Package ‘ravetools’

February 16, 2022

Type Package

Title Signal Processing Toolbox for Analyzing 'Electrophysiology' Data

Version 0.0.3

Language en-US

Description Implemented fast and memory-efficient 'Notch'-filter, 'Welch-periodogram', and discrete wavelet transform algorithm for hours of high-resolution signals; providing fundamental toolbox for 'iEEG' preprocess pipelines.


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

URL http://dipterix.org/ravetools/

License GPL-3

Encoding UTF-8

RoxygenNote 7.1.2

Depends R (>= 4.0.0)

SystemRequirements fftw3 (libfftw3-dev (deb), or fftw-devel (rpm))

Imports graphics, stats, filearray (>= 0.1.3), Rcpp (>= 1.0.8), waveslim (>= 1.8.2), signal (>= 0.7.7), digest (>= 0.6.29)

LinkingTo Rcpp

Suggests fftwtools, pracma, microbenchmark, testthat (>= 3.0.0)

Config/testthat/edition 3

NeedsCompilation yes

Author Zhengjia Wang [aut, cre, cph], Beauchamp lab [cph], Karim Rahim [cph] (R package fftwtools), Prerau Lab [cph] (Multitaper Spectrogram Code),
baseline_array

Calculate Contrasts of Arrays in Different Methods

Description

Provides five methods to baseline an array and calculate contrast.

Usage

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

## 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_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.
- **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 (automatically 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:

\[ \text{session} x \text{frequency} x \text{time} x \text{location} \]

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

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

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*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"**

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

- **"sqrt_percentage"**

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

"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()){
collapse

Description

Collapse array

Usage

collapse(x, keep, ...)

## S3 method for class 'array'
collapse(
  x,
  keep,
  average = TRUE,
  baseline_indexpoints = baseline_window,
  unit_dims = c(1, 2, 4),
  method = 'percentage'
)


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)
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))
```
decimate

```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
    })
)
```

decimate

Decimate with 'FIR' or 'IIR' filter

Description

Decimate with 'FIR' or 'IIR' filter

Usage

```r
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'

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

```r
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' (\texttt{pwelch}) and histogram (\texttt{hist})

Usage

\begin{verbatim}
diagnose_channel(
    s1,
    s2 = NULL,
    sc = NULL,
    srate,
    name = "",
    try_compress = TRUE,
    max_freq = 300,
    window = ceiling(srate * 2),
    noverlap = window/2,
    std = 3,
    cex = 1.5,
    lwd = 0.5,
    plim = NULL,
    nclass = 100,
    main = "Channel Inspection",
    col = c("black", "red"),
    which = NULL,
    start_time = 0,
    boundary = NULL,
    mar = c(5.2, 5.1, 4.1, 2.1),
    mai = c(0.6, 0.54, 0.4, 0.1),
    ...
)
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{s1} \hspace{1cm} the main signal to draw
  \item \texttt{s2} \hspace{1cm} the comparing signal to draw; usually \texttt{s1} after some filters; must be in the same sampling rate with \texttt{s1}; can be \texttt{NULL}
  \item \texttt{sc} \hspace{1cm} decimated \texttt{s1} to show if \texttt{srate} is too high; will be automatically generated if \texttt{NULL}
  \item \texttt{srate} \hspace{1cm} sampling rate
  \item \texttt{name} \hspace{1cm} name of \texttt{s1}, or a vector of two names of \texttt{s1} and \texttt{s2} if \texttt{s2} is provided
  \item \texttt{try_compress} \hspace{1cm} whether try to compress (decimate) \texttt{s1} if \texttt{srate} is too high for performance concerns
\end{itemize}
max_freq the maximum frequency to display in 'Welch Periodograms'

window, noverlap

see \texttt{pwelch}

std the standard deviation of the channel signals used to determine boundary; default is plus-minus 3 standard deviation

cex, lwd, mar, mai, ...

graphical parameters; see \texttt{par}

plim the y-axis limit to draw in 'Welch Periodograms'

nclass number of classes to show in histogram (\texttt{hist})

main the title of the signal plot

col colors of s1 and s2

which NULL or integer from 1 to 4; if NULL, all plots will be displayed; otherwise only the subplot will be displayed

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

\textbf{Value}

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

\textbf{Examples}

\begin{verbatim}
library(ravetools)

# Generate 20 second data at 2000 Hz
time <- seq(0, 20, by = 1 / 2000)
signal <- sin(120 * pi * time) + sin(time * 20*pi) +
    exp(-time^2) * cos(time * 10*pi) +
    rnorm(length(time))

signal2 <- notch_filter(signal, 2000)

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

---

\textbf{fast_c cov}

\textit{Calculate massive covariance matrix in parallel}

\textbf{Description}

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

```r
cov(x, y = NULL, col_x = NULL, col_y = NULL, df = NA)
```

Arguments

- **x**: a numeric vector, matrix or data frame; a matrix is highly recommended to maximize the performance.
- **y**: NULL (default) or a vector, matrix or data frame with compatible dimensions to `x`; the default is equivalent to `y = x`.
- **col_x**: integers indicating the subset indices (columns) of `x` to calculate the covariance, or NULL to include all the columns; default is NULL.
- **col_y**: integers indicating the subset indices (columns) of `y` to calculate the covariance, or NULL to include all the columns; default is NULL.
- **df**: a scalar indicating the degrees of freedom; default is `nrow(x)-1`.

Value

A covariance matrix of `x` and `y`. Note that there is no NA handling. Any missing values will lead to NA in the resulting covariance matrices.

Examples

```r
# 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)
# Call cov(x) to compare
fast_cov(x)

# Calculate covariance of subsets
fast_cov(x, col_x = 1, col_y = 1:2)

if(interactive(){
# Speed comparison, better to use multiple cores (4, 8, or more)
# to show the differences.

ravetools_threads(n_threads = -1)
x <- matrix(rnorm(100000), nrow = 1000)
microbenchmark::microbenchmark(
  fast_cov = {
    fast_cov(x, col_x = 1:50, col_y = 51:100)
  },
  cov = {
    cov(x[,1:50], x[,51:100])
  },
  unit = 'ms', times = 10
)
```
multitaper

Description

'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,
multitaper <-

  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 com-
  puted 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'
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 1E-7 with standard deviation at most 1E-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/).

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

  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

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

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

if(interactive()){
  detect_threads()
  ravetools_threads(n_threads = 2)
}
**pwelch**  
*Calculate 'Welch Periodogram'*

**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**

```r
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", ""),
  type = "l",
  add = FALSE,
  col = 1,
  cex = 1,
  cex.main = cex,
  cex.sub = cex,
  cex.lab = cex * 0.8,
  cex.axis = cex * 0.7,
  las = 1,
  main = "Welch periodogram",
  xlab,
  ylab,
  xlim = NULL,
  ylim = NULL,
  ...
)
```
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, cex.main, cex.sub, cex.lab, cex.axis, las, xlab, ylab` 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 \( \log_{10} \)-transformed, used by the plot function. For 'x' axis, it's \( \log_{10} \)-transform; for 'y' axis, it's \( 10 \log_{10} \)-transform (decibel unit). Choices are "xy", "x", "y", and "."
- `...` will be passed to `plot.pwelch` or ignored
- `add` logical, whether the plot should be added to existing canvas
- `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

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[1,1:4] = x[1,2:5]
y[2,1:3] = x[2,3:5]
```
```r
# 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**

```r
wavelet_kernels(freqs, srate, wave_num)
morlet_wavelet(
  data,
  freqs,
  srate,
  wave_num,
  precision = c("float", "double"),
  trend = c("constant", "linear", "none"),
  ...
)
```
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
...  further passed to detrend;

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

# 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
coef[1:10, ,]

# power
power <- Mod(coef[])^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")

# 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

baseline_array, 2

collapse, 5
cov, 10

decimate, 7
detect_threads (parallel-options), 15
detrend, 8, 20
diagnose_channel, 9

fast_cov, 10

hist, 9, 10

matlab_palette, 12
morlet_wavelet (wavelet), 19
multitaper, 12
multitaper_config (multitaper), 12
mv_pwelch (pwelch), 16

notch_filter, 14

par, 10
parallel-options, 15
plot.default, 17
plot.pwelch (pwelch), 16
print.pwelch (pwelch), 16
pwelch, 9, 10, 16

ravetools_threads (parallel-options), 15

shift_array, 18

wavelet, 19
wavelet_kernels (wavelet), 19