Package ‘ravetools’

July 16, 2023

Type Package
Title Signal and Image Processing Toolbox for Analyzing Intracranial 'Electroencephalography' Data
Version 0.1.2
Language en-US
Description Implemented fast and memory-efficient 'Notch'-filter, 'Welch-periodogram', discrete wavelet transform algorithm for hours of high-resolution signals, fast 3D convolution, and image alignment; providing fundamental toolbox for 'iEEG' pipelines. Documentation and examples about 'RAVE' project are provided at <https://openwetware.org/wiki/RAVE>, and the paper by John F. Magnotti, Zhengjia Wang, Michael S. Beauchamp (2020) <doi:10.1016/j.neuroimage.2020.117341>; see 'citation(``ravetools'')' for details.

BugReports https://github.com/dipterix/ravetools/issues

URL https://dipterix.org/ravetools/
License GPL-3
Encoding UTF-8
RoxygenNote 7.2.3
Depends R (>= 4.0.0)
SystemRequirements fftw3 (libfftw3-dev (deb), or fftw-devel (rpm))
Imports graphics, stats, filearray (>= 0.1.3), Rcpp, waveslim (>= 1.8.2), signal (>= 0.7.7), pracma, digest (>= 0.6.29), splines, RNiftyReg (>= 2.7.1), Rvcg (>= 0.22.1)

LinkingTo Rcpp
Suggests fftwtools, bit64, grDevices, microbenchmark, freesurferformats, testthat

NeedsCompilation yes
Author Zhengjia Wang [aut, cre, cph], Beauchamp lab [cph],
Band-pass signals

**Description**

Band-pass signals
band_pass

Usage

band_pass1(x, sample_rate, lb, ub, domain = 1, ...)

band_pass2(
   x, 
   sample_rate, 
   lb, 
   ub, 
   order, 
   method = c("fir", "butter"), 
   direction = c("both", "forward", "backward"), 
   window = "hamming", 
   ...
)

Arguments

- **x**: input signals, numeric vector or matrix. x must be row-major if input is a matrix: each row is a channel, and each column is a time-point.
- **sample_rate**: sampling frequency
- **lb**: lower frequency bound of the band-passing filter, must be positive
- **ub**: upper frequency bound of the band-passing filter, must be greater than the lower bound and smaller than the half of sampling frequency
- **domain**: 1 if x is in time-domain, or 0 if x is in frequency domain
- **order**: the order of the filter, must be positive integer and be less than one-third of the sample rate
- **method**: filter type, choices are 'fir' and 'butter'
- **direction**: filter direction, choices are 'forward', 'backward', and 'both' directions
- **window**: window type, can be a character, a function, or a vector. For character, window is a function name in the signal package, for example, 'hanning'; for a function, window takes one integer argument and returns a numeric vector with length of that input; for vectors, window is a numeric vector of length order+1.

Value

Filtered signals, vector if x is a vector, or matrix of the same dimension as x

Examples

t <- seq(0, 1, by = 0.0005)
x <- sin(t * 0.4 * pi) + sin(t * 4 * pi) + 2 * sin(t * 120 * pi)
y1 <- band_pass1(x, 2000, 0.1, 1)
```r
y2 <- band_pass1(x, 2000, 1, 5)
y3 <- band_pass1(x, 2000, 10, 80)

order <- floor(2000 / 3)
z1 <- band_pass2(x, 2000, 0.1, 1, method = "fir", order = order)
z2 <- band_pass2(x, 2000, 1, 5, method = "fir", order = order)
z3 <- band_pass2(x, 2000, 10, 80, method = "fir", order = order)

par(mfrow = c(1, 2))
plot(t, x, type = 'l', xlab = "Time", ylab = "
    main = "Mixture of 0.2, 2, and 60Hz")
lines(t, y1, col = "red")
lines(t, y2, col = "blue")
lines(t, y3, col = "green")
legend(  "topleft", c("Input", "Pass: 0.1-1Hz", "Pass 1-5Hz", "Pass 10-80Hz"),
    col = c(par("fg"), "red", "blue", "green"), lty = 1
)

plot(t, x, type = 'l', xlab = "Time", ylab = "
    main = "Mixture of 0.2, 2, and 60Hz")
lines(t, z1, col = "red")
lines(t, z2, col = "blue")
lines(t, z3, col = "green")
legend(  "topleft", c("Input", "Pass: 0.1-1Hz", "Pass 1-5Hz", "Pass 10-80Hz"),
    col = c(par("fg"), "red", "blue", "green"), lty = 1
)
```

### baseline_array

**Calculate Contrasts of Arrays in Different Methods**

**Description**

Provides five methods to baseline an array and calculate contrast.

**Usage**

```r
baseline_array(x, along_dim, unit_dims = seq_along(dim(x))[-along_dim], ...)
```

```r
## S3 method for class 'array'
baseline_array(  
    x,
    along_dim,
    unit_dims = seq_along(dim(x))[-along_dim],
    method = c("percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore",
```
baseline_array

"subtract_mean"),
baseline_indexpoints = NULL,
baseline_subarray = NULL,
...
)

Arguments

x array (tensor) to calculate contrast
along_dim integer range from 1 to the maximum dimension of x. baseline along this
dimension, this is usually the time dimension.
unit_dims integer vector, baseline unit: see Details.
... passed to other methods
method character, baseline method options are: "percentage", "sqrt_percentage",
"decibel", "zscore", and "sqrt_zscore"
baseline_indexpoints integer vector, which index points are counted into baseline window? Each
index ranges from 1 to dim(x)[[along_dim]]. See Details.
baseline_subarray sub-arrays that should be used to calculate baseline; default is NULL (automati-
cally determined by baseline_indexpoints).

Details

Consider a scenario where we want to baseline a bunch of signals recorded from different locations.
For each location, we record n sessions. For each session, the signal is further decomposed into
frequency-time domain. In this case, we have the input x in the following form:

\text{session}\times\text{frequency}\times\text{time}\times\text{location}

Now we want to calibrate signals for each session, frequency and location using the first 100 time
points as baseline points, then the code will be

\text{baseline_array}(x, along_dim = 3, baseline_window = 1:100, unit_dims = c(1, 2, 4))

along_dim=3 is dimension of time, in this case, it’s the third dimension of x. baseline_indexpoints=1:100,
meaning the first 100 time points are used to calculate baseline. unit_dims defines the unit signal.
Its value c(1,2,4) means the unit signal is per session (first dimension), per frequency (second)
and per location (fourth).

In some other cases, we might want to calculate baseline across frequencies then the unit signal is
frequency\times time, i.e. signals that share the same session and location also share the same baseline.
In this case, we assign unit_dims=c(1,4).

There are five baseline methods. They fit for different types of data. Denote \( z \) is an unit signal, \( z_0 \)
is its baseline slice. Then these baseline methods are:

"percentage"

\[ \frac{z - z_0}{z_0} \times 100\% \]
baseline_array

"sqrt_percentage"
\[
\sqrt{z} - \sqrt{z_0} \times 100\%
\]

"decibel"
\[
10 \times (\log_{10}(z) - \log_{10}(z_0))
\]

"zscore"
\[
\frac{z - z_0}{sd(z_0)}
\]

"sqrt_zscore"
\[
\frac{\sqrt{z} - \sqrt{z_0}}{sd(\sqrt{z_0})}
\]

Value
Contrast array with the same dimension as x.

Examples

# Set ncores = 2 to comply to CRAN policy. Please don't run this line
ravetools_threads(n_threads = 2L)

library(ravetools)
set.seed(1)

# Generate sample data
dims = c(10,20,30,2)
x = array(rnorm(prod(dims))^2, dims)

# Set baseline window to be arbitrary 10 timepoints
baseline_window = sample(30, 10)

# ----- baseline percentage change -----
# Using base functions
re1 <- aperm(apply(x, c(1,2,4), function(y){
  m <- mean(y[baseline_window])
  (y/m - 1) * 100
}), c(2,3,1,4))

# Using ravetools
re2 <- baseline_array(x, 3, c(1,2,4),
  baseline_indexpoints = baseline_window,
  method = 'percentage')

# Check different, should be very tiny (double precisions)
range(re2 - re1)

# Check speed for large dataset
if(interactive()){
  ravetools_threads(n_threads = -1)

dims <- c(200,20,300,2)
x <- array(rnorm(prod(dims))^2, dims)
# Set baseline window to be arbitrary 10 timepoints
baseline_window <- seq_len(100)
f1 <- function()
  aperm(apply(x, c(1,2,4), function(y){
    m <- mean(y[baseline_window])
    (y/m - 1) * 100
  })), c(2,3,1,4))
}
f2 <- function(){
  # equivalent as bl = x[,,baseline_window, ]
  # baseline_array(x, along_dim = 3,
  # baseline_indexpoints = baseline_window,
  # unit_dims = c(1,2,4), method = 'percentage')
}
range(f1() - f2())
microbenchmark::microbenchmark(f1(), f2(), times = 10L)

---

**collapse**  

**Collapse array**

**Description**

Collapse array

**Usage**

```r
collapse(x, keep, ...)
```

## S3 method for class 'array'
collapse(
  x,
  keep,
  average = TRUE,
  transform = c("asis", "10log10", "square", "sqrt"),
  ...
)
```
**Arguments**

- **x**: A numeric multi-mode tensor (array), without NA
- **keep**: Which dimension to keep
- **average**: collapse to sum or mean
- **transform**: transform on the data before applying collapsing; choices are 'asis' (no change), '10log10' (used to calculate decibel), 'square' (sum-squared), 'sqrt' (square-root and collapse)

**Value**

A collapsed array with values to be mean or summation along collapsing dimensions

**Examples**

```r
# Set ncores = 2 to comply to CRAN policy. Please don't run this line
ravetools_threads(n_threads = 2L)

# Example 1
x = matrix(1:16, 4)
# Keep the first dimension and calculate sums along the rest
collapse(x, keep = 1)
rowMeans(x)  # Should yield the same result

# Example 2
x = array(1:120, dim = c(2,3,4,5))
result = collapse(x, keep = c(3,2))
compare = apply(x, c(3,2), mean)
sum(abs(result - compare))  # The same, yield 0 or very small number (1e-10)

if(interactive()){
  ravetools_threads(n_threads = -1)
  # Example 3 (performance)
  # Small data, no big difference
  x = array(rnorm(240), dim = c(4,5,6,2))
  microbenchmark::microbenchmark(
    result = collapse(x, keep = c(3,2)),
    compare = apply(x, c(3,2), mean),
    times = 1L, check = function(v){
      max(abs(range(do.call('-', v)))) < 1e-10
    })
}

# large data big difference
x = array(rnorm(prod(300,200,105)), c(300,200,105,1))
```
convolve

```r
microbenchmark::microbenchmark(
  result = collapse(x, keep = c(3,2)),
  compare = apply(x, c(3,2), mean),
  times = 1L , check = function(v){
    max(abs(range(do.call('-', v)))) < 1e-10
  }
)
```

---

**convolve**  
*Convolution of 1D, 2D, 3D data via FFT*

**Description**
Use the 'Fast-Fourier' transform to compute the convolutions of two data with zero padding.

**Usage**
- `convolve_signal(x, filter)`
- `convolve_image(x, filter)`
- `convolve_volume(x, filter)`

**Arguments**
- `x` one-dimensional signal vector, two-dimensional image, or three-dimensional volume; numeric or complex
- `filter` kernel with the same number of dimensions as `x`

**Details**
This implementation uses 'Fast-Fourier' transform to perform 1D, 2D, or 3D convolution. Compared to implementations using original mathematical definition of convolution, this approach is much faster, especially for image and volume convolutions.

The input `x` is zero-padded beyond edges. This is most common in image or volume convolution, but less optimal for periodic one-dimensional signals. Please use other implementations if non-zero padding is needed.

The convolution results might be different to the ground truth by a precision error, usually at `1e-13` level, depending on the 'FFTW3' library precision and implementation.

**Value**
Convolution results with the same length and dimensions as `x`. If `x` is complex, results will be complex, otherwise results will be real numbers.
Examples

```r
# ---- 1D convolution ------------------------------------
x <- cumsum(rnorm(100))
filter <- dnorm(-2:2)
# normalize
filter <- filter / sum(filter)
smoothed <- convolve_signal(x, filter)

plot(x, pch = 20)
lines(smoothed, col = "red")

# ---- 2D convolution ------------------------------------
x <- array(0, c(100, 100))
x[
  floor(runif(10, min = 1, max = 100)),
  floor(runif(10, min = 1, max = 100))
] <- 1

# smooth
kernel <- outer(dnorm(-2:2), dnorm(-2:2), FUN = "*")
kernel <- kernel / sum(kernel)
y <- convolve_image(x, kernel)

par(mfrow = c(1,2))
image(x, asp = 1, axes = FALSE, main = "Origin")
image(y, asp = 1, axes = FALSE, main = "Smoothed")
```

decimate

Decimate with 'FIR' or 'IIR' filter

Description

Decimate with 'FIR' or 'IIR' filter

Usage

decimate(x, q, n = if (ftype == "iir") 8 else 30, ftype = "fir")

Arguments

- **x**: signal to be decimated
- **q**: integer factor to down-sample by
- **n**: filter order used in the down-sampling; default is 30 if ftype='fir', or 8 if ftype='iir'
- **ftype**: filter type, choices are 'fir' (default) and 'iir'
**detrend**

Remove the trend for one or more signals

### Details

This function is migrated from signal package, but with bugs fixed on 'FIR' filters. The result agrees with 'Matlab' decimate function with 'FIR' filters. Under 'IIR' filters, the function is identical with signal::decimate, and is slightly different with 'Matlab' version.

### Value

Decimated signal

### Examples

```r
x <- 1:100
y <- decimate(x, 2, ftype = "fir")
y

# compare with signal package
z <- signal::decimate(x, 2, ftype = "fir")

# Compare decimated results
plot(x, type = "l")
points(seq(1, 100, 2), y, col = "green")
points(seq(1, 100, 2), z, col = "red")
```

---

**detrend**

Remove the trend for one or more signals

### Description

'Detrending' is often used before the signal power calculation.

### Usage

```r
detrend(x, trend = c("constant", "linear"), break_points = NULL)
```

### Arguments

- **x**
  - numerical or complex, a vector or a matrix
  
- **trend**
  - the trend of the signal; choices are 'constant' and 'linear'
  
- **break_points**
  - integer vector, or NULL; only used when trend is 'linear' to remove piecewise linear trend; will throw warnings if trend is 'constant'

### Value

The signals with trend removed in matrix form; the number of columns is the number of signals, and number of rows is length of the signals
Examples

```r
x <- rnorm(100, mean = 1) + c(
  seq(0, 5, length.out = 50),
  seq(5, 3, length.out = 50))
plot(x)

plot(detrend(x, 'constant'))
plot(detrend(x, 'linear'))
plot(detrend(x, 'linear', 50))
```

diagnose_channel

Show channel signals with diagnostic plots

Description

The diagnostic plots include 'Welch Periodogram' (`pwelch`) and histogram (`hist`).

Usage

```r
diagnose_channel(
  s1,
  s2 = NULL,
  sc = NULL,
  srate,
  name = "",
  try_compress = TRUE,
  max_freq = 300,
  window = ceiling(srate * 2),
  noverlap = window/2,
  std = 3,
  which = NULL,
  main = "Channel Inspection",
  col = c("black", "red"),
  cex = 1.2,
  cex.lab = 1,
  lwd = 0.5,
  plim = NULL,
  nclass = 100,
  start_time = 0,
  boundary = NULL,
  mar = c(3.1, 4.1, 2.1, 0.8) * (0.25 + cex * 0.75) + 0.1,
  mgp = cex * c(2, 0.5, 0),
  xaxs = "i",
  yaxs = "i",
  xline = 1.66 * cex,
)```
Arguments

- **s1**: the main signal to draw
- **s2**: the comparing signal to draw; usually s1 after some filters; must be in the same sampling rate with s1; can be NULL
- **sc**: decimated s1 to show if srate is too high; will be automatically generated if NULL
- **srate**: sampling rate
- **name**: name of s1, or a vector of two names of s1 and s2 if s2 is provided
- **try_compress**: whether try to compress (decimate) s1 if srate is too high for performance concerns
- **max_freq**: the maximum frequency to display in 'Welch Periodograms'
- **window, noverlap**: see `pwelch`
- **std**: the standard deviation of the channel signals used to determine boundary; default is plus-minus 3 standard deviation
- **which**: NULL or integer from 1 to 4; if NULL, all plots will be displayed; otherwise only the subplot will be displayed
- **main**: the title of the signal plot
- **col**: colors of s1 and s2
- **cex, lwd, mar, cex.lab, mgp, xaxs, yaxs, tck, ...**: graphical parameters; see `par`
- **plim**: the y-axis limit to draw in 'Welch Periodograms'
- **nclass**: number of classes to show in histogram (`hist`)
- **start_time**: the starting time of channel (will only be used to draw signals)
- **boundary**: a red boundary to show in channel plot; default is to be automatically determined by std
- **xline, yline**: distance of axis labels towards ticks

Value

A list of boundary and y-axis limit used to draw the channel

Examples

```r
library(ravetools)

# Generate 20 second data at 2000 Hz
time <- seq(0, 20, by = 1 / 2000)
signal <- sin(120 * pi * time) +
```
\begin{verbatim}
sin(time * 20*pi) +
exp(-time^2) *
cos(time * 10*pi) +
runif(length(time))
\end{verbatim}

\begin{verbatim}
signal2 <- notch_filter(signal, 2000)
diagnose_channel(signal, signal2, srate = 2000,
    name = c("Raw", "Filtered"), cex = 1)
\end{verbatim}

\section*{fast_cov}
\textit{Calculate massive covariance matrix in parallel}

\section*{Description}
Speed up covariance calculation for large matrices. The default behavior is the same as \texttt{cov} ('pearson', no NA handling).

\section*{Usage}
\texttt{fast_cov(x, y = NULL, col_x = NULL, col_y = NULL, df = NA)}

\section*{Arguments}
\begin{itemize}
\item \texttt{x} \hspace{1cm} a numeric vector, matrix or data frame; a matrix is highly recommended to maximize the performance
\item \texttt{y} \hspace{1cm} NULL (default) or a vector, matrix or data frame with compatible dimensions to \texttt{x}; the default is equivalent to \texttt{y = x}
\item \texttt{col_x} \hspace{1cm} integers indicating the subset indices (columns) of \texttt{x} to calculate the covariance, or NULL to include all the columns; default is NULL
\item \texttt{col_y} \hspace{1cm} integers indicating the subset indices (columns) of \texttt{y} to calculate the covariance, or NULL to include all the columns; default is NULL
\item \texttt{df} \hspace{1cm} a scalar indicating the degrees of freedom; default is \texttt{nrow(x)-1}
\end{itemize}

\section*{Value}
A covariance matrix of \texttt{x} and \texttt{y}. Note that there is no NA handling. Any missing values will lead to NA in the resulting covariance matrices.

\section*{Examples}
\begin{verbatim}
# Set ncores = 2 to comply to CRAN policy. Please don't run this line
ravetools_threads(n_threads = 2L)

x <- matrix(rnorm(400), nrow = 100)
\end{verbatim}
### fast_quantile

Compute quantiles

**Description**

Compute quantiles

**Usage**

```r
fast_quantile(x, prob = 0.5, na.rm = FALSE, ...)  
fast_median(x, na.rm = FALSE, ...)  
fast_mvquantile(x, prob = 0.5, na.rm = FALSE, ...)  
fast_mvmedian(x, na.rm = FALSE, ...)
```

**Arguments**

- `x` numerical-value vector for `fast_quantile` and `fast_median`, and column-major matrix for `fast_mvquantile` and `fast_mvmedian`
- `prob` a probability with value from 0 to 1
na.rm
logical; if true, any NA are removed from x before the quantiles are computed

... reserved for future use

Value

`fast_quantile` and `fast_median` calculate univariate quantiles (single-value return); `fast_mvquantile` and `fast_mvmedian` calculate multivariate quantiles (for each column, result lengths equal to the number of columns).

Examples

```r
fast_quantile(runif(1000), 0.1)
fast_median(1:100)

x <- matrix(rnorm(100), ncol = 2)
fast_mvquantile(x, 0.2)
fast_mvmedian(x)

# Compare speed for vectors (usually 30% faster)
x <- rnorm(10000)
microbenchmark::microbenchmark(
  fast_median = fast_median(x),
  base_median = median(x),
  # bioc_median = Biobase::rowMedians(matrix(x, nrow = 1)),
  times = 100, unit = "milliseconds"
)

# Multivariate cases
# (5-7x faster than base R)
# (3-5x faster than Biobase rowMedians)
x <- matrix(rnorm(100000), ncol = 20)
microbenchmark::microbenchmark(
  fast_median = fast_mvmedian(x),
  base_median = apply(x, 2, median),
  # bioc_median = Biobase::rowMedians(t(x)),
  times = 10, unit = "milliseconds"
)
```

fill_surface
Fill a volume cube based on water-tight surface

Description

Create a cube volume (256 'voxels' on each margin), fill in the 'voxels' that are inside of the surface.
`fill_surface`  

**Usage**

```r
fill_surface(
  surface,
  inflate = 0,
  IJK2RAS = NULL,
  preview = FALSE,
  preview_frame = 128
)
```

**Arguments**

- `surface`: a surface mesh, can be mesh objects from `rgl` or `freesurferformats` packages
- `inflate`: amount of 'voxels' to inflate on the final result; must be a non-negative integer. A zero `inflate` value means the resulting volume is tightly close to the surface
- `IJK2RAS`: volume 'IJK' (zero-indexed coordinate index) to 'tkrRAS' transform, default is automatically determined leave it `NULL` if you don’t know how to set it
- `preview`: whether to preview the results; default is `false`
- `preview_frame`: integer from 1 to 256 the depth frame used to generate preview.

**Details**

This function creates a volume (256 on each margin) and fill in the volume from a surface mesh. The surface vertex points will be embedded into the volume first. These points may not be connected together, hence for each 'voxel', a cube patch will be applied to grow the volume. Then, the volume will be bucket-filled from a corner, forming a negated mask of "outside-of-surface" area. The inverted bucket-filled volume is then shrunk so the mask boundary tightly fits the surface.

**Value**

A list containing the filled volume and parameters used to generate the volume.

**Author(s)**

Zhengjia Wang

**Examples**

```r
if(interactive()) { # takes > 5s to run example

  # Generate a sphere
  surface <- Rvcg::vcgSphere()
  surface$vb[1:3, ] <- surface$vb[1:3, ] * 50

  fill_surface(surface, preview = TRUE)
}
```
filter_signal  
*Filter one-dimensional signal*

**Description**

The function is written from the scratch. The result has been compared against the 'Matlab' filter function with one-dimensional real inputs. Other situations such as matrix b or multi-dimensional x are not implemented.

**Usage**

`filter_signal(b, a, x, z)`

**Arguments**

- `b` one-dimensional real numerical vector, the moving-average coefficients of an ARMA filter
- `a` the auto-regressive (recursive) coefficients of an ARMA filter
- `x` numerical vector input (real value)
- `z` initial condition, must have length of `n-1`, where `n` is the maximum of lengths of `a` and `b`; default is all zeros

**Value**

A list of two vectors: the first vector is the filtered signal; the second vector is the final state of `z`

**Examples**

```r
t <- seq(0, 1, by = 0.01)
x <- sin(2 * pi * t * 2.3)
bf <- signal::butter(2, c(0.15, 0.3))
res <- filter_signal(bf$b, bf$a, x)
y <- res[[1]]
z <- res[[2]]

## Matlab (2022a) equivalent:
# t = [0:0.01:1];
# x = sin(2 * pi * t * 2.3);
# [b,a] = butter(2,[0.15,.3]);
# [y,z] = filter(b, a, x)
```
filtfilt

Forward and reverse filter a one-dimensional signal

Description

The result has been tested against 'Matlab' filtfilt function. Currently this function only supports one filter at a time.

Usage

```r
filtfilt(b, a, x)
```

Arguments

- `b`: one-dimensional real numerical vector, the moving-average coefficients of an ARMA filter
- `a`: the auto-regressive (recursive) coefficients of an ARMA filter
- `x`: numerical vector input (real value)

Value

The filtered signal, normally the same length as the input signal `x`.

Examples

```r
t <- seq(0, 1, by = 0.01)
x <- sin(2 * pi * t * 2.3)
bf <- signal::butter(2, c(0.15, 0.3))
res <- filtfilt(bf$b, bf$a, x)
## Matlab (2022a) equivalent:
# t = [0:0.01:1];
# x = sin(2 * pi * t * 2.3);
# [b,a] = butter(2,[.15,.3]);
# res = filtfilt(b, a, x)
```
grow_volume

Grow volume mask

**Usage**

```r
grow_volume(volume, x, y = x, z = x, threshold = 0.5)
```

**Arguments**

- `volume` : volume mask array, must be 3-dimensional array
- `x, y, z` : size of grow along each direction
- `threshold` : threshold after convolution

**Examples**

```r
par(mfrow = c(2,3), mar = c(0.1,0.1,3.1,0.1))
mask <- array(0, c(21,21,21))
mask[11,11,11] <- 1
image(mask[11,,], asp = 1,
      main = "Original mask", axes = FALSE)
image(grow_volume(mask, 2)[11,,], asp = 1,
      main = "Dilated (size=2) mask", axes = FALSE)
image(grow_volume(mask, 5)[11,,], asp = 1,
      main = "Dilated (size=5) mask", axes = FALSE)

mask[11, sample(11,2), sample(11,2)] <- 1
image(mask[11,,], asp = 1,
      main = "Original mask", axes = FALSE)
image(grow_volume(mask, 2)[11,,], asp = 1,
      main = "Dilated (size=2) mask", axes = FALSE)
image(grow_volume(mask, 5)[11,,], asp = 1,
      main = "Dilated (size=5) mask", axes = FALSE)
```
interpolate_stimulation

Find and interpolate stimulation signals

Description
Find and interpolate stimulation signals

Usage

```r
interpolate_stimulation(
  x,
  sample_rate,
  duration = 40/sample_rate,
  ord = 4L,
  nknots = 100,
  nsd = 1,
  nstim = NULL,
  regularization = 0.5
)
```

Arguments
- **x**: numerical vector representing a analog signal
- **sample_rate**: sampling frequency
- **duration**: time in second: duration of interpolation
- **ord**: spline order, default is 4
- **nknots**: a rough number of knots to use, default is 100
- **nsd**: number of standard deviation to detect stimulation signals, default is 1
- **nstim**: number of stimulation pulses, default is to auto-detect
- **regularization**: regularization parameter in case of inverting singular matrices, default is 0.5

Value
Interpolated signal with an attribute of which sample points are interpolated

Examples

```r
x0 <- rnorm(1000) / 5 + sin(1:1000 / 300)
# Simulates pulse signals
x <- x0
x[400:410] <- -100
x[420:430] <- 100
```
fitted <- interpolate_stimulation(x, 100, duration = 0.3, nknots = 10, nsd = 2)

par(mfrow = c(2, 1))

plot(fitted, type = 'l', col = 'blue', lwd = 2)
lines(x, col = 'red')
lines(x0, col = 'black')
legend("topleft", c("Interpolated", "Observed", "Underlying"),
      lty = 1, col = c("blue", "red", "black"))
pwelch(x0, 100, 200, 100, plot = 1, col = 'black', ylim = c(-50, 50))
pwelch(x, 100, 200, 100, plot = 2, col = 'red')
pwelch(fitted, 100, 200, 100, plot = 2, col = 'blue')

---

**matlab_palette**  
'Matlab' heat-map plot palette

**Description**

'Matlab' heat-map plot palette

**Usage**

matlab_palette()

**Value**

vector of 64 colors

---

**mesh_from_volume**  
Generate 3D mesh surface from volume data

**Description**

Internally calls *vcgIsosurface*, optionally calls *vcgUniformRemesh* and *vcgSmooth*.

**Usage**

mesh_from_volume(
  volume,
  output_format = c("rgl", "freesurfer"),
  IJK2RAS = NULL,
  threshold = 0,
  verbose = TRUE,
  remesh = TRUE,
`mesh_from_volume`

```r
remesh_voxel_size = 1,
remesh_multisample = TRUE,
remesh_automerge = TRUE,
smooth = FALSE,
smooth_lambda = 10,
smooth_delta = 20,
smooth_method = "surfPreserveLaplace"
```

**Arguments**

- **volume**: 3-dimensional volume array
- **output_format**: resulting data format, choices are 'rgl' and 'freesurfer'
- **IJK2RAS**: volume 'IJK' (zero-indexed coordinate index) to 'tkrRAS' transform, default is automatically determined
- **threshold**: threshold used to create volume mask; the surface will be created to fit the mask boundaries
- **verbose**: whether to verbose the progress
- **remesh**: whether to re-sample the mesh using `vcgUniformRemesh`
- **remesh_voxel_size**, **remesh_multisample**, **remesh_automerge**: see arguments in `vcgUniformRemesh`
- **smooth**: whether to smooth the mesh via `vcgSmooth`
- **smooth_lambda**, **smooth_delta**, **smooth_method**: see `vcgSmooth`

**Value**

A 'mesh3d' surface if `output_format` is 'rgl', or 'fs.surface' surface otherwise.

**Examples**

```r
volume <- array(0, dim = c(8,8,8))
volume[4:5, 4:5, 4:5] <- 1

graphics::image(x = volume[4,,])

# you can use rgl::wire3d(mesh) to visualize the mesh
mesh <- mesh_from_volume(volume, verbose = FALSE)
```
multitaper

Compute 'multitaper' spectral densities of time-series data

Description

Compute 'multitaper' spectral densities of time-series data

Usage

multitaper_config(
  data_length,
  fs,
  frequency_range = NULL,
  time_bandwidth = 5,
  num_tapers = NULL,
  window_params = c(5, 1),
  nfft = NA,
  detrend_opt = "linear"
)

multitaper(
  data,
  fs,
  frequency_range = NULL,
  time_bandwidth = 5,
  num_tapers = NULL,
  window_params = c(5, 1),
  nfft = NA,
  detrend_opt = "linear"
)

Arguments

data_length  length of data
fs   sampling frequency in 'Hz'
frequency_range  frequency range to look at; length of two
time_bandwidth  a number indicating time-half bandwidth product; i.e. the window duration times the half bandwidth of main lobe; default is 5
num_tapers  number of 'DPSS' tapers to use; default is NULL and will be automatically computed from floor(2*time_bandwidth - 1)
window_params  vector of two numbers; the first number is the window size in seconds; the second number if the step size; default is c(5, 1)
nfft  'NFFT' size, positive; see 'Details'
**multitaper**

```r
multitaper_config
```

**detrend_opt** how you want to remove the trend from data window; options are 'linear' (default), 'constant', and 'off'

**data** numerical vector, signal traces

**Details**

The original source code comes from 'Prerau' Lab (see 'Github' repository 'multitaper_toolbox' under user 'preraulab'). The results tend to agree with their 'Python' implementation with precision on the order of at $10^{-7}$ with standard deviation at most $10^{-5}$. The original copy was licensed under a Creative Commons Attribution 'NC'-'SA' 4.0 International License ([https://creativecommons.org/licenses/by-nc-sa/4.0/](https://creativecommons.org/licenses/by-nc-sa/4.0/)).

This package ('ravetools') redistributes the multitaper function under minor modifications on `nfft`. In the original copy there is no parameter to control the exact numbers of `nfft`, and the `nfft` is always the power of 2. While choosing `nfft` to be the power of 2 is always recommended, the modified code allows other choices.

**Value**

`multitaper_config` returns a list of configuration parameters for the filters; `multitaper` also returns the time, frequency and corresponding spectral power.

**Examples**

```r
if(interactive()) {

    time <- seq(0, 3, by = 0.001)
    x <- sin(time * 20*pi) + exp(-time^2) * cos(time * 10*pi)

    res <- multitaper(
        x, 1000, frequency_range = c(0,15),
        time_bandwidth=1.5,
        window_params=c(2,0.01)
    )

    image(
        x = res$time,
        y = res$frequency,
        z = 10 * log10(res$spec),
        xlab = "Time (s)",
        ylab = 'Frequency (Hz)',
        col = matlab_palette()
    )
}
```
notch_filter (Apply 'Notch' filter)

**Description**

Apply 'Notch' filter

**Usage**

```r
notch_filter(
  s,
  sample_rate,
  lb = c(59, 118, 178),
  ub = c(61, 122, 182),
  domain = 1
)
```

**Arguments**

- `s` numerical vector if domain=1 (voltage signals), or complex vector if domain=0
- `sample_rate` sample rate
- `lb` filter lower bound of the frequencies to remove
- `ub` filter upper bound of the frequencies to remove; shares the same length as lb
- `domain` 1 if the input signal is in the time domain, 0 if it is in the frequency domain

**Details**

Mainly used to remove electrical line frequencies at 60, 120, and 180 Hz.

**Value**

filtered signal in time domain (real numerical vector)

**Examples**

```r
time <- seq(0, 3, 0.005)
s <- sin(120 * pi * time) + rnorm(length(time))

# Welch periodogram shows a peak at 60Hz
pwelch(s, 200, plot = 1, log = "y")

# notch filter to remove 60Hz
s1 <- notch_filter(s, 200, lb = 59, ub = 61)
pwelch(s1, 200, plot = 2, log = "y", col = "red")
```
**parallel-options**

*Set or get thread options*

**Description**

Set or get thread options

**Usage**

```r
detect_threads()

ravetools_threads(n_threads = "auto", stack_size = "auto")
```

**Arguments**

- `n_threads`: number of threads to set
- `stack_size`: Stack size (in bytes) to use for worker threads. The default used for "auto" is 2MB on 32-bit systems and 4MB on 64-bit systems.

**Value**

- `detect_threads` returns an integer of default threads that is determined by the number of CPU cores; `ravetools_threads` returns nothing.

**Examples**

```r
if(interactive()){
  detect_threads()
  ravetools_threads(n_threads = 2)
}
```

---

**plot_signals**

*Plot one or more signal traces in the same figure*

**Description**

Plot one or more signal traces in the same figure
Usage

plot_signals(
  signals,
  sample_rate = 1,
  col = graphics::par("fg"),
  space = 0.995,
  space_mode = c("quantile", "absolute"),
  start_time = 0,
  duration = NULL,
  compress = TRUE,
  channel_names = NULL,
  time_shift = 0,
  xlab = "Time (s)",
  ylab = "Electrode",
  lwd = 0.5,
  new_plot = TRUE,
  xlim = NULL,
  cex = 1,
  cex.lab = 1,
  mar = c(3.1, 2.1, 2.1, 0.8) * (0.25 + cex * 0.75) + 0.1,
  mgp = cex * c(2, 0.5, 0),
  xaxs = "r",
  yaxs = "i",
  xline = 1.5 * cex,
  yline = 1 * cex,
  tck = -0.005 * (3 + cex),
  ...
)

Arguments

signals  numerical matrix with each row to be a signal trace and each column contains the signal values at a time point
sample_rate  sampling frequency
col  signal color, can be vector of one or more
space  vertical spacing among the traces; for values greater than 1, the spacing is absolute; default is 0.995; for values less equal to 1, this is the percentile of the whole data. However, the quantile mode can be manually turned off is "absolute" is required; see space_mode
space_mode  mode of spacing, only used when space is less equal to one; default is quantile
start_time  the time to start drawing relative to the first column
duration  duration of the signal to draw
compress  whether to compress signals if the data is too large
channel_names  NULL or a character vector of channel names
time_shift  the actual start time of the signal. Unlike start_time, this should be the actual physical time represented by the first column
pwelch

{xlab, ylab, lwd, xlim, cex, cex.lab, mar, mgp, xaxs, yaxs, tck, ...}

plot parameters; see plot and par

new_plot  whether to draw a new plot; default is true

xline, yline  the gap between axis and label

Examples

n <- 1000
base_signal <- c(rep(0, n/2), sin(seq(0,10,length.out = n/2))) * 10
signals <- rbind(rnorm(n) + base_signal,
                 rbinom(n, 10, 0.3) + base_signal,
                 rt(n, 5) + base_signal)
plot_signals(signals, sample_rate = 100)
plot_signals(signals, sample_rate = 100, start_time = 5)
plot_signals(signals, sample_rate = 100, start_time = 5, time_shift = 100)

Description

pwelch is for single signal trace only; mv_pwelch is for multiple traces. Currently mv_pwelch is experimental and should not be called directly.

Usage

pwelch(
  x,
  fs,
  window = 64,
  noverlap = 8,
  nfft = 256,
  col = "black",
  xlim = NULL,
  ylim = NULL,
  main = "Welch periodogram",
  plot = 0,
  log = c("xy", ",", "x", "y"),
  ...
)

## S3 method for class 'pwelch'
print(x, ...)
## S3 method for class 'pwelch'
plot(
  x,
  log = c("xy", "x", "y", ""),
  se = FALSE,
  xticks,
  type = "l",
  add = FALSE,
  col = graphics::par("fg"),
  col.se = "orange",
  alpha.se = 0.5,
  lty = 1,
  lwd = 1,
  cex = 1,
  las = 1,
  main = "Welch periodogram",
  xlab,
  ylab,
  xlim = NULL,
  ylim = NULL,
  xaxs = "i",
  yaxs = "i",
  xline = 1.2 * cex,
  yline = 2 * cex,
  mar = c(2.6, 3.8, 2.1, 0.6) * (0.5 + cex/2),
  mgp = cex * c(2, 0.5, 0),
  tck = -0.02 * cex,
  grid = TRUE,
  ...
)

mv_pwelch(x, margin, fs, nfft)

### Arguments

- **x** 'pwelch' instance returned by `pwelch` function
- **fs** sample rate, average number of time points per second
- **window** window length in time points, default size is 64
- **noverlap** overlap between two adjacent windows, measured in time points; default is 8
- **nfft** number of basis functions to apply
- **col, xlim, ylim, main, type, cex, las, xlab, ylab, lty, lwd, xaxs, yaxs, mar, mgp, tck** parameters passed to `plot.default`
- **plot** integer, whether to plot the result or not; choices are 0, no plot; 1 plot on a new canvas; 2 add to existing canvas
- **log** indicates which axis should be log10-transformed, used by the `plot` function. For 'x' axis, it's log10-transform; for 'y' axis, it's 10log10-transform (decibel unit). Choices are "xy", "x", "y", and "."
... will be passed to `plot.pwelch` or ignored

`se` logical or a positive number indicating whether to plot standard error of mean; default is false. If provided with a number, then a multiple of standard error will be drawn. This option is only available when power is in log-scale (decibel unit)

`xticks` ticks to show on frequency axis

`add` logical, whether the plot should be added to existing canvas

`col.se, alpha.se` controls the color and opacity of the standard error

`xline, yline` controls how close the axis labels to the corresponding axes

`grid` whether to draw rectangular grid lines to the plot; only respected when `add=FALSE`; default is true

`margin` the margin in which `pwelch` should be applied to

**Value**

A list with class `'ravetools-pwelch'` that contains the following items:

- `freq` frequencies used to calculate the 'periodogram'
- `spec` resulting spectral power for each frequency
- `window` window function (in numerical vector) used
- `noverlap` number of overlapping time-points between two adjacent windows
- `nfft` number of basis functions
- `fs` sample rate
- `x_len` input signal length
- `method` a character string 'Welch'

**Examples**

```r
x <- rnorm(1000)
pwel <- pwelch(x, 100)
pwel

plot(pwel, log = "xy")
```
Description

Convert raw vectors to R vectors

Usage

raw_to_uint8(x)
raw_to_uint16(x)
raw_to_uint32(x)
raw_to_int8(x)
raw_to_int16(x)
raw_to_int32(x)
raw_to_int64(x)
raw_to_float(x)
raw_to_string(x)

Arguments

x raw vector of bytes

Details

For numeric conversions, the function names are straightforward. For example, raw_to_uintN converts raw vectors to unsigned integers, and raw_to_intN converts raw vectors to signed integers. The number 'N' stands for the number of bits used to store the integer. For example raw_to_uint8 uses 8 bits (1 byte) to store an integer, hence the value range is 0-255.

The input data length must be multiple of the element size represented by the underlying data. For example uint16 integer uses 16 bits, and one raw number uses 8 bits, hence two raw vectors can form one unsigned integer-16. That is, raw_to_uint16 requires the length of input to be multiple of two. An easy calculation is: the length of x times 8, must be divided by 'N' (see last paragraph for definition).

The returned data uses the closest available R native data type that can fully represent the data. For example, R does not have single float type, hence raw_to_float returns double type, which can represent all possible values in float. For raw_to_uint32, the potential value range is 0 - (2^32-1). This exceeds the limit of R integer type (-2^31) - (2^31-1). Therefore, the returned values will be real (double float) data type.
There is no native data type that can store integer-64 data in R. Package bit64 provides integer64 type, which will be used by raw_to_int64. Currently there is no solution to convert raw to unsigned integer-64 type.

raw_to_string converts raw to character string. This function respects null character, hence is slightly different than the native rawToChar, which translates raw byte-by-byte. If each raw byte represents a valid character, then the above two functions returns the same result. However, when the characters represented by raw bytes are invalid, raw_to_string will stop parsing and returns only the valid characters, while rawToChar will still try to parse, and most likely to result in errors. Please see Examples for comparisons.

Value

Numeric vectors, except for raw_to_string, which returns a string.

Examples

```r
# 0x00, 0x7f, 0x80, 0xFF
x <- as.raw(c(0, 127, 128, 255))
raw_to_uint8(x)

# The first bit becomes the integer sign
# 128 -> -128, 255 -> -1
raw_to_int8(x)

## Comments based on little endian system
# 0x7f00 (32512), 0xFF80 (65408 unsigned, or -128 signed)
raw_to_uint16(x)
raw_to_int16(x)

# 0xFF807F00 (4286611200 unsigned, -8356096 signed)
raw_to_uint32(x)
raw_to_int32(x)

# ---------------------------- String ---------------------------
# ASCII case: all valid
x <- charToRaw("This is an ASCII string")
raw_to_string(x)
rawToChar(x)

x <- c(charToRaw("This is the end."),
       as.raw(0),
       charToRaw("*** is invalid"))

# rawToChar will raise error
raw_to_string(x)

# ---------------------------- Integer64 ------------------------
```
# Runs on little endian system
x <- as.raw(c(0x80, 0x00, 0x7f, 0x80, 0xFF, 0x50, 0x7f, 0x00))

# Calculate bitstring, which concatenates the followings
# 10000000 (0x80), 00000000 (0x00), 01111111 (0x7f), 10000000 (0x80),
# 11111111 (0xFF), 01010000 (0x50), 01111111 (0x7f), 00000000 (0x00)

if(.Platform$endian == "little") {
  bitstring <- paste0(
    "000000001111111010100001111111",  
    "1000000011111110000000010000000"  
  )
} else {
  bitstring <- paste0(
    "00000001000000001111111000000001",  
    "11111111000010101111111000000000"  
  )
}

# This is expected value
bit64::as.integer64(structure(
  bitstring,
  class = "bitstring"
))

# This is actual value
raw_to_int64(x)

---

**register_volume**  
*Imaging registration using ‘NiftyReg’*

**Description**

Registers ‘CT’ to ‘MRI’, or ‘MRI’ to another ‘MRI’

**Usage**

```r
register_volume(
  source,
  target,
  method = c("rigid", "affine", "nonlinear"),
  interpolation = c("cubic", "trilinear", "nearest"),
  threads = detect_threads(),
  symmetric = TRUE,
  verbose = TRUE,
  ...
)
```

Arguments

source  source imaging data, or a 'nifti' file path; for example, 'CT'
target  target imaging data to align to; for example, 'MRI'
method  method of transformation, choices are 'rigid', 'affine', or 'nonlinear'
interpolation  how volumes should be interpolated, choices are 'cubic', 'trilinear', or 'nearest'
threads, symmetric, verbose, ...

see niftyreg

Value

See niftyreg

Examples

if(interactive()) {

source <- system.file("extdata", "epi_t2.nii.gz", package="RNiftyReg")
target <- system.file("extdata", "flash_t1.nii.gz", package="RNiftyReg")
aligned <- register_volume(source, target, verbose = FALSE)

source_img <- aligned$source[[1]]
target_img <- aligned$target
aligned_img <- aligned$image

par(mfrow = c(2, 2), mar = c(0.1, 0.1, 3.1, 0.1))

pal <- grDevices::grey.colors(256, alpha = 1)
image(source_img[,30], asp = 1, axes = FALSE,
col = pal, main = "Source image")
image(target_img[,64], asp = 1, axes = FALSE,
col = pal, main = "Target image")
image(aligned_img[,64], asp = 1, axes = FALSE,
col = pal, main = "Aligned image")

# bucket fill and calculate differences
aligned_img[is.nan(aligned_img) | aligned_img <= 1] <- 1
target_img[is.nan(target_img) | aligned_img <= 1] <- 1
diff <- abs(aligned_img / target_img - 1)
image(diff[,64], asp = 1, axes = FALSE,
col = pal, main = "Percentage Difference")
}


shift_array  

Shift array by index

Description

Re-arrange arrays in parallel

Usage

shift_array(x, along_margin, unit_margin, shift_amount)

Arguments

- **x**: array, must have at least matrix
- **along_margin**: which index is to be shifted
- **unit_margin**: which dimension decides shift_amount
- **shift_amount**: shift amount along along_margin

Details

A simple use-case for this function is to think of a matrix where each row is a signal and columns stand for time. The objective is to align (time-lock) each signal according to certain events. For each signal, we want to shift the time points by certain amount.

In this case, the shift amount is defined by shift_amount, whose length equals to number of signals. along_margin=2 as we want to shift time points (column, the second dimension) for each signal. unit_margin=1 because the shift amount is depend on the signal number.

Value

An array with same dimensions as the input x, but with index shifted. The missing elements will be filled with NA.

Examples

```r
# Set ncores = 2 to comply to CRAN policy. Please don't run this line
ravetools_threads(n_threads = 2L)

x <- matrix(1:10, nrow = 2, byrow = TRUE)
z <- shift_array(x, 2, 1, c(1,2))

y <- NA * x
y[,1:4] = x[,2:5]
y[,2:3] = x[,3:5]
```
# Check if z and y are the same
z - y

# array case
# x is Trial x Frequency x Time
x <- array(1:27, c(3,3,3))

# Shift time for each trial, amount is 1, -1, 0
shift_amount <- c(1,-1,0)
z <- shift_array(x, 3, 1, shift_amount)

if(interactive(){
  par(mfrow = c(3, 2), mai = c(0.8, 0.6, 0.4, 0.1))
  for( ii in 1:3 ){
    image(t(x[ii, ,]), ylab = 'Frequency', xlab = 'Time',
         main = paste('Trial', ii))
    image(t(z[ii, ,]), ylab = 'Frequency', xlab = 'Time',
         main = paste('Shifted amount:', shift_amount[ii]))
  }
}

---

wavelet

'Morlet' wavelet transform (Discrete)

Description

Transform analog voltage signals with 'Morlet' wavelets: complex wavelet kernels with $\pi/2$ phase differences.

Usage

wavelet_kernels(freqs, srate, wave_num)

morlet_wavelet(
  data,
  freqs,
  srate,
  wave_num,
  precision = c("float", "double"),
  trend = c("constant", "linear", "none"),
  signature = NULL,
  ...
)

wavelet_cycles_suggest(
  freqs,
frequency_range = c(2, 200),
cycle_range = c(3, 20)
)

Arguments

freqs frequency in which data will be projected on
srate sample rate, number of time points per second
wave_num desired number of cycles in wavelet kernels to balance the precision in time and
amplitude (control the smoothness); positive integers are strongly suggested
data numerical vector such as analog voltage signals
precision the precision of computation; choices are 'float' (default) and 'double'.
trend choices are 'constant': center the signal at zero; 'linear': remove the linear
trend; 'none' do nothing
signature signature to calculate kernel path to save, internally used
... further passed to detrend;
frequency_range frequency range to calculate, default is 2 to 200
cycle_range number of cycles corresponding to frequency_range. For default frequency
range (2 - 200), the default cycle_range is 3 to 20. That is, 3 wavelet kernel
cycles at 2 Hertz, and 20 cycles at 200 Hertz.

Value

wavelet_kernels returns wavelet kernels to be used for wavelet function; morlet_wavelet returns
a file-based array if precision is 'float', or a list of real and imaginary arrays if precision is
'double'

Examples

if(interactive()){

# generate sine waves
time <- seq(0, 3, by = 0.01)
x <- sin(time * 20*pi) + exp(-time^2) * cos(time * 10*pi)

plot(time, x, type = 'l')

# freq from 1 - 15 Hz; wavelet using float precision
freq <- seq(1, 15, 0.2)
coef <- morlet_wavelet(x, freq, 100, c(2,3))

# to get coefficients in complex number from 1-10 time points
coeff1:10, ]

# power
power <- Mod(coef[1])^2
# Power peaks at 5Hz and 10Hz at early stages
# After 1.0 second, 5Hz component fade away
image(power, x = time, y = freq, ylab = "frequency")

# wavelet using double precision
coeff2 <- morlet_wavelet(x, freq, 100, c(2,3), precision = "double")
power2 <- (coeff2$real[])^2 + (coeff2$imag[])^2

image(power2, x = time, y = freq, ylab = "frequency")

# The maximum relative change of power with different precisions
max(abs(power/power2 - 1))

# display kernels
freq <- seq(1, 15, 1)
kern <- wavelet_kernels(freq, 100, c(2,3))
print(kern)

plot(kern)
}
Index

band_pass, 2
band_pass1 (band_pass), 2
band_pass2 (band_pass), 2
baseline_array, 4
collapse, 7
convolve, 9
convolve_image (convolve), 9
convolve_signal (convolve), 9
convolve_volume (convolve), 9
cov, 14
decimate, 10
detect_threads (parallel-options), 27
detrend, 11, 38
diagnose_channel, 12
fast_cov, 14
fast_median (fast_quantile), 15
fast_mvmmedian (fast_quantile), 15
fast_mvquantile (fast_quantile), 15
fast_quantile, 15
fill_surface, 16
filter_signal, 18
filtfilt, 19
grow_volume, 20
hist, 12, 13
interpolate_stimulation, 21
matlab_palette, 22
mesh_from_volume, 22
morlet_wavelet (wavelet), 37
multitaper, 24
multitaper_config (multitaper), 24
mv_pwelch (pwelch), 29
niftyreg, 35
notch_filter, 26
par, 13, 29
parallel-options, 27
plot, 29
plot.default, 30
plot.pwelch (pwelch), 29
plot_signals, 27
print.pwelch (pwelch), 29
pwelch, 12, 13, 29
ravetools_threads (parallel-options), 27
raw-to-sexp, 32
raw_to_float (raw-to-sexp), 32
raw_to_int16 (raw-to-sexp), 32
raw_to_int32 (raw-to-sexp), 32
raw_to_int64 (raw-to-sexp), 32
raw_to_int8 (raw-to-sexp), 32
raw_to_string (raw-to-sexp), 32
raw_to_uint16 (raw-to-sexp), 32
raw_to_uint32 (raw-to-sexp), 32
raw_to_uint8 (raw-to-sexp), 32
rawToChar, 33
register_volume, 34
shift_array, 36
vcgIsosurface, 22
vcgSmooth, 22, 23
vcgUniformRemesh, 22, 23
wavelet, 37
wavelet_cycles_suggest (wavelet), 37
wavelet_kernels (wavelet), 37