end


  %
  % Done.

end



% This plots an extended signal.
% FIXME - This is quick and dirty!

function helper_plotSignal( samprate, lfprate, ...
  rawseries, cleanseries, lfpseries, spikeseries, ...
  casetitle, caselabel, obase )
```

```
colblu = [ 0.0 0.4 0.7 ];
colbrn = [ 0.8 0.4 0.1 ];
colgrn = [ 0.5 0.7 0.2 ];
colmag = [ 0.5 0.2 0.5 ];
colcyn = [ 0.3 0.7 0.9 ];

zoomlut = struct( 'limits', { [], [ 500 520 ] }, ...
  'title', { 'Full', 'Detail' }, 'label', { 'all', 'det' } );

thisfig = figure();

for zidx = 1:length(zoomlut)

  thisrange = zoomlut(zidx).limits;
  ztitle = zoomlut(zidx).title;
  zlabel = zoomlut(zidx).label;

  clf('reset');


  subplot(4,1,1);

  timeseries = 1:length(rawseries);
  timeseries = (timeseries - 1) / samprate;

  hold on;
  plot( timeseries, rawseries, 'Color', colmag );
  plot( timeseries, cleanseries, 'Color', colcyn );
  hold off;

  title(sprintf('%s - Raw - %s', casetitle, ztitle));
  xlabel('Time (s)');
  ylabel('Amplitude (a.u.)');

  if (2 == length(thisrange))
    xlim(thisrange);
  else
    xlim auto;
  end


  subplot(4,1,2);

  timeseries = 1:length(cleanseries);
  timeseries = (timeseries - 1) / samprate;

  plot( timeseries, cleanseries, 'Color', colblu );

  title(sprintf('%s - Clean - %s', casetitle, ztitle));
  xlabel('Time (s)');
  ylabel('Amplitude (a.u.)');

  if (2 == length(thisrange))
    xlim(thisrange);
  else
    xlim auto;
  end


  subplot(4,1,3);

  timeseries = 1:length(lfpseries);
  timeseries = (timeseries - 1) / lfprate;

  plot( timeseries, lfpseries, 'Color', colbrn );

  title(sprintf('%s - LFP - %s', casetitle, ztitle));
  xlabel('Time (s)');
  ylabel('Amplitude (a.u.)');

  if (2 == length(thisrange))
    xlim(thisrange);
  else
    xlim auto;
  end


  subplot(4,1,4);

  timeseries = 1:length(spikeseries);
  timeseries = (timeseries - 1) / samprate;

  plot( timeseries, spikeseries, 'Color', colgrn );

  title(sprintf('%s - Spikes - %s', casetitle, ztitle));
  xlabel('Time (s)');
  ylabel('Amplitude (a.u.)');

  if (2 == length(thisrange))
    xlim(thisrange);
  else
    xlim auto;
  end


  saveas( thisfig, sprintf('%s-%s-%s.png', obase, caselabel, zlabel) );

end

close(thisfig);
```

```
  %
  % Done.

end


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