Package 'ravetools'

May 31, 2026

Type Package

Title Signal and Image Processing Toolbox for Analyzing Intracranial Electroencephalography Data

Version 0.2.6

Language en-US

Description Implemented fast and memory-efficient Notch-filter, Welch-periodogram, discrete wavelet spectrogram for minutes of high-resolution signals, fast 3D convolution, image registration, 3D mesh manipulation; providing fundamental toolbox for intracranial Electroencephalography (iEEG) pipelines. Documentation and examples about 'RAVE' project are provided at <https://rave.wiki>, 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://rave.wiki, https://dipterix.org/ravetools/, https://github.com/dipterix/ravetools

License GPL (>= 2)

Encoding UTF-8

Depends R (>= 4.0.0)

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

Copyright Karim Rahim (author of R package 'fftwtools', licensed under 'GPL-2' or later) is the original author of 'src/ffts.h' and 'src/ffts.cpp'. Prerau's Lab wrote the original 'R/multitaper.R', licensed under 'MIT'. Marcus Geelnard wrote the source code of 'TinyThread' library ('MIT' license) located at 'inst/include/tthread'. Stefan Schlager wrote the original code that converts R objects to 'vcg' (see 'src/vcgCommon.h', licensed under 'GPL-2' or later). Visual Computing Lab is the copyright holder of 'vcglib' source code (see 'src/vcglib', licensed under GPL-2 or later).

Imports graphics, stats, filearray (>= 0.1.3), Rcpp, waveslim (>= 1.8.2), pracma, digest (>= 0.6.29), splines, RNiftyReg (>= 2.7.1), R6 (>= 2.5.1), gsignal (>= 0.3.5)

LinkingTo Rcpp, RcppEigen

Suggests fftwtools, bit64, grDevices, microbenchmark, freesurferformats, testthat, vctrs

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Zhengjia Wang [aut, cre],
John Magnotti [aut],
Michael Beauchamp [aut],
Trustees of the University of Pennsylvania [cph] (All files in this package unless explicitly stated in the file or listed in the 'Copyright' section below.),
Karim Rahim [cph, ctb] (Contributed to src/ffts.h and stc/ffts.cpp),
Thomas Possidente [cph, ctb] (Contributed to R/multitaper.R),
Michael Prerau [cph, ctb] (Contributed to R/multitaper.R),
Marcus Geelnard [ctb, cph] (TinyThread library, tinythreadpp.bitsnbites.eu, located at inst/include/tthread/),
Stefan Schlager [ctb, cph] (R-vcg interface, located at src/vcgCommon.h),
Visual Computing Lab, ISTI [ctb, cph] (Copyright holder of vcglib, located at src/vcglib/)

Maintainer Zhengjia Wang <dipterix.wang@gmail.com>

Repository CRAN

Date/Publication 2026-05-31 05:10:27 UTC

Contents

---

band_pass *Band-pass signals*

Description

Band-pass signals

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
...	ignored
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 o 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)

oldpar <- par(mfrow = c(2, 2), mar = c(3.1, 2.1, 3.1, 0.1))
# ---- Using band_pass1 -------------------------------------------------

y1 <- band_pass1(x, 2000, 0.1, 1)
y2 <- band_pass1(x, 2000, 1, 5)
y3 <- band_pass1(x, 2000, 10, 80)

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,
  cex = 0.6
)

# plot pwelch
pwelch(x, fs = 2000, window = 4000, noverlap = 2000, plot = 1)
pwelch(y1, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "red")
pwelch(y2, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "blue")
pwelch(y3, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "green")


# ---- Using band_pass2 with FIR filters --------------------------------

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)

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,
  cex = 0.6
)

# plot pwelch
pwelch(x, fs = 2000, window = 4000, noverlap = 2000, plot = 1)
pwelch(z1, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "red")
pwelch(z2, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "blue")
pwelch(z3, fs = 2000, window = 4000, noverlap = 2000,
       plot = 2, col = "green")

# ---- Clean this demo ----------------------------------------------------
par(oldpar)
```

---

baseline_array *Calculate Contrasts of Arrays in Different Methods*

Description

Provides seven 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",
    "db_zscore", "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; one of "percentage", "sqrt_percentage", "decibel", "zscore", "sqrt_zscore", "db_zscore", or "subtract_mean". See Details.
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:

$$session \times frequency \times time \times location$$

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 `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 $frequencyxtime$, 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 seven baseline methods. They fit for different types of data. Denote z is a unit signal and z_0 is its baseline slice. Then these baseline methods are:

"percentage"

$$\frac{z - \bar{z_0}}{\bar{z_0}} \times 100\%$$

"sqrt_percentage"

$$\frac{\sqrt{z} - \sqrt{\bar{z_0}}}{\sqrt{\bar{z_0}}} \times 100\%$$

"decibel"

$$10 \times (\log_{10}(z) - \bar{\log_{10}}(z_0))$$

"zscore"

$$\frac{z - \bar{z_0}}{sd(z_0)}$$

`"sqrt_zscore"`

$$\frac{\sqrt{z} - \bar{\sqrt{z_0}}}{sd(\sqrt{z_0})}$$

`"db_zscore"` Z-score applied in the decibel (10log10) domain:

$$\frac{10\log_{10}(z) - \bar{10\log_{10}}(z_0)}{sd(10\log_{10}(z_0))}$$

`"subtract_mean"` Simple mean subtraction with no scaling:

$$z - \bar{z_0}$$

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, might take a while to profile

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)
```

`butter_max_order` *'Butterworth' filter with maximum order*

Description

Large filter order might not be optimal, but at lease this function provides a feasible upper bound for the order such that the filter has a stable `AR` component.

Usage

```
butter_max_order(
  w,
  type = c("low", "high", "pass", "stop"),
  r = 10 * log10(2),
  tol = .Machine$double.eps
)
```

Arguments

`w`	scaled frequency ranging from 0 to 1, where 1 is 'Nyquist' frequency
`type`	filter type
`r`	decibel attenuation at frequency `w`, default is around `3` `dB` (half power)
`tol`	tolerance of reciprocal condition number, default is `.Machine$double.eps`.

Value

'Butterworth' filter in 'Arma' form.

Examples

```
# Find highest order (sharpest transition) of a band-pass filter
sample_rate <- 500
nyquist <- sample_rate / 2

type <- "pass"
w <- c(1, 50) / nyquist
Rs <- 6     # power attenuation at w

# max order filter
filter <- butter_max_order(w, "pass", Rs)

# -6 dB cutoff should be around 1 ~ 50 Hz
diagnose_filter(filter$b, filter$a, fs = sample_rate)
```

---

carla *Common Average Re-referencing by Least Anti-Correlation (CARLA)*

Description

Selects an optimal subset of channels to use as the common average reference (CAR) for cortico-cortical evoked potential (`CCEP`) data, following the `CARLA` (see 'Reference' and 'Citation'). Channels are ranked in increasing order of their cross-trial covariance (or variance when only one trial is available); subsets are then iteratively grown and the size that yields the least anti-correlation between the candidate reference and the remaining unreferenced channels is selected as optimal.

Usage

```
carla(
  x,
  nboot = 100L,
  sensitive = FALSE,
  min_size = NULL,
  absolute_rank = FALSE,
  virtual_reference = FALSE
)
```

Arguments

`x`	numeric array of shape `channels` x `time` x `trials`; if a matrix is provided, it is treated as `channels` x `time` (single trial). The signal must already be cropped to the responsive time window (typically `0.01` s to `0.3` s post-stimulus).

`nboot` integer, number of bootstrapped trial resamplings used to estimate the optimization statistic; defaults to `100`. Ignored when a single trial is supplied.

`sensitive` logical; if `TRUE` (and more than one trial is supplied), the more sensitive cutoff is used, corresponding to the channel count just before the first statistically significant decrease in the mean anti-correlation curve. The default `FALSE` returns the global maximum of that curve.

`min_size` integer, minimum subset size considered when `sensitive = TRUE`; defaults to `max(2, ceiling(0.1 * nchan_good))`, where `nchan_good` is the number of usable channels after the bad-channel mask has been applied (see Details). Floored at 2 because subset size 1 is never evaluated.

`absolute_rank` logical; if `FALSE` (the default), the per-channel ranking statistic is the signed mean of the off-diagonal trial-to-trial covariances, as in the original CARLA manuscript. This assumes a phase-locked evoked response, so trial-pair covariances on responsive channels are positive and a clean signed mean separates them from non-responsive channels. Set to `TRUE` to use the mean of the **absolute** covariances instead, which is more robust to evoked responses whose polarity flips across trials (e.g. alternating-polarity stimulation, biphasic `CCEPs` with jitter), at the cost of an upward bias on the non-responsive floor. Ignored when only one trial is supplied (the per-channel variance is used in that case).

`virtual_reference` logical; if `TRUE`, runs the modified CARLA before the iterative subset evaluation: rank channels once, designate the channel with the median rank as a "virtual reference", subtract its signal from every channel, then re-rank on the subtracted data. The virtual channel itself is pinned to the very end of the new ordering so it is never picked into the CAR until the final subset size. This is intended for data where the recording reference is contaminated by stimulation artifact or evoked activity: a contaminated reference makes every channel look artificially similar and biases the covariance/variance ranking statistic, but subtracting a mid-rank proxy of that contamination unbiases the ranking so genuinely responsive channels rise to the top. Default `FALSE` reproduces the original CARLA algorithm bit-for-bit.

Details

The function is a faithful port of the core `CARLA.m` routine; it does not perform notch filtering, time-window cropping, or grouping by stimulation site. Those steps belong to the surrounding preprocess pipeline (see the example below).

For each candidate subset size $n = 2, \ldots, N$, the candidate reference is computed as the channel-wise mean of the n lowest-ranked channels. Each of those n channels is then correlated, in its unreferenced form, against every channel of the candidate re-referenced subset. The resulting Pearson correlations are Fisher z-transformed; the row corresponding to the most globally anti-correlated channel is recorded as `zmin`. The optimal n is the one that maximizes (i.e. makes least negative) the mean of `zmin`.

Bad-channel mask. Channels whose ranking statistic is exactly zero or `NA` are flagged as "bad" and excluded both from the CAR candidate pool and from the target-channel correlation rows. This catches flat / dead channels (constant signal, no usable variance) and , when `virtual_reference = TRUE`, automatically removes the virtual channel itself, since subtracting it from itself yields a zero

trace. All references to channel indices in the returned `order`, `n_optimum`, and `zmin_mean` are with respect to the **good** channels only; `vars` is full-length so callers can inspect the raw scores.

Value

A list with the following elements:

`channels` integer vector, sorted indices (1-based) of the channels chosen to construct the common average reference.

`car` numeric matrix of shape `time` x `trials` (or numeric vector when only one trial is supplied) holding the common average signal computed from `channels`. Subtract this from each channel of the original signal to obtain the re-referenced data.

`order` integer vector, indices of the **good** channels sorted in increasing order of the ranking statistic. Bad channels (zero variance / all-`NA`; see Details) are excluded.

`vars` numeric vector of length `nchan` containing the per-channel ranking statistic (mean cross-trial covariance, or variance for a single trial). Bad channels keep their raw value (zero or `NA`) so callers can audit the mask.

`n_optimum` integer, the optimal subset size selected (indexes into `order`).

`zmin_mean` numeric matrix of shape `length(order)` x `nboot` (or a length-`length(order)` vector for a single trial / `nboot = 1`) holding, for each subset size and bootstrap, the mean Fisher z-transformed correlation of the most globally anti-correlated unreferenced channel against the candidate CAR. Row 1 is always `NA` (subset size 1 is not evaluated).

`bad_channels` integer vector of channel indices that were excluded from the analysis because their ranking statistic was zero or `NA` (flat / dead channels, plus the virtual channel itself when `virtual_reference = TRUE`).

`virtual_channel` integer, the index (1-based) of the channel used as the virtual reference when `virtual_reference = TRUE`; `NA_integer_` otherwise.

`vars1` numeric vector of the first-pass ranking statistic when `virtual_reference = TRUE` (the post-subtraction statistic is returned in `vars`); `NULL` otherwise.

References

The CARLA algorithm (`virtual_reference = FALSE`) is described in doi:10.1016/j.jneumeth.2024.110153; the modified CARLA precursor (`virtual_reference = TRUE`) is described in doi:10.1016/j.jneumeth.2025.110461, with a reference implementation at https://github.com/hharveygit/SPES_reference_contam. See `citation("ravetools")` for the full bibliographic entries of both manuscripts.

Examples

```
# ---- Simulate a small CCEP-like dataset --------------------------------
# 16 channels, 12 trials, sampled at 1 kHz, 0.5 s peri-stimulus epoch.
# Channels 1:4 are "responsive" (carry an evoked potential); the rest are
# noise-only and should make up the optimal CAR.
srate <- 1000
tt <- seq(-0.1, 0.4 - 1 / srate, by = 1 / srate)   # time, seconds
nchan <- 16
ntrial <- 30
resp_ch <- 1:4
```

```
noise_ch <- 4:7

# Evoked potential template: damped sinusoid starting at t = 0
ep <- ifelse(tt >= 0,
             80 * exp(-tt / 0.05) * sin(2 * pi * 12 * tt),
             0)

# channels x time x trials
x_full <- array(rnorm(nchan * length(tt) * ntrial, sd = 5),
                dim = c(nchan, length(tt), ntrial))
for (ch in resp_ch) {
  for (k in seq_len(ntrial)) {
    x_full[ch, , k] <- x_full[ch, , k] + ep * runif(1, -0.8, 1.2)
  }
}
for (ch in noise_ch) {
  for (k in seq_len(ntrial)) {
    tmp <- x_full[ch, , k]
    x_full[ch, , k] <- tmp + sign(ch %% 2 - 0.5) * 5 *
      runif(length(tmp), 0.8, 1.2)
  }
}
# Add artifacts common to all channels and trials
artifacts <- 6 * sin(2 * pi * 60 * tt) + 7 * sin(2 * pi * 24 * tt)
x_full <- sweep(x_full, 2L, artifacts, "+")

# ---- 1. Notch filter line noise (per channel, per trial) ---------------
# The CARLA paper notch-filters before ranking; the re-reference itself
# is applied to the original (unfiltered) signal.
x_clean <- x_full
for (ch in seq_len(nchan)) {
  for (k in seq_len(ntrial)) {
    x_clean[ch, , k] <- notch_filter(
      x_full[ch, , k], sample_rate = srate,
      lb = c(59, 119, 179), ub = c(61, 121, 181)
    )
  }
}

# ---- 2. Crop to the responsive window (0.01 s to 0.3 s post-stim) ------
resp_idx <- which(tt > 0.0 & tt <= 0.3)
x_resp <- x_clean[, resp_idx, , drop = FALSE]

# ---- 3. Run CARLA to pick reference channels ---------------------------
fit <- carla(x_resp, sensitive = TRUE, absolute_rank = TRUE,
             virtual_reference = TRUE)
fit$channels        # selected reference channels (should exclude 1:4)
fit$n_optimum       # number of channels in the optimal CAR

# ---- 4. Re-reference the ORIGINAL (unfiltered) signal ------------------
# mean or median, your choice!
car_full <- apply(x_full[fit$channels, , , drop = FALSE], c(2, 3), mean)
```

```
# old-style: using all channels for CAR
car_old <- apply(x_full, c(2, 3), mean)

x_reref <- sweep(x_full, c(2, 3), car_full, "-")
x_compare <- sweep(x_full, c(2, 3), car_old, "-")

# ---- 5. Inspect: evoked potential is preserved on responsive channels --
op <- graphics::par(mfrow = c(2, 4), mar = c(4, 4, 2, 1))
ravetools::plot_signals(
  signals = x_full[, , 1],
  sample_rate = srate,
  main = "Trial 1 - (raw)")

ravetools::plot_signals(
  signals = x_clean[, , 1],
  sample_rate = srate,
  main = "Notch-filtered")

ravetools::plot_signals(
  signals = x_reref[, , 1],
  sample_rate = srate,
  main = sprintf("CARLA-ref (n=%d)", length(fit$channels)))

ravetools::plot_signals(
  signals = x_compare[, , 1],
  sample_rate = srate,
  main = "Conventional CAR for comparison")

col <- adjustcolor(seq_len(nchan))
col[resp_ch] <- adjustcolor(col[resp_ch], alpha.f = 0.2)

graphics::matplot(tt, t(rowMeans(x_full, dims = 2L)),
                  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
                  main = "Trial-averaged (raw", col = col)

graphics::matplot(tt, t(rowMeans(x_clean, dims = 2L)),
                  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
                  main = "Notch-filtered", col = col)

graphics::matplot(tt, t(rowMeans(x_reref, dims = 2L)),
                  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
                  main = "CARLA-referenced", col = col)

graphics::matplot(tt, t(rowMeans(x_compare, dims = 2L)),
                  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
                  main = "conventional CAR-referenced", col = col)

graphics::par(op)
```

`catmull_rom_3d` Catmull-Rom *3D Spline Curve*

Description

Creates a smooth Catmull-Rom spline curve through a set of 3D key points.

Usage

```
catmull_rom_3d(
  points,
  curve_type = c("centripetal", "chordal", "uniform"),
  tension = 0.5,
  closed = FALSE
)
```

Arguments

points	numeric matrix with at least 2 rows and exactly 3 columns (x, y, z), giving the key (control) points through which the curve passes.
curve_type	character; One of "centripetal" (default), "chordal", or "uniform". "centripetal" uses $\alpha = 0.5$ (square-root of chord length), "chordal" uses $\alpha = 1$ (chord length), and "uniform" is the classic formulation controlled by tension.
tension	numeric scalar in $[0, 1]$; tangent scaling factor used only when curve_type = "uniform". At 0.5 (default) the curve matches the standard Catmull-Rom formulation.
closed	logical; if TRUE the curve closes on itself by connecting the last point back to the first. Default is FALSE.

Value

An object of class "ravetools_curve" (a list) with the following elements:

points The input key-point matrix ($n \times 3$).

curve_type Character, the parameterization type.

tension Numeric, the tension value (relevant for "uniform" only).

closed Logical, whether the curve is closed.

get_point A function(t) that accepts a scalar t in $[0, 1]$ and returns a named numeric vector on the curve.

get_points A function(n) that returns an $n \times 3$ matrix of n evenly spaced points along the curve, with column names "x", "y", "z".

get_closest_t A function(query, coarse_n = 200L) that, given a 3-element numeric vector query (x, y, z), returns a list with elements t (the parameter value in $[0, 1]$ of the nearest point), point (the closest point on the curve as a named numeric vector), and distance (Euclidean distance from query to the curve). The search uses coarse_n uniform samples for an initial bracket followed by scalar optimization.

t_keypoints Numeric vector of length n with the t parameter value where each key point lies on the curve. First element is always 0, last is always 1.

segment_lengths Numeric vector of length $n - 1$ (open curve) or n (closed curve) containing the arc length of each spline segment, estimated by numerical integration.

See Also

print.ravetools_curve, plot.ravetools_curve

Examples

```
pts <- matrix(c(
  -33.0534, -10.6213, -21.8328,
  -34.7526, -25.5089, -14.5390,
  -41.2002, -10.4606, -22.0032,
  -46.4717, -10.3567, -22.1134,
  -51.7431, -10.2528, -22.2237,
  -57.0146, -10.1488, -22.3339,
  -62.2860, -10.0449, -22.4442,
  -67.5575,  -9.9410, -22.5544
), ncol = 3, byrow = TRUE)

curve <- catmull_rom_3d(pts)
print(curve)

# Sample 100 evenly spaced points along the curve
smooth <- curve$get_points(100)
head(smooth)

# Evaluate the curve at t = 0.5 (midpoint)
curve$get_point(0.5)

# get closest point on curve
curve$get_closest_t(c(-49, -10, -22))

plot(curve, use_rgl = FALSE)
```

---

check_filter *Check 'Arma' filter*

Description

Check 'Arma' filter

Usage

```
check_filter(b, a, w = NULL, r_expected = NULL, fs = NULL)
```

Arguments

b	moving average (MA) polynomial coefficients.
a	auto-regressive (AR) polynomial coefficients.
w	normalized frequency, ranging from 0 to 1, where 1 is 'Nyquist'

r_expected	attenuation in decibel of each w
fs	sample rate, used to infer the frequencies and formatting print message, not used in calculation; leave it blank by default

Value

A list of power estimation and the reciprocal condition number of the AR coefficients.

Examples

```
# create a butterworth filter with -3dB (half-power) at [1, 5] Hz
# and -60dB stop-band attenuation at [0.5, 6] Hz

sample_rate <- 20
nyquist <- sample_rate / 2

specs <- buttord(
  Wp = c(1, 5) / nyquist,
  Ws = c(0.5, 6) / nyquist,
  Rp = 3,
  Rs = 60
)
filter <- butter(specs)

# filter quality is poor because the AR-coefficients
# creates singular matrix with unstable inverse,
# this will cause `filtfilt` to fail
check_filter(
  b = filter$b, a = filter$a,

  # frequencies (normalized) where power is evaluated
  w = c(1, 5, 0.5, 6) / nyquist,

  # expected power
  r_expected = c(3, 3, 60, 60)

)
```

---

collapse *Collapse array*

Description

Collapse array

Usage

```
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
...	passed to other methods
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

```
# 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)




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))
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. This function is mainly designed for image convolution. For forward and backward convolution/filter, see filtfilt.

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

```
# ---- 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)

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

crp *Canonical Response Parameterization (*CRP*)*

Description

Parameterizes single-trial evoked responses (e.g. cortico-cortical evoked potentials, CCEPs) using the Canonical Response Parameterization method (see 'Citation'). The function estimates the response duration τ_R, the time after stimulus at which the evoked response has its most consistent, shared structure across trials. The estimator is obtained from the time course of cross-trial projection magnitudes, extracts the canonical response shape $C(t)$ via a linear kernel-trick PCA on the trial matrix truncated at τ_R, and reports per-trial weights, residuals, signal-to-noise, explained variance and extraction-significance statistics.

This is an R translation of CRP_method.m (and the surrounding artifact-rejection / duration-uncertainty logic in CRP_illustration.m) from the upstream MATLAB reference implementation; see 'References'.

Usage

```
crp(
  x,
  time,
  t_start = 0.015,
  t_end = 1,
  remove_artifacts = TRUE,
  artifact_interval = c("full", "tR"),
  artifact_p_threshold = 1e-05,
  threshold_quantile = 0.98,
  time_step = 5L
)
```

Arguments

x	numeric matrix of single-trial evoked voltages with shape time x trials (i.e. rows are timepoints, columns are trials); the matrix orientation matches the variable V / data in the MATLAB reference. At least two trials are required.
time	numeric vector of length nrow(x) giving the stimulus-aligned time (in seconds) of each row of x; must be monotonically increasing and span [t_start, t_end].
t_start, t_end	numeric scalars, post-stimulation start and end times (in seconds) defining the analysis window. Defaults match the MATLAB illustration (0.015 s to 1 s).
remove_artifacts	logical; if TRUE (the default), an initial CRP pass is run to identify outlier/artifact trials, which are then dropped before the final pass. See 'Details'.

`artifact_interval`
: character, one of `"full"` (the default, matching the active option in the MATLAB illustration) or `"tR"`; selects whether per-trial outlier statistics are computed on the projection magnitudes for the full window or only at the response duration τ_R.

`artifact_p_threshold`
: numeric, p-value threshold below which a trial is flagged as artifact (provided its mean projection is also below the cohort mean); defaults to `1e-5`.

`threshold_quantile`
: numeric in `(0, 1)`; the fraction of the peak mean projection magnitude used to derive the duration-uncertainty bounds `tau_R_lower` and `tau_R_upper`. Defaults to `0.98` as in the manuscript.

`time_step`
: integer, sampling step (in samples) used when sweeping candidate response duration; defaults to `5L`, matching the MATLAB `t_step`. Larger values are faster but smooth the projection profile.

Details

Briefly, the algorithm proceeds in three stages:

1. For a sweep of candidate durations k, compute pairwise L2-normalized cross-projection magnitudes between trials truncated to $[0, k]$. The duration that maximizes the mean projection magnitude is taken as the response duration τ_R.
2. Apply linear kernel-trick PCA to the trial matrix truncated to τ_R; the first principal component is the canonical response shape $C(t)$.
3. Project $C(t)$ into each trial to obtain per-trial weights α_k; the residual $\epsilon_k = V_k - \alpha_k C$ summarizes trial-by-trial deviation from the canonical shape.

Significance is assessed by a one-sided t-test on the off-diagonal projection magnitudes against zero, restricted to a non-overlapping subset of comparison pairs to avoid double-counting.

When `remove_artifacts = TRUE`, the function performs an initial `CRP` pass and runs an unpaired t-test for each trial comparing the projections it participates in against all other off-diagonal projections. Trials with `p < artifact_p_threshold` *and* mean projection below the cohort mean are dropped, and `CRP` is re-run.

Value

A named list with the following elements:

`parameters` A list of single-trial parameterizations (`crp_parms` in MATLAB):

- `C` Numeric vector of length T_R (timepoints up to τ_R), the canonical response shape $C(t)$: the first eigenvector of the linear kernel PCA on `V_tR`, unit-normalized ($\|C\| = 1$). The matching time axis is in `params_times`.
- `al` Numeric vector of length K (number of trials), the per-trial alpha coefficient $\alpha_k = C^\top V_k$: scalar projection of trial k onto $C(t)$. Larger magnitude means the trial resembles the canonical shape more strongly; sign reflects polarity relative to C.
- `al_p` Numeric vector of length K, alpha-prime $\alpha_k / \sqrt{T_R}$: `al` rescaled to remove the duration dependence from the unit-norm convention on C. Expressed in μV and comparable across electrodes or conditions with different τ_R.

`ep` Numeric matrix of shape $T_R \times K$, the per-trial residual $\epsilon_k(t) = V_k(t) - \alpha_k C(t)$ after the shared component is removed. Access trial k via `ep[, k]`.

`epep_root` Numeric vector of length K, $\|\epsilon_k\| = \sqrt{\epsilon_k^\top \epsilon_k}$: L2 norm of the residual per trial. Smaller values indicate the canonical shape describes that trial more faithfully.

`Vsnr` Numeric vector of length K, per-trial signal-to-noise $\alpha_k / \|\epsilon_k\|$. Values > 1 indicate the canonical component is larger than the residual.

`expl_var` Numeric vector of length K, per-trial explained variance $1 - \|\epsilon_k\|^2 / \|V_k\|^2$: fraction of each trial's energy accounted for by $\alpha_k C(t)$. Ranges in $[0, 1]$.

`tR` Numeric scalar, response duration τ_R in seconds: the time at which mean cross-trial projection magnitude is maximized.

`params_times` Numeric vector of length T_R, time axis for `C`, `V_tR`, `ep`, and `avg_trace_tR`.

`V_tR` Numeric matrix $T_R \times K$, trial matrix truncated to τ_R - the data actually decomposed.

`avg_trace_tR` Numeric vector of length T_R, simple trial average truncated to τ_R.

`projections` A list of projection-stage outputs (`crp_projs` in MATLAB):

`proj_tpts` Numeric vector, candidate duration time points (seconds) at which projection magnitudes were evaluated.

`S_all` Numeric matrix; rows are non-redundant off-diagonal trial-pair projections, columns correspond to `proj_tpts`. Units: $\mu V \cdot s^{1/2}$.

`mean_proj_profile` Numeric vector, mean of `S_all` across trial pairs at each candidate duration, the profile whose maximum defines τ_R.

`var_proj_profile` Numeric vector, variance of `S_all` across trial pairs at each candidate duration.

`tR_index` Integer, column index into `S_all` and `proj_tpts` corresponding to τ_R.

`avg_trace_input` Numeric vector, simple trial average over the full analysis window (not truncated to τ_R).

`stat_indices` Integer vector, row indices of `S_all` used for the significance t-tests, constructed so each trial-pair comparison appears at most once.

`t_value_tR`, `p_value_tR` t-statistic and one-sided p-value (H1: mean projection > 0) at τ_R. Primary extraction-significance test reported in the manuscript.

`t_value_full`, `p_value_full` Same test at the full analysis-window duration.

`bad_trials` Integer vector of column indices into the original `x` flagged and removed as artifacts; `integer(0)` when none removed or when `remove_artifacts = FALSE`.

`tau_R` Numeric scalar, estimated response duration τ_R in seconds (convenience copy of `parameters$tR`).

`tau_R_lower`, `tau_R_upper` Numeric scalars, lower and upper threshold-crossing times (seconds) bracketing τ_R at the `threshold_quantile` fraction of the peak mean projection magnitude.

`t_start`, `t_end` The analysis window used.

`sample_rate` Numeric, sampling rate inferred from `time`.

References

The `CRP` algorithm is described in doi:10.1371/journal.pcbi.1011105, with a reference MATLAB implementation at `https://github.com/kaijmiller/crp_scripts`. See `citation("ravetools")` for the full bibliographic entry.

Examples

```
set.seed(42)

# Synthetic CCEP-like data: shared canonical shape with per-trial scaling
n_time   <- 500L
n_trials <- 20L
tt <- seq(-0.05, 1, length.out = n_time)
canonical <- exp(-((tt - 0.10) / 0.05)^2) -
             0.5 * exp(-((tt - 0.30) / 0.10)^2)
V <- (outer(canonical, runif(n_trials, 0.5, 1.5)) +
       matrix(rnorm(n_time * n_trials, sd = 0.3), n_time, n_trials)) * 2

res <- crp(V, tt)

op <- par(mfrow = c(1, 3), mar = c(4.5, 4, 3, 1))
on.exit({ par(op) })

# ---- Panel 1: all trials (full window) + mean + C(t) overlay ----------
parms <- res$parameters
matplot(tt, V, type = "l", lty = 1,
        col = "#80808060", xlab = "Time (s)",
        ylab = expression(mu * V),
        main = expression("Canonical shape " * C(t)))
# scale C(t) to the amplitude of the mean trace for overlay;
# C(t) ends at tau_R so the line is cut off there naturally
C_scaled <- parms$C * max(abs(rowMeans(V))) / max(abs(parms$C))
lines(parms$params_times, C_scaled, col = "#FFFF0080", lwd = 3)

# Mean
lines(tt, rowMeans(V), col = "black", lwd = 1)
legend("topright", c("mean", "C(t) scaled"),
       col = c("black", "#FFFF00"), lty = c(1, 2), lwd = 2,
       bty = "n", cex = 0.8)

# ---- Panel 2: per-trial alpha-prime weights ----------------------------
barplot(sort(parms$al_p), col = "steelblue", border = NA, las = 1,
        xlab = "Trial (sorted)",
        ylab = expression(alpha * "'" ~ (mu * V)),
        main = expression("Per-trial " * alpha * "' (alpha-prime)"))
abline(h = c(0, mean(parms$al_p)), lty = 2)

# ---- Panel 3: mean projection profile with tau_R bounds ----------------
proj <- res$projections
plot(proj$proj_tpts, proj$mean_proj_profile, type = "l", lwd = 2,
     xlab = "Candidate duration (s)",
     ylab = expression(bar(S) ~ (mu * V %.% s^{0.5})),
     main = expression("Projection profile & " * tau[R]),
     las = 1)
abline(v = c(res$tau_R_lower, res$tau_R, res$tau_R_upper),
       col = c("cyan3", "orange2", "red"),
       lty = c(2, 1, 2), lwd = 2)
legend("topright",
```

```
        legend = expression(tau[lb], tau[R], tau[ub]),
        col = c("cyan3", "orange2", "red"),
        lty = c(2, 1, 2), lwd = 2, bty = "n")

par(op)
```

`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'`

Details

This function is migrated from `gsignal` package, but with padding and indexing fixed. The results agree with 'Matlab'.

Value

Decimated signal

Examples

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

# compare with signal package
z <- gsignal::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")
```

`design_filter` *Design a digital filter*

Description

Provides 'FIR' and 'IIR' filter options; default is 'FIR', see also `design_filter_fir`; for 'IIR' filters, see `design_filter_iir`.

Usage

```
design_filter(
  sample_rate,
  data = NULL,
 method = c("fir_kaiser", "firls", "fir_remez", "butter", "cheby1", "cheby2", "ellip"),
  high_pass_freq = NA,
  high_pass_trans_freq = NA,
  low_pass_freq = NA,
  low_pass_trans_freq = NA,
  passband_ripple = 0.1,
  stopband_attenuation = 40,
  filter_order = NA,
  use_sos = TRUE,
  ...,
  data_size = length(data)
)
```

Arguments

`sample_rate`	data sample rate
`data`	data to be filtered, can be optional (`NULL`)
`method`	filter method, options are `"fir"` (default), `"butter"`, `"cheby1"`, `"cheby2"`, and `"ellip"`
`high_pass_freq, low_pass_freq`	high-pass or low-pass frequency, see `design_filter_fir` or `design_filter_iir`
`high_pass_trans_freq, low_pass_trans_freq`	transition bandwidths, see `design_filter_fir` or `design_filter_iir`
`passband_ripple`	allowable pass-band ripple in decibel; default is `0.1`
`stopband_attenuation`	minimum stop-band attenuation (in decibel) at transition frequency; default is `40` dB.
`filter_order`	suggested filter order; for 'IIR' methods, see `design_filter_iir` for details on how this interacts with `use_sos`; for 'FIR' methods 'RAVE' may or may not adopt this suggestion depending on the data and numerical feasibility

use_sos	logical; passed to design_filter_iir for 'IIR' methods (ignored for 'FIR'). When TRUE (default), filtering is done via 'SOS' form using gsignal::filtfilt; when FALSE, the ravetools 'ARMA' filtfilt is used.
...	passed to filter generator functions
data_size	used by 'FIR' filter design to determine maximum order, ignored in 'IIR' filters; automatically derived from data

Value

If data is specified and non-empty, this function returns filtered data via forward and backward filtfilt; if data is NULL, then returns the generator function.

Examples

```
sample_rate <- 200
t <- seq(0, 10, by = 1 / sample_rate)
x <- sin(t * 4 * pi) + sin(t * 20 * pi) +
  2 * sin(t * 120 * pi) + rnorm(length(t), sd = 0.4)

# ---- Using FIR ------------------------------------------------

# Low-pass filter
y1 <- design_filter(
  data = x,
  sample_rate = sample_rate,
  low_pass_freq = 3, low_pass_trans_freq = 0.5
)

# Band-pass cheby1 filter 8-12 Hz with custom transition
y2 <- design_filter(
  data = x,
  method = "cheby1",
  sample_rate = sample_rate,
  low_pass_freq = 12, low_pass_trans_freq = .25,
  high_pass_freq = 8, high_pass_trans_freq = .25
)

y3 <- design_filter(
  data = x,
  sample_rate = sample_rate,
  low_pass_freq = 80,
  high_pass_freq = 30
)

oldpar <- par(mfrow = c(2, 1),
              mar = c(3.1, 2.1, 3.1, 0.1))
plot(t, x, type = 'l', xlab = "Time", ylab = "",
     main = "Mixture of 2, 10, and 60Hz", xlim = c(0,1))
# lines(t, y, col = 'red')
lines(t, y3, col = 'green')
lines(t, y2, col = 'blue')
```

```
lines(t, y1, col = 'red')
legend(
  "topleft", c("Input", "Low: 3Hz", "Pass 8-12Hz", "Pass 30-80Hz"),
  col = c(par("fg"), "red", "blue", "green"), lty = 1,
  cex = 0.6
)

# plot pwelch
pwelch(x, fs = sample_rate, window = sample_rate * 2,
       noverlap = sample_rate, plot = 1, ylim = c(-100, 10))
pwelch(y1, fs = sample_rate, window = sample_rate * 2,
       noverlap = sample_rate, plot = 2, col = "red")
pwelch(y2, fs = sample_rate, window = sample_rate * 2,
       noverlap = sample_rate, plot = 2, col = "blue")
pwelch(y3, fs = sample_rate, window = sample_rate * 2,
       noverlap = sample_rate, plot = 2, col = "green")


# ---- Clean this demo ----------------------------------------------------
par(oldpar)
```

`design_filter_fir` *Design* `'FIR'` *filter using* `firls`

Description

Design `'FIR'` filter using `firls`

Usage

```
design_filter_fir(
  sample_rate,
  filter_order = NA,
  data_size = NA,
  high_pass_freq = NA,
  high_pass_trans_freq = NA,
  low_pass_freq = NA,
  low_pass_trans_freq = NA,
  stopband_attenuation = 40,
  scale = TRUE,
  method = c("kaiser", "firls", "remez")
)
```

Arguments

`sample_rate`	sampling frequency
`filter_order`	filter order, leave `NA` (default) if undecided

data_size minimum length of data to apply the filter, used to decide the maximum filter order. For 'FIR' filter, data length must be greater than 3xfilter_order

high_pass_freq high-pass frequency; default is NA (no high-pass filter will be applied)

high_pass_trans_freq
high-pass frequency band-width; default is automatically inferred from data size. Frequency high_pass_freq - high_pass_trans_freq is the corner of the stop-band

low_pass_freq low-pass frequency; default is NA (no low-pass filter will be applied)

low_pass_trans_freq
low-pass frequency band-width; default is automatically inferred from data size. Frequency low_pass_freq + low_pass_trans_freq is the corner of the stop-band

stopband_attenuation
allowable power attenuation (in decibel) at transition frequency; default is 40 dB.

scale whether to scale the filter for unity gain

method method to generate 'FIR' filter, default is using kaiser estimate, other choices are firls (with hamming window) and remez design.

Details

Filter type is determined from high_pass_freq and low_pass_freq. High-pass frequency is ignored if high_pass_freq is NA, hence the filter is low-pass filter. When low_pass_freq is NA, then the filter is high-pass filter. When both high_pass_freq and low_pass_freq are valid (positive, less than 'Nyquist'), then the filter is a band-pass filter if band-pass is less than low-pass frequency, otherwise the filter is band-stop.

Although the peak amplitudes are set at 1 by low_pass_freq and high_pass_freq, the transition from peak amplitude to zero require a transition, which is tricky but also important to set. When 'FIR' filters have too steep transition boundaries, the filter tends to have ripples in peak amplitude, introducing artifacts to the final signals. When the filter is too flat, components from unwanted frequencies may also get aliased into the filtered signals. Ideally, the transition bandwidth cannot be too steep nor too flat. In this function, users may control the transition frequency bandwidths via low_pass_trans_freq and high_pass_trans_freq. The power at the end of transition is defined by stopband_attenuation, with default value of 40 (i.e. -40 dB, this number is automatically negated during the calculation). By design, a low-pass 5 Hz filter with 1 Hz transition bandwidth results in around -40 dB power at 6 Hz.

Value

'FIR' filter in 'Arma' form.

Examples

```
# ---- Basic -----------------------------

sample_rate <- 500
data_size <- 1000
```

```
# low-pass at 5 Hz, with auto transition bandwidth
# from kaiser's method, with default stopband attenuation = 40 dB
filter <- design_filter_fir(
  low_pass_freq = 5,
  sample_rate = sample_rate,
  data_size = data_size
)

# Passband ripple is around 0.08 dB
# stopband attenuation is around 40 dB
print(filter)

diagnose_filter(
  filter$b, filter$a,
  fs = sample_rate,
  n = data_size,
  cutoffs = c(-3, -6, -40),
  vlines = 5
)

# ---- Advanced -------------------------------------------

sample_rate <- 500
data_size <- 1000

# Rejecting 3-8 Hz, with transition bandwidth 0.5 Hz at both ends
# Using least-square (firls) to generate FIR filter
# Suggesting the filter order n=160
filter <- design_filter_fir(
  low_pass_freq = 3, low_pass_trans_freq = 0.5,
  high_pass_freq = 8, high_pass_trans_freq = 0.5,
  filter_order = 160,
  sample_rate = sample_rate,
  data_size = data_size,
  method = "firls"
)

#
print(filter)

diagnose_filter(
  filter$b, filter$a,
  fs = sample_rate,
  n = data_size,
  cutoffs = c(-1, -40),
  vlines = c(3, 8)
)
```

`design_filter_iir` *Design an 'IIR' filter*

Description

Design an 'IIR' filter

Usage

```
design_filter_iir(
  method = c("butter", "cheby1", "cheby2", "ellip"),
  sample_rate,
  filter_order = NA,
  use_sos = TRUE,
  high_pass_freq = NA,
  high_pass_trans_freq = NA,
  low_pass_freq = NA,
  low_pass_trans_freq = NA,
  passband_ripple = 0.1,
  stopband_attenuation = 40
)
```

Arguments

`method`	filter method name, choices are `"butter"`, `"cheby1"`, `"cheby2"`, and `"ellip"`
`sample_rate`	sampling frequency
`filter_order`	suggested filter order. When `use_sos=TRUE` (default) this order is used as-is since 'SOS' form is numerically stable at any order. When `use_sos=FALSE` and 'ARMA' form becomes numerically unstable, the order will be automatically reduced with a warning; leave `NA` (default) to let the function choose automatically.
`use_sos`	logical; when `TRUE` (default), the filter is also stored as a `Sos` (second-order sections) object in `filter$sos`, and no stability-based order correction is performed. When `FALSE`, the order is capped at the maximum stable 'ARMA' order (with a warning if a specific `filter_order` was requested).
`high_pass_freq`	high-pass frequency; default is `NA` (no high-pass filter will be applied)
`high_pass_trans_freq`	high-pass frequency band-width; default is automatically inferred from filter type.
`low_pass_freq`	low-pass frequency; default is `NA` (no low-pass filter will be applied)
`low_pass_trans_freq`	low-pass frequency band-width; default is automatically inferred from filter type.
`passband_ripple`	allowable pass-band ripple in decibel; default is `0.1`

`stopband_attenuation`
: minimum stop-band attenuation (in decibel) at transition frequency; default is `40` dB.

Value

A filter object with $b/$a 'ARMA' coefficients and `$sos` second-order sections (a `gsignal` `Sos` object).

Examples

```
sample_rate <- 500

my_diagnose <- function(
    filter, vlines = c(8, 12), cutoffs = c(-3, -6)) {
  diagnose_filter(
    b = filter$b,
    a = filter$a,
    fs = sample_rate,
    vlines = vlines,
    cutoffs = cutoffs
  )
}

# ---- Default using butterworth to generate 8-12 bandpass filter ----

# Butterworth filter with cut-off frequency
# 7 ~ 13 (default transition bandwidth is 1Hz) at -3 dB
filter <- design_filter_iir(
  method = "butter",
  low_pass_freq = 12,
  high_pass_freq = 8,
  sample_rate = 500
)

filter

my_diagnose(filter)

## explicit bandwidths and attenuation (sharper transition)

# Butterworth filter with cut-off frequency
# passband ripple is 0.5 dB (8-12 Hz)
# stopband attenuation is 40 dB (5-18 Hz)
filter <- design_filter_iir(
  method = "butter",
  low_pass_freq = 12, low_pass_trans_freq = 6,
  high_pass_freq = 8, high_pass_trans_freq = 3,
  sample_rate = 500,
  passband_ripple = 0.5,
  stopband_attenuation = 40
)
```

```
filter

my_diagnose(filter)

# ---- cheby1 --------------------------------

filter <- design_filter_iir(
  method = "cheby1",
  low_pass_freq = 12,
  high_pass_freq = 8,
  sample_rate = 500
)

my_diagnose(filter)

# ---- cheby2 --------------------------------

filter <- design_filter_iir(
  method = "cheby2",
  low_pass_freq = 12,
  high_pass_freq = 8,
  sample_rate = 500
)

my_diagnose(filter)

# ----- ellip --------------------------------

filter <- design_filter_iir(
  method = "ellip",
  low_pass_freq = 12,
  high_pass_freq = 8,
  sample_rate = 500
)

my_diagnose(filter)
```

`detrend` *Remove the trend for one or more signals*

Description

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

Usage

```
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

```
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

```
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,
  yline = 2.66 * cex,
  tck = -0.005 * (3 + 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

```
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)
```

---

diagnose_filter *Diagnose digital filter*

Description

Generate frequency response plot with sample-data simulation

Usage

```
diagnose_filter(
  b,
  a,
  fs,
  n = 512,
  whole = FALSE,
  sample = stats::rnorm(n, mean = sample_signal(n), sd = 0.2),
  vlines = NULL,
  xlim = "auto",
  cutoffs = c(-3, -6, -12)
)
```

Arguments

b	the moving-average coefficients of an ARMA model
a	the auto-regressive coefficients of an ARMA filter; default is 1
fs	sampling frequency in Hz

n	number of points at which to evaluate the frequency response; default is 512
whole	whether to evaluate beyond Nyquist frequency; default is false
sample	sample signal of length n for simulation
vlines	additional vertical lines (frequencies) to plot
xlim	frequency limit of frequency response plot; default is "auto", can be "full" or a numeric of length 2
cutoffs	cutoff decibel powers to draw on the frequency plot, also used to calculate the frequency limit when xlim is "auto"

Value

Nothing

Examples

```
library(ravetools)

# sample rate
srate <- 500

# signal length
npts <- 1000

# band-pass
bpass <- c(1, 50)

# Nyquist
fn <- srate / 2
w <- bpass / fn

# ---- FIR filter -------------------------------------------------
order <- 160

# FIR1 is MA filter, a = 1
filter <- fir1(order, w, "pass")

diagnose_filter(
  b = filter$b, a = filter$a, n = npts,
  fs = srate, vlines = bpass
)

# ---- Butter filter ----------------------------------------------
filter <- butter(3, w, "pass")

diagnose_filter(
  b = filter$b, a = filter$a, n = npts,
  fs = srate, vlines = bpass
)
```

`dijkstras-path` *Calculate distances along a surface*

Description

Calculate surface distances of graph or mesh using 'Dijkstra' method.

Usage

```
dijkstras_surface_distance(
  positions,
  faces,
  start_node,
  face_index_start = NA,
  max_search_distance = NA,
  ...
)

surface_path(x, target_node)
```

Arguments

`positions`	numeric matrix with no `NA` values. The number of row is the total count of nodes (vertices), and the number of columns represent the node dimension. Each row represents a node.
`faces`	integer matrix with each row containing indices of nodes. For graphs, `faces` is a matrix with two columns defining the connecting edges; for '3D' mesh, `faces` is a three-column matrix defining the face index of mesh triangles.
`start_node`	integer, row index of `positions` on where to start calculating the distances. This integer must be 1-indexed and cannot exceed the total number of `positions` rows
`face_index_start`	integer, the start of the nodes in `faces`; please specify this input explicitly if the first node is not contained in `faces`. Default is `NA` (determined by the minimal number in `faces`). The reason to set this input is because some programs use `1` to represent the first node, some start from `0`.
`max_search_distance`	numeric, maximum distance to iterate; default is `NA`, that is to iterate and search the whole mesh
`...`	reserved for backward compatibility
`x`	distance calculation results returned by `dijkstras_surface_distance` function
`target_node`	the target node number to reach (from the starting node); `target_node` is always 1-indexed.

Value

`dijkstras_surface_distance` returns a list distance table with the meta configurations. `surface_path` returns a data frame of the node ID (from `start_node` to `target_node`) and cumulative distance along the shortest path.

Examples

```
# ---- Toy example --------------------

# Position is 2D, total 6 points
positions <- matrix(runif(6 * 2), ncol = 2)

# edges defines connected nodes
edges <- matrix(ncol = 2, byrow = TRUE, data = c(
  1,2,
  2,3,
  1,3,
  2,4,
  3,4,
  2,5,
  4,5,
  2,5,
  4,6,
  5,6
))

# calculate distances
ret <- dijkstras_surface_distance(
  start_node = 1,
  positions = positions,
  faces = edges,
  face_index_start = 1
)

# get shortest path from the first node to the last
path <- surface_path(ret, target_node = 6)

# plot the results
from_node <- path$path[-nrow(path)]
to_node <- path$path[-1]
plot(positions, pch = 16, axes = FALSE,
     xlab = "X", ylab = "Y", main = "Dijkstra's shortest path")
segments(
  x0 = positions[edges[,1],1], y0 = positions[edges[,1],2],
  x1 = positions[edges[,2],1], y1 = positions[edges[,2],2]
)

points(positions[path$path,], col = "steelblue", pch = 16)
arrows(
  x0 = positions[from_node,1], y0 = positions[from_node,2],
  x1 = positions[to_node,1], y1 = positions[to_node,2],
  col = "steelblue", lwd = 2, length = 0.1, lty = 2
```

```
)

points(positions[1,,drop=FALSE], pch = 16, col = "orangered")
points(positions[6,,drop=FALSE], pch = 16, col = "purple3")

# ---- Example with mesh ---------------------------------

## Not run:

  # Please install the down-stream package `threeBrain`
  # and call library(threeBrain)
  # the following code set up the files

  read.fs.surface <- internal_rave_function(
    "read.fs.surface", "threeBrain")
  default_template_directory <- internal_rave_function(
    "default_template_directory", "threeBrain")
  surface_path <- file.path(default_template_directory(),
                            "N27", "surf", "lh.pial")
  if (!file.exists(surface_path)) {
    internal_rave_function(
      "download_N27", "threeBrain")()
  }

  # Example starts from here --->
  # Load the mesh
  mesh <- read.fs.surface(surface_path)

  # Calculate the path with maximum radius 100
  ret <- dijkstras_surface_distance(
    start_node = 1,
    positions = mesh$vertices,
    faces = mesh$faces,
    max_search_distance = 100,
    verbose = TRUE
  )

  # get shortest path from the first node to node 43144
  path <- surface_path(ret, target_node = 43144)

  # plot
  from_nodes <- path$path[-nrow(path)]
  to_nodes <- path$path[-1]
  # calculate colors
  pal <- colorRampPalette(
    colors = c("red", "orange", "orange3", "purple3", "purple4")
  )(1001)
  col <- pal[ceiling(
    path$distance / max(path$distance, na.rm = TRUE) * 1000
  ) + 1]
  oldpar <- par(mfrow = c(2, 2), mar = c(0, 0, 0, 0))
  for(xdim in c(1, 2, 3)) {
    if ( xdim < 3 ) {
```

```
    ydim <- xdim + 1
  } else {
    ydim <- 3
    xdim <- 1
  }
  plot(
    mesh$vertices[, xdim], mesh$vertices[, ydim],
    pch = ".", col = "#BEBEBE33", axes = FALSE,
    xlab = "P - A", ylab = "S - I", asp = 1
  )
  segments(
    x0 = mesh$vertices[from_nodes, xdim],
    y0 = mesh$vertices[from_nodes, ydim],
    x1 = mesh$vertices[to_nodes, xdim],
    y1 = mesh$vertices[to_nodes, ydim],
    col = col
  )
}

# plot distance map
distances <- ret$paths$distance
col <- pal[ceiling(distances / max(distances, na.rm = TRUE) * 1000) + 1]
selection <- !is.na(distances)

plot(
  mesh$vertices[, 2], mesh$vertices[, 3],
  pch = ".", col = "#BEBEBE33", axes = FALSE,
  xlab = "P - A", ylab = "S - I", asp = 1
)
points(
  mesh$vertices[selection, c(2, 3)],
  col = col[selection],
  pch = "."
)

# reset graphic state
par(oldpar)


## End(Not run)
```

---

ensure_mesh3d *Coerce a surface object to a* 'mesh3d' *mesh*

Description

Internal helper used throughout **ravetools** to accept a variety of surface representations and return a list of class `'mesh3d'` (the format used by the `rgl` package). Most mesh-consuming functions in this package call `ensure_mesh3d` on their surface arguments, so the coercion rules described here apply to all of them.

Usage

```
ensure_mesh3d(surface)
```

Arguments

`surface`	a surface object. One of the following: `'mesh3d'` returned unchanged. `'fs.surface'` a surface in the format produced by `freesurferformats::read.fs.surface`. Vertices and faces are copied into a `'mesh3d'` list; zero-indexed faces are bumped by 1. `'ieegio_surface'` a surface object produced by **ieegio** (e.g. from `read_surface` or `volume_to_surface`). See the section below for important behavior changes. **a bare** `list` must contain a numeric `vb` matrix with 3 or 4 rows; the list is reclassified as `'mesh3d'` with no other modification. Any other input triggers an error.

Value

An object of class `'mesh3d'` with at least the `vb` (vertex) component and, when face information is available, an `it` (triangle index) component.

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
# mesh3d input is returned unchanged in shape
sphere <- vcg_sphere()
m <- ensure_mesh3d(sphere)
identical(m$vb, sphere$vb)

# A bare list with a `vb` slot is reclassified to mesh3d
bare <- list(vb = rbind(matrix(rnorm(30), nrow = 3), 1))
m <- ensure_mesh3d(bare)
inherits(m, "mesh3d")
```

---

fast_cov *Calculate massive covariance matrix in parallel*

Description

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

Usage

```
fast_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

```
# 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)



# 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
)
```

`fast_quantile` *Compute quantiles*

Description

Compute quantiles

Usage

```
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

```
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"
)
```

---

fftw-internal *Low-level* **FFTW3** *wrappers*

Description

Thin R bindings around the **FFTW3** library. These are **low-level** routines exposed primarily for advanced users and other packages that need maximum throughput. They perform minimal input checking and follow **FFTW** conventions (e.g. unnormalized inverse transforms, one-sided real-to-complex spectra). For most user code prefer `fft`, `mvfft`, or higher-level helpers in this package such as `convolve`, `pwelch`, `multitaper`, and the filtering utilities.

Warning: the API is intentionally close to **FFTW**'s C interface and may change between releases. Outputs match the corresponding base R transforms up to floating-point round-off.

Arguments

`data`	Numeric (real) or complex input. For 2D/3D variants, a matrix or 3-dimensional array. For `mvfftw_*` functions, a matrix whose columns are transformed independently.
`HermConj`	Integer `0` or `1`. When `1`, return the full Hermitian-symmetric spectrum of length `N` (matching `stats::fft`); when `0`, return only the non-redundant one-sided half of length `floor(N/2) + 1`.
`inverse`	Integer `0` (forward) or `1` (inverse). The inverse transform is **unnormalized**; divide by the number of elements to invert a forward transform, mirroring `stats::fft(., inverse = TRUE)`.
`fftwplanopt`	Integer planner effort: `0` = `FFTW_ESTIMATE` (default, fast planning), `1` = `FFTW_MEASURE`, `2` = `FFTW_PATIENT`, `3` = `FFTW_EXHAUSTIVE`.
`retrows`	Integer; expected number of rows of the time-domain signal for `mvfftw_c2r` when reconstructing from a one-sided spectrum.
`ret`	Optional reusable output buffer of the correct type and length; pass `NULL` (default) to let the function allocate one. This input is for advanced users; leave `NULL` if you do not know what you are d

Details

All functions preserve their `data` argument: the input buffer is copied internally before planning when needed, so callers may safely reuse `data` after the call. For multi-dimensional variants, axis ordering follows R (column-major) conventions.

Value

A complex (or real, for `*_c2r`) vector / matrix / array matching the corresponding base R transform up to floating-point error.

Examples

```
set.seed(1)

## --- 1D real-to-complex ---------------------------------------------------
x <- rnorm(16)
a <- ravetools::fftw_r2c(x, HermConj = 1)
b <- stats::fft(x)
all.equal(a, b)                                 # TRUE (within tol)
```

```
# one-sided spectrum (length floor(N/2)+1)
a_half <- ravetools::fftw_r2c(x, HermConj = 0)
all.equal(a_half, b[seq_len(length(x) %/% 2 + 1)])

## --- 1D complex-to-complex ---------------------------------------------
z <- complex(real = rnorm(16), imaginary = rnorm(16))
all.equal(ravetools::fftw_c2c(z, inverse = 0), stats::fft(z))
all.equal(ravetools::fftw_c2c(z, inverse = 1), stats::fft(z, inverse = TRUE))

## --- 1D complex-to-real (inverse of fftw_r2c) --------------------------
# Using the full Hermitian spectrum:
xr <- ravetools::fftw_c2r(a, HermConj = 1) / length(x)
all.equal(xr, x)

## --- Multivariate (column-wise) ----------------------------------------
M <- matrix(rnorm(32), nrow = 8, ncol = 4)
all.equal(ravetools::mvfftw_r2c(M, HermConj = 1), stats::mvfft(M + 0i))

Mz <- matrix(complex(real = rnorm(32), imaginary = rnorm(32)),
             nrow = 8, ncol = 4)
all.equal(ravetools::mvfftw_c2c(Mz, inverse = 0), stats::mvfft(Mz))
all.equal(ravetools::mvfftw_c2c(Mz, inverse = 1),
          stats::mvfft(Mz, inverse = TRUE))

# one-sided -> back to real signal
Mh <- ravetools::mvfftw_r2c(M, HermConj = 0)
Mr <- ravetools::mvfftw_c2r(Mh, retrows = nrow(M)) / nrow(M)
all.equal(Mr, M)

## --- 2D ----------------------------------------------------------------
X2 <- matrix(rnorm(20), nrow = 5, ncol = 4)
all.equal(ravetools::fftw_r2c_2d(X2, HermConj = 1), stats::fft(X2 + 0i))

Z2 <- matrix(complex(real = rnorm(20), imaginary = rnorm(20)),
             nrow = 5, ncol = 4)
all.equal(ravetools::fftw_c2c_2d(Z2, inverse = 0), stats::fft(Z2))
all.equal(ravetools::fftw_c2c_2d(Z2, inverse = 1),
          stats::fft(Z2, inverse = TRUE))

## --- 3D ----------------------------------------------------------------
X3 <- array(rnorm(60), dim = c(5, 4, 3))
all.equal(ravetools::fftw_r2c_3d(X3, HermConj = 1), stats::fft(X3 + 0i))

Z3 <- array(complex(real = rnorm(60), imaginary = rnorm(60)),
            dim = c(5, 4, 3))
all.equal(ravetools::fftw_c2c_3d(Z3, inverse = 0), stats::fft(Z3))
all.equal(ravetools::fftw_c2c_3d(Z3, inverse = 1),
          stats::fft(Z3, inverse = TRUE))
```

`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.

Usage

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

Arguments

`surface`	a surface mesh; accepted classes include `'mesh3d'` (from `rgl`), `'fs.surface'` (from `freesurferformats`), `'ieegio_surface'` (from **ieegio**), or a bare list containing a `vb` vertex matrix; see `ensure_mesh3d` for details and for how `'ieegio_surface'` inputs are coerced
`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

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Author(s)

Zhengjia Wang

See Also

`ensure_mesh3d`

Examples

```
# takes > 5s to run example

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

fill_surface(surface, preview = TRUE)
```

`filter-window` *Filter window functions*

Description

Filter window functions

Usage

```
hanning(n)

hamming(n)

blackman(n)

blackmannuttall(n)

blackmanharris(n)

flattopwin(n)

bohmanwin(n)
```

Arguments

n	number of time-points in window

Value

A numeric vector of window with length n

Examples

```
hanning(10)
hamming(11)
blackmanharris(21)
```

`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. For double filters (forward-backward), see `filtfilt`.

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

```
t <- seq(0, 1, by = 0.01)
x <- sin(2 * pi * t * 2.3)
bf <- gsignal::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,[.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

```
filtfilt(b, a = 1, x)
```

Arguments

b	one-dimensional real numerical vector, the moving-average coefficients of an ARMA filter; alternatively, a Sos (second-order sections) object from gsignal, in which case a is ignored and gsignal::filtfilt is used internally
a	the auto-regressive (recursive) coefficients of an ARMA filter; ignored when b is a Sos object
x	numerical vector or matrix input (real value)

Value

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

Examples

```
t <- seq(0, 1, by = 0.01)
x <- sin(2 * pi * t * 2.3)
bf <- gsignal::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)
```

---

find_peaks *Find peaks of a signal*

Description

Find peaks of a signal

Usage

```
find_peaks(x, min_val = NA, min_distance = 4, min_width = 2)
```

Arguments

x	a numeric vector without missing values
min_val	find peaks that are greater than this value
min_distance	merge peaks that are less than min_distance time-points away
min_width	search radius (time-points) on whether the peak is "local"; this is for seasonal oscillations.

Value

A list of peak index (1-based) and the corresponding value.

Examples

```
# Basic example
x <- sin(seq(0, 10, 0.01)) + rnorm(1001) * 0.1

peaks <- find_peaks(x)

plot(x, type = 'l')
abline(v = peaks$index, col = 'red')

# merge peaks that are close
peaks <- find_peaks(x, min_distance = 400)

plot(x, type = 'l')
abline(v = peaks$index, col = 'red')

# with or without min_width
x <- c(0, 1, 0.5, 0.9, 0.2, 0.8, 0.2, 0.75, 0)

# without min_width
peaks <- find_peaks(x, min_width = 0)
plot(x, type = 'l')
abline(v = peaks$index, col = 'red')


# with min_width=2: t=4 is greater than t=6
peaks <- find_peaks(x, min_width = 2)
plot(x, type = 'l')
abline(v = peaks$index, col = 'red')
```

---

fir1 *Window-based* FIR *filter design*

Description

Generate a `fir1` filter that is checked against `Matlab` `fir1` function.

Usage

```
fir1(
  n,
  w,
  type = c("low", "high", "stop", "pass", "DC-0", "DC-1"),
```

```
  window = hamming,
  scale = TRUE,
  hilbert = FALSE
)
```

Arguments

n	filter order
w	band edges, non-decreasing vector in the range 0 to 1, where 1 is the Nyquist frequency. A scalar for high-pass or low-pass filters, a vector pair for band-pass or band-stop, or a vector for an alternating pass/stop filter.
type	type of the filter, one of "low" for a low-pass filter, "high" for a high-pass filter, "stop" for a stop-band (band-reject) filter, "pass" for a pass-band filter, "DC-0" for a band-pass as the first band of a multi-band filter, or "DC-1" for a band-stop as the first band of a multi-band filter; default "low"
window	smoothing window function or a numerical vector. The filter is the same shape as the smoothing window. When window is a function, window(n+1) will be called, otherwise the length of the window vector needs to have length of n+1; default: hamming
scale	whether to scale the filter; default is true
hilbert	whether to use 'Hilbert' transformer; default is false

Value

The FIR filter coefficients with class 'Arma'. The moving average coefficient is a vector of length n+1.

---

firls *Least-squares linear-phase* FIR *filter design*

Description

Produce a linear phase filter from the weighted mean squared such that error in the specified bands is minimized.

Usage

```
firls(N, freq, A, W = NULL, ftype = "", legacy = FALSE)
```

Arguments

N	filter order, must be even (if odd, then will be increased by one)
freq	vector of frequency points in the range from 0 to 1, where 1 corresponds to the Nyquist frequency.
A	vector of the same length as freq containing the desired amplitude at each of the points specified in freq.

W	weighting function that contains one value for each band that weights the mean squared error in that band. W must be half the length of freq.
ftype	transformer type; default is ""; alternatively, 'h' or 'hilbert' for 'Hilbert' transformer.
legacy	whether to use the legacy implementations, which uses qr.solve instead of faster chol

Value

The FIR filter coefficients with class 'Arma'. The moving average coefficient is a vector of length n+1.

---

freqz2 *Frequency response of digital filter*

Description

Compute the z-plane frequency response of an ARMA model.

Usage

```
freqz2(b, a = 1, fs = 2 * pi, n = 512, whole = FALSE, ...)
```

Arguments

b	the moving-average coefficients of an ARMA model
a	the auto-regressive coefficients of an ARMA filter; default is 1
fs	sampling frequency in Hz
n	number of points at which to evaluate the frequency response; default is 512
whole	whether to evaluate beyond Nyquist frequency; default is false
...	ignored

Value

A list of frequencies and corresponding responses in complex vector

gammatone_fast *Apply gamma-tone filters to obtain auditory envelopes*

Description

Apply gamma-tone filters to obtain auditory envelopes

Usage

```
gammatone_fast(
  x,
  sample_rate,
  center_frequencies,
  n_bands,
  use_hilbert = TRUE,
  downsample = NA,
  downsample_before_hilbert = FALSE
)
```

Arguments

x	a numeric vector or matrix; if x is a matrix, it should be column-major (each column is a sound track)
sample_rate	sampling frequency
center_frequencies	center frequencies at which the envelopes will be derived; can be either a length of two defining the lower and upper bound, and using n_bands to interpolate automatically, or a length of multiple, with the frequencies specified explicitly
n_bands	number of the center frequencies, can be missing if center_frequencies is explicit and no interpolation is needed; if specified, then the frequencies will be interpolated using equivalent rectangular bandwidth rate ('ERB')
use_hilbert	whether to apply 'Hilbert' transform; default is true, which calculates the magnitude; set to false when only the filter is needed
downsample	whether to down-sample the envelopes after the filters; default is NA (no down-sample).
downsample_before_hilbert	whether the down-sample happens before or after the 'Hilbert' transform so speed up the computation if the signal is too long; only used when downsample is greater than 1; default is FALSE. Use with caution, especially when the voice center frequency is close to the 'Nyquist' frequency. However, if used properly, there will be significant performance boost on large signals with high sampling rates

Value

A file-array object of filtered and potentially down-sampled data; see 'Examples' on how to use this function.

Examples

```
fs <- 4000
time <- seq_len(8000) / fs
x <- sin(160 * pi * time) +
  sin(1000 * pi * time) * dnorm(time, mean = 1, sd = 0.1) +
  0.5 * rnorm(length(time))

# envelope
result <- gammatone_fast(
  x,
  sample_rate = fs,
  center_frequencies = c(20, 1000),
  n_bands = 128,

  # default downsample happens after hilbert
  downsample = 40
)


oldpar <- par(mfrow = c(2, 1))

plot(
  time,
  x,
  type = "l",
  xlab = "Time",
  ylab = "",
  main = "Original mixed 80Hz and 500Hz"
)

# only one channel
envelope <- subset(result, Channel ~ Channel == 1, drop = TRUE)
dnames <-  dimnames(envelope)
image(
  x = as.numeric(dnames$Time),
  y = as.numeric(dnames$Frequency),
  z = envelope,
  xlab = "Time",
  ylab = "Frequency",
  main = "Envelope from 20Hz to 1000Hz"
)



par(oldpar) # reset graphics state
```

---

grow_volume *Grow volume mask*

Description

Grow volume mask

Usage

```
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

Value

A binary volume mask

Examples

```
oldpar <- 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)

par(oldpar)
```

internal_rave_function *Get external function from 'RAVE'*

Description

Internal function used for examples relative to 'RAVE' project and should not be used directly.

Usage

```
internal_rave_function(name, pkg, inherit = TRUE, on_missing = NULL)
```

Arguments

name	function or variable name
pkg	'RAVE' package name
inherit	passed to get0
on_missing	default value to return of no function is found

Value

Function object if found, otherwise on_missing.

left_hippocampus_mask *Left 'Hippocampus' of 'N27-Collin' brain*

Description

Left 'Hippocampus' of 'N27-Collin' brain

Usage

```
left_hippocampus_mask
```

Format

A three-mode integer mask array with values of 1 ('Hippocampus') and 0 (other brain tissues)

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

This function is soft-deprecated. Please use vcg_mesh_volume, vcg_uniform_remesh, and vcg_smooth_explicit or vcg_smooth_implicit.

Usage

```
mesh_from_volume(
  volume,
  output_format = c("rgl", "freesurfer"),
  IJK2RAS = NULL,
  threshold = 0,
  verbose = TRUE,
  remesh = TRUE,
  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 vcg_uniform_remesh
remesh_voxel_size, remesh_multisample, remesh_automerge	see arguments in vcg_uniform_remesh
smooth	whether to smooth the mesh via vcg_smooth_explicit
smooth_lambda, smooth_delta, smooth_method	see vcg_smooth_explicit

Value

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

Examples

```
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'
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

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

```
# Takes long to run

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()
)
```

---

naive_nmf *A naive implementation of non-negative matrix factorization*

Description

A pure-R vanilla implementation assuming inputs are non-negative matrices without NA.

Usage

```
naive_nmf(x, k, tol = c(1e-04, 1e-08), max_iters = 10000, verbose = TRUE)
```

Arguments

x	a matrix, or can be converted into a matrix; all negative or missing values will be treated as zero
k	decomposition rank
tol	stop criteria, a numeric of two; the first number is the tolerance for root-mean-squared residuals, relative to the largest number in x; the second number is the tolerance for weight differences; any stopping criteria met will result in the stop of iteration
max_iters	maximum iterations
verbose	whether to report the progress; logical or a positive integer (of step intervals)

Value

A list of weights (non-negative template matrix W and non-negative H) and errors (root mean squared error of fitted, matrix W, and W versus their previous iteration, respectively).

Examples

```
x <- stats::toeplitz(.9 ^ (0:31))

nmf <- naive_nmf(x, k = 7, verbose = FALSE)

fitted <- nmf$W %*% nmf$H

oldpar <- par(mfrow = c(1, 2))
on.exit({ par(oldpar )})

image(x, zlim = c(0, 1), main = "Input")
image(fitted, zlim = c(0, 1),
      main = sprintf("Fitted with rank=%d", nmf$k))
```

---

new_matrix4 *Create a* Matrix4 *instance for* 'Affine' *transform*

Description

Create a Matrix4 instance for 'Affine' transform

Usage

```
new_matrix4()

as_matrix4(m)
```

Arguments

m	a matrix or a vector to be converted to the Matrix4 instance; m must be one of the followings: for matrices, the dimension must be 4x4, 3x4 (the last row will be 0 0 0 1), or 3x3 (linear transform); for vectors, the length must be 16, 12 (will append 0 0 0 1 internally), 3 (translation), or 1 (scale).

Value

A Matrix4 instance

See Also

new_vector3, new_quaternion

---

new_quaternion *Create a* Quaternion *instance to store '3D' rotation*

Description

Create instances that mimic the 'three.js' syntax.

Usage

```
new_quaternion(x = 0, y = 0, z = 0, w = 1)

as_quaternion(q)
```

Arguments

x, y, z, w	numeric of length one
q	R object to be converted to Quaternion

Value

A Quaternion instance

See Also

new_vector3, new_matrix4

new_vector3 *Create a* Vector3 *instance to store '3D' points*

Description

Create instances that mimic the 'three.js' syntax.

Usage

```
new_vector3(x = 0, y = 0, z = 0)

as_vector3(v)
```

Arguments

x, y, z	numeric, must have the same length, 'xyz' positions
v	R object to be converted to Vector3 instance

Value

A Vector3 instance

See Also

new_matrix4, new_quaternion

Examples

```
vec3 <- new_vector3(
  x = 1:9,
  y = 9:1,
  z = rep(c(1,2,3), 3)
)

vec3[]

# transform
m <- new_matrix4()

# rotation xy plane by 30 degrees
m$make_rotation_z(pi / 6)

vec3$apply_matrix4(m)

vec3[]

as_vector3(c(1,2,3))
```

`notch_filter` *Apply 'Notch' filter*

Description

Apply 'Notch' filter

Usage

```
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

```
detect_threads()

ravetools_threads(n_threads = 2)
```

plane_geometry *Create a two-dimensional plane in three dimensional space*

Description

Create a two-dimensional plane in three dimensional space

Usage

```
plane_geometry(width = 1, height = 1, shape = c(2, 2))
```

Arguments

width, height	width and height of the plane, must not be NA
shape	length of two to indicate the number of vertices along width and height, default is only c(2, 2) (2 vertices each side, hence one grid)

Value

A triangular mesh of class 'mesh3d'

Examples

```
plane <- plane_geometry(5, 10, c(12, 22))

if(FALSE) {

  rgl_view({

    rgl_call("shade3d", plane, col = 3)
    rgl_call("wire3d", plane, col = 1)

  })

}
```

---

plot.ravetools_crp *Plot* CRP *results*

Description

S3 plot method for objects of class ravetools_crp returned by crp. Produces a three-panel figure:

1. Single-trial traces over the full analysis window with the mean and the scaled canonical shape $C(t)$ overlaid (the shape is drawn only up to τ_R, so the cut-off is itself informative).
2. Per-trial α' weights sorted in ascending order.
3. Mean cross-trial projection profile with vertical lines marking τ_{lb}, τ_R and τ_{ub}.

Usage

```
## S3 method for class 'ravetools_crp'
plot(x, ...)
```

Arguments

x	an object of class ravetools_crp as returned by crp.
...	additional graphical parameters passed to par (e.g. mar, cex.axis); currently unused beyond restoring the previous par state on exit.

Value

Invisibly returns x.

plot.ravetools_curve *Plot method for* ravetools_curve

Description

Plots a ravetools_curve object created by catmull_rom_3d. When the rgl package is available and use_rgl = TRUE (default), an interactive 3D scene is opened. Otherwise three 2-D projection panels (*x*-*y*, *x*-*z*, *y*-*z*) are drawn using base R graphics.

Usage

```
## S3 method for class 'ravetools_curve'
plot(x, n = 200L, col = "steelblue", pch = 19L, cex = 1, use_rgl = TRUE, ...)
```

Arguments

x	an object of class ravetools_curve.
n	integer; number of sample points used to draw the smooth curve. Default is 200L.
col	color for the spline curve line. Default "steelblue".
pch, cex	plotting character and scaling for the key control points (base-R fallback only). Default pch = 19, cex = 1.
use_rgl	logical; if TRUE (default) and the rgl package is installed, an interactive 3D window is used. Set to FALSE to force the base-R 2D projection panels.
...	additional graphical parameters forwarded to the underlying plot calls.

Value

Invisibly returns x.

plot_mesh_dotcloud *Render one or more meshes as an orthographic dot cloud in base R*

Description

Projects mesh vertices onto a 2D plane using an orthographic camera defined by an eye position, a look-at point, and an up direction, then draws the projected dots with base-R plot. Each dot is rendered opaque, but its cex is modulated by a rim-light weight $1 - |n \cdot z_{cam}|$: front- and back-facing vertices shrink toward zero size, while grazing-edge vertices keep their full size. This size-modulated trick gives a rim-light look without paying R's per-point transparency-blending cost.

Multiple meshes share a single depth space: all vertices are projected together and sorted globally by depth before a single plot() call, so the painter's algorithm works correctly across meshes. Meshes without faces (point clouds) are rendered at full size and are not affected by the side filter.

Usage

```
plot_mesh_dotcloud(
  mesh,
  eye = c(0, 0, 1000),
  lookat = c(0, 0, 0),
  up = c(0, 1, 0),
  col = c("white", "gray30"),
  pch = 16L,
  cex = 0.1,
  add = FALSE,
  axes = FALSE,
  asp = 1,
  xlim = NULL,
  ylim = NULL,
  zoom = 1,
  xlab = "",
  ylab = "",
  normal_weight = c("auto", "area", "angle"),
  side = c("front", "back", "both"),
  mesh_clipping = 0.7,
  alpha = 1,
  clipping_plane = NULL,
  clipping_plane_enabled = TRUE,
  ...
)
```

Arguments

mesh	a 'mesh3d' object, or a list of 'mesh3d' objects. Meshes without a face matrix ($it) are treated as point clouds. If mesh$normals is absent and the mesh has faces, normals are recomputed with vcg_update_normals.
eye	numeric vector of length 3 - camera position in world space.
lookat	numeric vector of length 3 - the world-space point the camera is looking at.
up	numeric vector of length 3 - a world-space vector indicating which direction is "up" for the camera; defaults to c(0, 1, 0).
col	base color(s) for the dots. Accepted forms: • A single color string - applied to all meshes. • A character vector of length 2 or more (not matching any vertex count) - used as a depth-ordered color gradient applied to all meshes. • A character vector of length equal to the number of vertices in a single mesh - per-vertex colors for that mesh. • A list of length equal to the number of meshes - each element is one of the above, applied to the corresponding mesh. Default "gray30".
pch	point character; a scalar or vector/list with one value per mesh, recycled as necessary. Default 16L.

`cex` point expansion factor; a scalar or vector/list with one value per mesh, recycled as necessary. Default `0.1`.

`add` logical; if `TRUE` the dots are added to an existing plot instead of opening a new one. Default `FALSE`.

`axes` logical; whether to draw axes on a new plot. Ignored when `add = TRUE`. Default `FALSE`.

`asp` aspect ratio of the new plot; default `1` (equal scaling). Ignored when `add = TRUE`.

`xlim, ylim` axis limits for the new plot; `NULL` (default) lets R choose automatically from all meshes' projected vertices. Ignored when `add = TRUE`.

`zoom` positive numeric magnification applied to the auto-computed axis limits when `xlim` or `ylim` is `NULL`. Values greater than `1` zoom in, values in `(0, 1)` zoom out. When `asp` is set, the zoom is plot-region aware (queried via `par("pin")`): the axis that already fills the device after `asp`-induced expansion is zoomed by exactly `zoom`, and the other axis is zoomed less so the data fills more of the plot region while preserving the requested `asp`. Ignored for any axis whose limit was supplied explicitly, and when `add = TRUE`. Default `1`.

`xlab, ylab` axis labels for the new plot. Ignored when `add = TRUE`.

`normal_weight` passed to `vcg_update_normals` when normals must be recomputed; one of `"auto"` (area if no normals present, otherwise skip), `"area"`, or `"angle"`. Default `"auto"`.

`side` which side of meshed surfaces to render. One of `"front"` (default), `"back"`, or `"both"` (renders all vertices). Point clouds are always rendered regardless of this setting.

`mesh_clipping` numeric in `[0, 1]` controlling how much of the surface is kept relative to the camera-facing direction. Vertices whose rim-light weight ($1 - |n \cdot z_{cam}|$) is at or below `1 - mesh_clipping` are dropped, leaving only the silhouette/grazing band. `mesh_clipping = 1` (no clipping) keeps all vertices; smaller values peel away more of the front/back-facing dots. Surviving vertices are drawn opaque, and their `cex` is multiplied by the rim-light weight so grazing-edge dots appear larger than near-front-facing dots. Drawing opaque points with size-modulated weight is dramatically faster than R's true per-point transparency. Default `0.7`.

`alpha` numeric in `[0, 1]`; one value per mesh (recycled), sets the transparency of each mesh (`1` = fully opaque, `0` = fully transparent). Vertices belonging to a mesh with `alpha = 0` are dropped entirely. Default `1`.

`clipping_plane` optional list of world-space clipping planes used to hide parts of the scene. Each plane is a numeric vector of length 5: the first three entries are the plane normal (n_x, n_y, n_z) (must be non-zero; normalized internally), the fourth is the signed distance from the world origin to the plane along that normal (so the plane equation is $n \cdot x = d$), and the fifth indicates which half-space is kept: `1` keeps the front side (the side the normal points to), `-1` keeps the back side, and `0` keeps whichever side currently faces the camera (auto-flipped per call based on `eye`). Multiple planes are intersected. Clipping is applied per vertex. A single length-5 numeric vector is also accepted as shorthand for a one-plane list. Default `NULL` (no clipping).

`clipping_plane_enabled`
: logical vector, one entry per mesh (recycled), controlling whether `clipping_plane` is applied to that mesh. `TRUE` (default) means the mesh participates in clipping; `FALSE` exempts the mesh entirely (all of its vertices are kept regardless of the clipping planes). Has no effect when `clipping_plane` is `NULL`.

`...`
: additional graphical parameters forwarded to `plot.default` (new plot) or `points` (when `add = TRUE`).

Value

Invisibly returns a list with components `xlim` and `ylim` (the plot limits used).

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

See Also

`vcg_update_normals`, `vcg_isosurface`

Examples

```
mesh <- vcg_isosurface(left_hippocampus_mask)

# Side view - rim-light shows the outline of the hippocampus
plot_mesh_dotcloud(
  mesh,
  eye = c(150, 0, 0),
  lookat = c(0, 0, 0),
  up = c(0, 0, 1),
  col = "steelblue",
  cex = 2
)

# Two meshes: surface + electrode point cloud
n_elec <- 20
```

```
electrodes <- structure(
  list(vb = matrix(rnorm(3 * n_elec, sd = 5), 3, n_elec) +
         rowMeans(mesh$vb)[1:3]),
  class = "mesh3d"
)
plot_mesh_dotcloud(
  mesh  = list(mesh, electrodes),
  eye    = c(150, 0, 0),
  lookat = c(0, 0, 0),
  up     = c(0, 0, 1),
  col    = list("steelblue", "red"),
  pch    = c(16L, 17L),
  cex    = c(2, 1.2)
)
```

`plot_mesh_polygon` *Render one or more meshes as flat-shaded triangles in base R*

Description

Projects each triangular face onto a 2D plane using an orthographic camera, shades it with a single color proportional to how directly it faces the camera (Lambert), depth-sorts all faces across all meshes, and draws them in a single `polygon` call.

Meshes without faces (point clouds) are substituted by a small sphere (`vcg_sphere`) centered at each point and scaled by `cex`; they then participate in the same rendering pipeline as ordinary faced meshes.

A camera-facing clipping pass discards triangles whose outward normal points along the camera ray (signed $n \cdot z_{cam} > 1 - \text{mesh\_clipping}$), peeling the front cap off the surface so the back wall (and any interior meshes) become visible. Set `mesh_clipping = 1` to disable clipping. Point-cloud meshes (those rendered as substitute `vcg_sphere` instances) are exempt from this clip so they remain solid even when the enclosing surface is peeled.

Multiple meshes share a single depth space: all faces are projected, sorted, and drawn together so the painter's algorithm works correctly across meshes.

Usage

```
plot_mesh_polygon(
  mesh,
  eye = c(0, 0, 1000),
  lookat = c(0, 0, 0),
  up = c(0, 1, 0),
  col = c("white", "gray30"),
  cex = 1,
  add = FALSE,
  axes = FALSE,
  asp = 1,
```

```
  xlim = NULL,
  ylim = NULL,
  zoom = 1,
  xlab = "",
  ylab = "",
  side = c("front", "back", "both"),
  mesh_clipping = 1,
  sphere_subdivision = 1L,
  alpha = 1,
  shadow_color = NULL,
  light_intensity = 1,
  ambient_intensity = 0.2,
  clipping_plane = NULL,
  clipping_plane_enabled = TRUE,
  ...
)
```

Arguments

mesh	a 'mesh3d' object, or a list of 'mesh3d' objects. Meshes without a face matrix ($it) are rendered as sphere instances (one vcg_sphere per vertex, radius cex).
eye	numeric vector of length 3 - camera position in world space.
lookat	numeric vector of length 3 - the world-space point the camera is looking at.
up	numeric vector of length 3 - world-space "up" direction; defaults to c(0, 1, 0).
col	base color(s) per mesh. Same forms as plot_mesh_dotcloud: a single color, a depth-gradient vector, a per-vertex character vector, or a list of any of these (one element per mesh). Default c("white", "gray30").
cex	radius of the substitute sphere used for point-cloud meshes (world units). Has no effect on meshes that already have faces. Default 1.
add	logical; if TRUE the faces are added to an existing plot instead of opening a new one. Default FALSE.
axes, asp, xlim, ylim, xlab, ylab	passed to plot.default when add = FALSE.
zoom	positive numeric magnification applied to the auto-computed axis limits when xlim or ylim is NULL. Values greater than 1 zoom in, values in (0, 1) zoom out. When asp is set, the zoom is plot-region aware (queried via par("pin")): the axis that already fills the device after asp-induced expansion is zoomed by exactly zoom, and the other axis is zoomed less so the data fills more of the plot region while preserving the requested asp. Ignored for any axis whose limit was supplied explicitly, and when add = TRUE. Default 1.
side	which side of each triangle to render. One of "front" (default), "back", or "both" (shows all triangles).
mesh_clipping	numeric in [0, 1] controlling camera-facing clipping: any triangle whose signed Lambert dot product ($n \cdot z_{cam}$, i.e. positive for front-facing) is at or above mesh_clipping is discarded. A value of 1 (default) keeps the full surface; 0.5 peels a 60-degree cap off the front and reveals the back wall (whose absolute

Lambert shade is still large, so it renders brightly); `0` clips every front-facing triangle and shows the interior only. Back-facing triangles are never clipped by this rule. Default `1`.

`sphere_subdivision`
: integer subdivision level forwarded to `vcg_sphere` when substituting point clouds. Higher values give smoother spheres at the cost of more triangles. Default `1` (80 triangles per sphere).

`alpha`
: numeric in `[0, 1]`; one value per mesh (recycled), sets the transparency of each mesh (`1` = fully opaque, `0` = fully transparent). Faces belonging to a mesh with `alpha = 0` are dropped entirely. Because faces are drawn in back-to-front (painter's) order, alpha blending across multiple meshes follows that order. Default `1`.

`shadow_color`
: color used for fully unlit (grazing/back) faces. The Lambert shade linearly interpolates from `shadow_color` (at `shade = 0`) toward the face color lit by a white light (at `shade = 1`), so unlit faces fade to this color instead of an implicit black background. Default `par("fg")`, so the shadow contrasts with the current device's canvas.

`light_intensity`
: non-negative scalar controlling the brightness of the (white) light source: at `shade = 1` the face color is multiplied by `light_intensity` (clamped to `[0, 1]`). `1` (default) reproduces the face color exactly when fully lit; smaller values darken the whole mesh, larger values saturate.

`ambient_intensity`
: scalar in `[0, 1]` acting as a lower bound on the Lambert shade so grazing/back faces still receive some light and never collapse fully to `shadow_color`. An effective shade of `max(shade, ambient_intensity)` is used in the background-to-light blend. Default `0.2`.

`clipping_plane`
: optional list of world-space clipping planes used to hide parts of the scene. Each plane is a numeric vector of length 5: the first three entries are the plane normal (n_x, n_y, n_z) (must be non-zero; normalized internally), the fourth is the signed distance from the world origin to the plane along that normal (so the plane equation is $n \cdot x = d$), and the fifth indicates which half-space is kept: `1` keeps the front side (the side the normal points to), `-1` keeps the back side, and `0` keeps whichever side currently faces the camera (auto-flipped per call based on `eye`). Multiple planes are intersected. Clipping is applied per face using the face centroid (a face is kept only when its centroid lies on the kept side of every plane). A single length-5 numeric vector is also accepted as shorthand for a one-plane list. Default `NULL` (no clipping).

`clipping_plane_enabled`
: logical vector, one entry per mesh (recycled), controlling whether `clipping_plane` is applied to that mesh. `TRUE` (default) means the mesh participates in clipping; `FALSE` exempts the mesh entirely (all of its faces are kept regardless of the clipping planes). Has no effect when `clipping_plane` is `NULL`.

`...`
: additional graphical parameters forwarded to `plot.default` (new plot only).

Details

Limitations of the base-R polygon path (no `rgl`):

- Flat shading only - one color per triangle. No per-vertex color interpolation.
- No depth buffer - faces are depth-sorted by centroid (painter's algorithm). Interpenetrating triangles can render in the wrong order.
- Anti-aliasing seams can appear between adjacent triangles on raster devices; Cairo-based devices (`png(type = "cairo")`, `svg()`, `pdf()`) produce cleaner output than the default quartz/X11 path.

Value

Invisibly returns a list with components `xlim` and `ylim` (the plot limits used).

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

See Also

`plot_mesh_dotcloud`, `vcg_isosurface`

Examples

```
mesh <- vcg_isosurface(left_hippocampus_mask)

# Surface alone
plot_mesh_polygon(
  mesh,
  eye    = c(150, 30, 0),
  lookat = c(0, 0, 0),
  up     = c(0, 0, 1),
  col    = "steelblue"
)

# Surface + electrode point cloud (rendered as small icospheres)
n_elec <- 20
electrodes <- structure(
  list(vb = matrix(rnorm(3 * n_elec, sd = 5), 3, n_elec) +
```

```
        rowMeans(mesh$vb)[1:3]),
  class = "mesh3d"
)
plot_mesh_polygon(
  mesh = list(mesh, electrodes),
  eye    = c(150, -30, 0),
  lookat = c(0, 0, 0),
  up     = c(0, 0, 1),
  alpha  = c(0.5, 1),
  col    = list("steelblue", "red"),
  cex    = 1.5
)
```

`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
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)
```

print.ravetools_curve *Print method for* ravetools_curve

Description

Prints a concise summary of a ravetools_curve object returned by catmull_rom_3d.

Usage

```
## S3 method for class 'ravetools_curve'
print(x, ...)
```

Arguments

x	an object of class ravetools_curve.
...	additional arguments passed to print.data.frame.

Value

Invisibly returns x.

project_plane *Project plane to a surface*

Description

Project a two-dimensional plane (such as 'ECoG' grid) to a three-dimensional surface while preserving the order

Usage

```
project_plane(
  target,
  width,
  height,
  shape,
  initial_positions,
  translate_first = TRUE,
  diagnostic = FALSE,
  n_iters = 5
)
```

Arguments

target	target surface to be projected to, must be object that can be converted to 'mesh3d' ('rgl' surface), for example, 'fs.surface' (from 'freesurferformat' package) or 'ieegio_surface' from **ieegio** package; see ensure_mesh3d for the coercion rules.
width, height	width and height of the plane in world space (for 'ECoG' grid, the unit is millimeter)
shape	vector of two integers: the first element is the number of vertices (or electrode contacts) along 'width' direction; the second element is the number of vertices along 'height' direction. The total number of vertices of the plane will be prod(shape). Notice
initial_positions	a shape[[1]] x shape[[2]] x 3 array or a n x 3 matrix, where n is prod(shape), the number of vertices indicating the initial vertex positions of the plane
translate_first	whether to translate the plane first if the plane center is far from the surface; default is FALSE; set to TRUE for a warm start
diagnostic	whether to plot diagnostic figures showing the morphing progress.
n_iters	number of iterations; default is five

Value

The projected vertex locations, same order as initial_positions.

Coercing Surface Inputs

The surface objects are converted to 'mesh3d' object before applying further calculations.

When surface is a surface **ieegio** object, the returned mesh3d$vb contains vertices that have been left-multiplied by surface$geometry$transforms[[1]] (the first transform stored in the geometry, typically the ScannerAnat or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw surface$geometry$vertices without applying any transform, so downstream code often multiplied by surface$geometry$transforms[[1]] (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from surface$geometry$transforms by hand after calling a **ravetools** mesh function on an 'ieegio_surface', remove that manual step.

Surfaces with an empty or missing geometry$transforms list (for example, surfaces produced by **ieegio**'s volume_to_surface, which stores an identity transform) are unaffected.

If geometry$transforms contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
# Construct target surface
```

```
sphere <- vcg_sphere()
target <- structure(
  class = "mesh3d",
  list(
    vb = cbind(
      sphere$vb[1:3, ] - c(0.8, 0, 0),
      sphere$vb[1:3, ] + c(0.8, 0, 0)
    ),
    it = cbind(
      sphere$it[1:3, ],
      sphere$it[1:3, ] + ncol(sphere$vb)
    )
  )
)
n_surfverts <- ncol(target$vb)

plane <- plane_geometry(width = 3, height = 3, shape = c(30, 30))
plane$vb <- plane$vb[1:3, , drop = FALSE] + c(0, 0, 2)
n_contacts <- ncol(plane$vb)

# First plot
x <- t(cbind(target$vb, plane$vb))
colnames(x) <- c('x', 'y', 'z')
graphics::pairs(
  x = x, asp = 1,
  col = c(
    rep("black", n_surfverts),
    rep("green", n_contacts)
  ),
  pch = c(
    rep(46, n_surfverts),
    rep(20, n_contacts)
  )
)

projected <- project_plane(
  target = target, width = 3, height = 3, shape = c(30, 30),
  initial_positions = t(plane$vb),
  translate_first = TRUE, diagnostic = FALSE
)

y <- rbind(x, projected)
graphics::pairs(
  x = y, asp = 1,
  col = c(
    rep("black", ncol(target$vb)),
    rep("green", n_contacts),
    rep("red", n_contacts)
  ),
  pch = c(
    rep(46, n_surfverts),
    rep(1, n_contacts),
    rep(20, n_contacts)
```

```
  )
)
```

---

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

```
pwelch(
  x,
  fs,
  window = 64,
  noverlap = window/2,
  nfft = "auto",
  window_family = hamming,
  col = "black",
  xlim = NULL,
  ylim = NULL,
  main = "Welch periodogram",
  plot = 0,
  log = c("xy", "", "x", "y"),
  ...
)

## S3 method for class '`ravetools-pwelch`'
print(x, ...)

## S3 method for class '`ravetools-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,
  window = 64,
  noverlap = window/2,
  nfft = "auto",
  window_family = hamming
)
```

Arguments

x	numerical vector or a row-major vector, signals. If x is a matrix, then each row is a channel. For plot function, x is the 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 half of the window
nfft	number of points in window function; default is automatically determined from input data and window, scaled up to the nearest power of 2
window_family	function generator for generating filter windows, default is hamming. This can be any window function listed in the filter window family, or any window generator function from package gsignal. Default is hamming. For 'iEEG' users, both hamming and blackmanharris are offered by 'EEG-lab'; while blackmanharris offers better attenuation than Hamming windows, it also has lower spectral resolution. hamming has a 42.5 dB side-lobe attenuation. This may mask spectral content below this value (relative to the peak spectral content). Choosing different windows enables you to make trade-off between resolution (e.g., using a rectangular window) and side-lobe attenuation (e.g., using a hanning window)

`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

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

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

raw-to-sexp *Convert raw vectors to R vectors*

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 bites, 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

```
# 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 concaternates the followings
# 10000000 (0x80), 00000000 (0x00), 01111111 (0x7f), 10000000 (0x80),
# 11111111 (0xFF), 01010000 (0x50), 01111111 (0x7f), 00000000 (0x00)

if(.Platform$endian == "little") {
  bitstring <- paste0(
    "00000000011111111010100001111111",
    "10000000011111110000000010000000"
  )
} else {
  bitstring <- paste0(
    "00000001000000000111111100000001",
    "11111111000010101111111000000000"
  )
}

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

# This is actual value
raw_to_int64(x)
```

`rcond_filter_ar` *Computer reciprocal condition number of an 'Arma' filter*

Description

Test whether the filter is numerically stable for `filtfilt`.

Usage

```
rcond_filter_ar(a)
```

Arguments

`a`	auto-regression coefficient, numerical vector; the first element must not be zero

Value

Reciprocal condition number of matrix `z1`, used in `filtfilt`. If the number is less than `.Machine$double.eps`, then `filtfilt` will fail.

See Also

check_filter

Examples

```
# Butterworth filter with low-pass at 0.1 Hz (order = 4)
filter <- butter(4, 0.1, "low")

# TRUE
rcond_filter_ar(filter$a) > .Machine$double.eps

diagnose_filter(filter$b, filter$a, 500)

# Bad filter (order is too high)
filter <- butter(50, 0.1, "low")

rcond_filter_ar(filter$a) > .Machine$double.eps

# filtfilt needs to inverse a singular matrix
diagnose_filter(filter$b, filter$a, 500)
```

---

register_volume *Imaging registration using* 'NiftyReg'

Description

Registers 'CT' to 'MRI', or 'MRI' to another 'MRI'

Usage

```
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

```
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

oldpar <- 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")

par(oldpar)
```

---

resample_3d_volume *Sample '3D' volume in the world (anatomical 'RAS') space*

Description

Low-level implementation to sample a '3D' volume into given orientation and shape via a nearest-neighbor sampler.

Usage

```
resample_3d_volume(
  x,
  new_dim,
  vox2ras_old,
  vox2ras_new = vox2ras_old,
  na_fill = NA
)
```

Arguments

x	image (volume) to be sampled: dim(x) must have length of 3
new_dim	target dimension, integers of length 3
vox2ras_old	from volume index (column-row-slice) to 'RAS' (right-anterior-superior) transform: the volume index starts from 0 (C-style) instead of 1 (R-style) to comply with 'NIfTI' transform.
vox2ras_new	the targeting transform from volume index to 'RAS'
na_fill	default numbers to fill if a pixel is out of bound; default is NA or as.raw(0) if input x is raw type

Value

A newly sampled volume that aligns with x in the anatomical 'RAS' coordinate system. The underlying storage mode is the same as x

Examples

```
# up-sample and rotate image
x <- array(0, c(9, 9, 9))
x[4:6, 4:6, 4:6] <- 1
vox2ras <- matrix(nrow = 4, byrow = TRUE, c(
  0.7071, -0.7071, 0, 0,
  0.7071, 0.7071, 0, -5.5,
  0, 0, 1, -4,
  0, 0, 0, 1
))

new_vox2ras <- matrix(nrow = 4, byrow = TRUE, c(
  0, 0.5, 0, -4,
  0, 0, -0.5, 4,
  0.5, 0, 0, -4,
  0, 0, 0, 1
))

y <- resample_3d_volume(
  x,
  c(17, 17, 17),
  vox2ras_old = vox2ras,
```

```
  vox2ras_new = new_vox2ras,
  na_fill = 0
)


image(y[9,,])
```

---

rgl-call *Safe ways to call package 'rgl' without requiring 'x11'*

Description

Internally used for example show-cases. Please install package 'rgl' manually to use these functions.

Usage

```
rgl_call(FUN, ...)

rgl_view(expr, quoted = FALSE, env = parent.frame())

rgl_plot_normals(x, length = 1, lwd = 1, col = 1, ...)
```

Arguments

FUN	'rgl' function name
...	passed to 'rgl' function
expr	expression within which 'rgl' functions are called
quoted	whether expr is quoted
env	environment in which expr is evaluated
x	triangular 'mesh3d' object
length, lwd, col	normal vector length, size, and color

Examples

```
# Make sure the example does not run when compiling
# or check the package
if(FALSE) {

  volume <- array(0, dim = c(8,8,8))
  volume[4:5, 4:5, 4:5] <- 1
  mesh <- mesh_from_volume(volume, verbose = FALSE)
```

```
  rgl_view({

    rgl_call("shade3d", mesh, col = 3)
    rgl_plot_normals(mesh)

  })

}
```

`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

```
# 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]

# Check if z ang 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)

oldpar <- 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]))
}
par(oldpar)
```

---

stimpulse_interpolate *Find and interpolate stimulation pulses*

Description

Find and interpolate stimulation pulses

Usage

```
stimpulse_find(
  signal,
  sample_rate,
  pulse_duration,
  n_pulses = NA,
  threshold = NA
```

```
)

stimpulse_extract(
  signal,
  pulse_info,
  expand_timepoints = c(-10, 20),
  center = TRUE
)

stimpulse_align(signal, pulse_info, expand_timepoints = c(-10, 20))

stimpulse_interpolate(
  signal,
  sample_rate,
  pulse_info,
  max_offset = c(-2e-04, 5e-04)
)
```

Arguments

`signal`	a channel signal trace
`sample_rate`	sample rate
`pulse_duration`	stimulation pulse duration in seconds
`n_pulses`	suggested number of pulses
`threshold`	suggested suggested threshold of responses to find stimulation pulses
`pulse_info`	a list containing number of pulses `n_pulse`, onset index (`onset_index`, first time-point is 1), offset index (`offset_index`); this can be generated by `stimpulse_find`; see 'Examples' below
`expand_timepoints`	point offsets allowed to align the pulses
`center`	whether to center the pulses by median; default is true
`max_offset`	maximum (edge) offsets in seconds to interpolate the pulses; default is `-0.0002` seconds before stimulation onset and `0.0005` seconds after the stimulation offset

Value

`stimpulse_find` and `stimpulse_align` returns the pulse information (`pulse_info`) with the time-points of detected or corrected stimulation on-set and off-set. The time-points are 1-indexed. `stimpulse_extract` extract the signals around pulses; `stimpulse_interpolate` returns interpolated signals.

Examples

```
data("stimulation_signal")
```

```
signal <- stimulation_signal$signal
sample_rate <- stimulation_signal$sample_rate

# each pulse is roughly <0.001 seconds
pulse_durations <- 0.001

# Initial pulses
pulse_info <- stimpulse_find(signal, sample_rate, pulse_durations)

# number of pulses detected
pulse_info$n_pulses

# extract responses -10 points before onset ~ 20 points after offset
expand_timepoints <- c(-20, 80)
pulses_snippets <- stimpulse_extract(
  signal = signal,
  pulse_info = pulse_info,
  expand_timepoints = expand_timepoints
)

# Visualize the pulses
snippet_time <- seq(
  expand_timepoints[[1]], by = 1,
  length.out = nrow(pulses_snippets)) / sample_rate * 1000
matplot(snippet_time, pulses_snippets, type = 'l', lty = 1, col = 'gray80',
        xlab = "Time (ms)", ylab = "uV", main = "Initial find")

# Align the pulses
pulse_info <- stimpulse_align(signal, pulse_info)

# Estimated pulse duration
estimated_duration <-
  (pulse_info$offset_index - pulse_info$onset_index + 1) / sample_rate

# reload aligned pulses
pulses_snippets <- stimpulse_extract(
  signal = signal,
  pulse_info = pulse_info,
  expand_timepoints = expand_timepoints
)
matplot(snippet_time, pulses_snippets, type = 'l', lty = 1, col = 'gray80',
        xlab = "Time (ms)", ylab = "uV", main = "Aligned pulses")
lines(snippet_time, rowMeans(pulses_snippets), col = 'red')


# Interpolate the pulses
interpolated <- stimpulse_interpolate(
  signal = signal,
  sample_rate = sample_rate,
  pulse_info = pulse_info,
  max_offset = c(-0.0003, 0.0005)
)
```

```
interp_snippets <- stimpulse_extract(
  signal = interpolated,
  pulse_info = pulse_info,
  expand_timepoints = expand_timepoints
)

oldpar <- par(mfrow = c(1, 2))
on.exit(par(oldpar))
matplot(snippet_time, pulses_snippets, type = 'l', lty = 1,
        col = 'gray80', xlab = "Time (ms)", ylab = "uV",
        main = "Stim pulses", ylim = c(-600, 400))
lines(snippet_time, rowMeans(pulses_snippets), col = 'red')
abline(v = max(estimated_duration) * 1000, lty = 2)

matplot(snippet_time, interp_snippets, type = 'l', lty = 1,
        col = 'gray80', xlab = "Time (ms)", ylab = "uV",
        main = "Interpolated 0.5 ms bandwidth")
lines(snippet_time, rowMeans(interp_snippets), col = 'red')
abline(v = max(estimated_duration) * 1000, lty = 2, col = "gray40")
abline(v = max(estimated_duration) * 1000 + 0.5, lty = 2)
```

`stimulation_signal` *Sample stimulation recording*

Description

Sample stimulation recording

Usage

```
stimulation_signal
```

Format

A list of one-second signal trace and sample rate (30000)

vcg_isosurface *Create surface mesh from 3D-array*

Description

Create surface from 3D-array using marching cubes algorithm

Usage

```
vcg_isosurface(
  volume,
  threshold_lb = 0,
  threshold_ub = NA,
  vox_to_ras = diag(c(-1, -1, 1, 1))
)
```

Arguments

volume	a volume or a mask volume
threshold_lb	lower-bound threshold for creating the surface; default is 0
threshold_ub	upper-bound threshold for creating the surface; default is NA (no upper-bound)
vox_to_ras	a 4x4 'affine' transform matrix indicating the 'voxel'-to-world transform.

Value

A triangular mesh of class 'mesh3d'

Examples

```
if(is_not_cran()) {

library(ravetools)
data("left_hippocampus_mask")

mesh <- vcg_isosurface(left_hippocampus_mask)


rgl_view({

  rgl_call("mfrow3d", 1, 2)

  rgl_call("title3d", "Direct ISOSurface")
  rgl_call("shade3d", mesh, col = 2)

  rgl_call("next3d")
  rgl_call("title3d", "ISOSurface + Implicit Smooth")
```

```
  rgl_call("shade3d",
           vcg_smooth_implicit(mesh, degree = 2),
           col = 3)
})

}
```

---

vcg_kdtree_nearest *Find nearest* k *points*

Description

For each point in the query, find the nearest k points in target using K-D tree.

Usage

```
vcg_kdtree_nearest(target, query, k = 1, leaf_size = 16, max_depth = 64)
```

Arguments

target	a matrix with n rows (number of points) and 2 or 3 columns, or a mesh3d object. This is the target point cloud where nearest distances will be sought
query	a matrix with n rows (number of points) and 2 or 3 columns, or a mesh3d object. This is the query point cloud where for each point, the nearest k points in target will be sought.
k	positive number of nearest neighbors to look for
leaf_size	the suggested leaf size for the K-D tree; default is 16; larger leaf size will result in smaller depth
max_depth	maximum depth of the K-D tree; default is 64

Value

A list of two matrices: index is a matrix of indices of target points, whose distances are close to the corresponding query point. If no point in target is found, then NA will be presented. Each distance is the corresponding distance from the query point to the target point.

Coercing Surface Inputs

The surface objects are converted to 'mesh3d' object before applying further calculations.

When surface is a surface **ieegio** object, the returned mesh3d$vb contains vertices that have been left-multiplied by surface$geometry$transforms[[1]] (the first transform stored in the geometry, typically the ScannerAnat or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw surface$geometry$vertices without applying any transform, so downstream code often multiplied by surface$geometry$transforms[[1]] (or an equivalent) manually before working in world space. Such code will now *double* apply the

transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
# Find nearest point in B with the smallest distance for each point in A

library(ravetools)

n <- 10
A <- matrix(rnorm(n * 2), nrow = n)
B <- matrix(rnorm(n * 4), nrow = n * 2)
result <- vcg_kdtree_nearest(
  target = B, query = A,
   k = 1
)

plot(
  rbind(A, B),
  pch = 20,
  col = c(rep("red", n), rep("black", n * 2)),
  xlab = "x",
  ylab = "y",
  main = "Black: target; Red: query"
)

nearest_points <- B[result$index, ]
arrows(A[, 1],
       A[, 2],
       nearest_points[, 1],
       nearest_points[, 2],
       col = "red",
       length = 0.1)

# ---- Sanity check ------------------------------------------------
nearest_index <- apply(A, 1, function(pt) {
  which.min(colSums((t(B) - pt) ^ 2))
})

result$index == nearest_index
```

`vcg_mesh_volume` *Compute volume for manifold meshes*

Description

Compute volume for manifold meshes

Usage

```
vcg_mesh_volume(mesh)
```

Arguments

`mesh`	triangular mesh of class `'mesh3d'`

Value

The numeric volume of the mesh

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
# Initial mesh
mesh <- vcg_sphere()

vcg_mesh_volume(mesh)
```

`vcg_raycaster` *Cast rays to intersect with mesh*

Description

Cast rays to intersect with mesh

Usage

```
vcg_raycaster(
  x,
  ray_origin,
  ray_direction,
  max_distance = Inf,
  both_sides = FALSE
)
```

Arguments

`x`	surface mesh
`ray_origin`	a matrix with 3 rows or a vector of length 3, the positions of ray origin
`ray_direction`	a matrix with 3 rows or a vector of length 3, the direction of the ray, will be normalized to length 1
`max_distance`	positive maximum distance to cast the normalized ray; default is infinity. Any invalid distances (negative, zero, or `NA`) will be interpreted as unset.
`both_sides`	whether to inverse the ray (search both positive and negative ray directions); default is false

Value

A list of ray casting results: whether any intersection is found, position and face normal of the intersection, distance of the ray, and the index of the intersecting face (counted from 1)

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
library(ravetools)
sphere <- vcg_sphere(normals = FALSE)
sphere$vb[1:3, ] <- sphere$vb[1:3, ] + c(10, 10, 10)
vcg_raycaster(
  x = sphere,
  ray_origin = array(c(0, 0, 0, 1, 0, 0), c(3, 2)),
  ray_direction = c(1, 1, 1)
)
```

---

vcg_smooth *Implicitly smooth a triangular mesh*

Description

Applies smoothing algorithms on a triangular mesh.

Usage

```
vcg_smooth_implicit(
  mesh,
  lambda = 0.2,
  use_mass_matrix = TRUE,
  fix_border = FALSE,
  use_cot_weight = FALSE,
  degree = 1L,
  laplacian_weight = 1
)

vcg_smooth_explicit(
  mesh,
  type = c("taubin", "laplace", "HClaplace", "fujiLaplace", "angWeight",
    "surfPreserveLaplace"),
  iteration = 10,
  lambda = 0.5,
  mu = -0.53,
  delta = 0.1
)
```

Arguments

mesh	triangular mesh stored as object of class 'mesh3d'.
lambda	In vcg_smooth_implicit, the amount of smoothness, useful only if use_mass_matrix is TRUE; default is 0.2. In vcg_smooth_explicit, parameter for 'taubin' smoothing.
use_mass_matrix	logical: whether to use mass matrix to keep the mesh close to its original position (weighted per area distributed on vertices); default is TRUE
fix_border	logical: whether to fix the border vertices of the mesh; default is FALSE
use_cot_weight	logical: whether to use cotangent weight; default is FALSE (using uniform 'Laplacian')
degree	integer: degrees of 'Laplacian'; default is 1
laplacian_weight	numeric: weight when use_cot_weight is FALSE; default is 1.0
type	method name of explicit smooth, choices are 'taubin', 'laplace', 'HClaplace', 'fujiLaplace', 'angWeight', 'surfPreserveLaplace'.
iteration	number of iterations
mu	parameter for 'taubin' explicit smoothing.
delta	parameter for scale-dependent 'Laplacian' smoothing or maximum allowed angle (in 'Radian') for deviation between surface preserving 'Laplacian'.

Value

An object of class "mesh3d" with:

vb	vertex coordinates
normals	vertex normal vectors
it	triangular face index

Coercing Surface Inputs

The surface objects are converted to 'mesh3d' object before applying further calculations.

When surface is a surface **ieegio** object, the returned mesh3d$vb contains vertices that have been left-multiplied by surface$geometry$transforms[[1]] (the first transform stored in the geometry, typically the ScannerAnat or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw surface$geometry$vertices without applying any transform, so downstream code often multiplied by surface$geometry$transforms[[1]] (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from surface$geometry$transforms by hand after calling a **ravetools** mesh function on an 'ieegio_surface', remove that manual step.

Surfaces with an empty or missing geometry$transforms list (for example, surfaces produced by **ieegio**'s volume_to_surface, which stores an identity transform) are unaffected.

If geometry$transforms contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
if(is_not_cran()) {

# Prepare mesh with no normals
data("left_hippocampus_mask")

# Grow 2mm on each direction to fill holes
volume <- grow_volume(left_hippocampus_mask, 2)

# Initial mesh
mesh <- vcg_isosurface(volume)

# Start: examples
rgl_view({
  rgl_call("mfrow3d", 2, 4)
  rgl_call("title3d", "Naive ISOSurface")
  rgl_call("shade3d", mesh, col = 2)

  rgl_call("next3d")
  rgl_call("title3d", "Implicit Smooth")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_implicit(mesh, degree = 2))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - taubin")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "taubin"))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - laplace")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "laplace"))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - angWeight")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "angWeight"))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - HClaplace")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "HClaplace"))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - fujiLaplace")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "fujiLaplace"))

  rgl_call("next3d")
  rgl_call("title3d", "Explicit Smooth - surfPreserveLaplace")
  rgl_call("shade3d", col = 2,
           x = vcg_smooth_explicit(mesh, "surfPreserveLaplace"))
```

```
})

}
```

`vcg_sphere` *Simple 3-dimensional sphere mesh*

Description

Simple 3-dimensional sphere mesh

Usage

```
vcg_sphere(sub_division = 3L, normals = TRUE)
```

Arguments

`sub_division`	density of vertex in the resulting mesh
`normals`	whether the normal vectors should be calculated

Value

A `'mesh3d'` object

Examples

```
vcg_sphere()
```

`vcg_subdivision` *Sub-divide (up-sample) a triangular mesh*

Description

Up-sample a triangular mesh by adding a vertex at each edge or face center.

Usage

```
vcg_subdivision(mesh, method = c("edge", "barycenter"))
```

Arguments

`mesh`	triangular mesh stored as object of class 'mesh3d'.
`method`	either `'edge'` (default) to add new mid-point vertices to edge, or `'barycenter'` to add new vertices at face `'Bary'` centers.

Value

An object of class "mesh3d"

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
mesh <- plane_geometry()

# default
mesh_edge <- vcg_subdivision(mesh, "edge")

# barycenter
mesh_face <- vcg_subdivision(mesh, "barycenter")

if(is_not_cran()) {

  rgl_view({
    rgl_call("wire3d", mesh, col = 1)
    rgl_call("wire3d", mesh_edge, col = 2)
    rgl_call("wire3d", mesh_face, col = 3)
  })


}
```

`vcg_subset_vertex` *Subset mesh by vertex*

Description

Subset mesh by vertex

Usage

```
vcg_subset_vertex(x, selector)
```

Arguments

`x`	surface mesh
`selector`	logical vector (must not contain NA), and length must be consistent with the number of vertices in `x`: which nodes are to be kept

Value

A triangular mesh of class `'mesh3d'`, a subset of `x`

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
sphere <- vcg_sphere()

nv <- ncol(sphere$vb)

selector <- seq_len(nv) > (nv / 2)
```

```
sub <- vcg_subset_vertex(sphere, selector)

if(is_not_cran()) {
  rgl_view({

    # subset sphere will be displayed in red
    rgl_call("shade3d", sub, col = 'red')

    # Original sphere will be displayed as wireframe
    rgl_call("wire3d", sphere, col = (2 - selector))

  })
}
```

---

vcg_uniform_remesh *Sample a surface mesh uniformly*

Description

Sample a surface mesh uniformly

Usage

```
vcg_uniform_remesh(
  x,
  voxel_size = NULL,
  offset = 0,
  discretize = FALSE,
  multi_sample = FALSE,
  absolute_distance = FALSE,
  merge_clost = FALSE,
  verbose = TRUE
)
```

Arguments

x	surface
voxel_size	'voxel' size for space 'discretization'
offset	offset position shift of the new surface from the input
discretize	whether to use step function(TRUE) instead of linear interpolation (FALSE) to calculate the position of the intersected edge of the marching cube; default is FALSE
multi_sample	whether to calculate multiple samples for more accurate results (at the expense of more computing time) to remove artifacts; default is FALSE

absolute_distance
: whether an unsigned distance field should be computed. When set to TRUE, non-zero offsets is to be set, and double-surfaces will be built around the original surface, like a sandwich.

merge_clost
: whether to merge close vertices; default is TRUE

verbose
: whether to verbose the progress; default is TRUE

Value

A triangular mesh of class 'mesh3d'

Coercing Surface Inputs

The surface objects are converted to 'mesh3d' object before applying further calculations.

When surface is a surface **ieegio** object, the returned mesh3d$vb contains vertices that have been left-multiplied by surface$geometry$transforms[[1]] (the first transform stored in the geometry, typically the ScannerAnat or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw surface$geometry$vertices without applying any transform, so downstream code often multiplied by surface$geometry$transforms[[1]] (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from surface$geometry$transforms by hand after calling a **ravetools** mesh function on an 'ieegio_surface', remove that manual step.

Surfaces with an empty or missing geometry$transforms list (for example, surfaces produced by **ieegio**'s volume_to_surface, which stores an identity transform) are unaffected.

If geometry$transforms contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
sphere <- vcg_sphere()
mesh <- vcg_uniform_remesh(sphere, voxel_size = 0.45)

if(is_not_cran()) {

rgl_view({

  rgl_call("mfrow3d", 1, 2)

  rgl_call("title3d", "Input")
  rgl_call("wire3d", sphere, col = 2)
  rgl_call("next3d")

  rgl_call("title3d", "Re-meshed to 0.1mm edge distance")
  rgl_call("wire3d", mesh, col = 3)
})

}
```

`vcg_update_normals` *Update vertex normal*

Description

Update vertex normal

Usage

```
vcg_update_normals(
  mesh,
  weight = c("area", "angle"),
  pointcloud = c(10, 0),
  verbose = FALSE
)
```

Arguments

`mesh`	triangular mesh or a point-cloud (matrix of 3 columns)
`weight`	method to compute per-vertex normal vectors: `"area"` weighted average of surrounding face normal, or `"angle"` weighted vertex normal vectors.
`pointcloud`	integer vector of length 2: containing optional parameters for normal calculation of point clouds; the first entry specifies the number of neighboring points to consider; the second entry specifies the amount of smoothing iterations to be performed.
`verbose`	whether to verbose the progress

Value

A `'mesh3d'` object with normal vectors.

Coercing Surface Inputs

The surface objects are converted to `'mesh3d'` object before applying further calculations.

When `surface` is a surface **ieegio** object, the returned `mesh3d$vb` contains vertices that have been left-multiplied by `surface$geometry$transforms[[1]]` (the first transform stored in the geometry, typically the `ScannerAnat` or voxel-to-world transform).

Breaking change: Earlier versions (before 0.2.6) of **ravetools** returned the raw `surface$geometry$vertices` without applying any transform, so downstream code often multiplied by `surface$geometry$transforms[[1]]` (or an equivalent) manually before working in world space. Such code will now *double* apply the transform and produce incorrect coordinates. If you previously applied a transform from `surface$geometry$transforms` by hand after calling a **ravetools** mesh function on an `'ieegio_surface'`, remove that manual step.

Surfaces with an empty or missing `geometry$transforms` list (for example, surfaces produced by **ieegio**'s `volume_to_surface`, which stores an identity transform) are unaffected.

If `geometry$transforms` contains multiple transforms targeting different coordinate spaces, only the first one is used. Callers that need a specific target space should select and apply that transform themselves before calling **ravetools** mesh functions.

Examples

```
if(is_not_cran()) {

# Prepare mesh with no normal
data("left_hippocampus_mask")
mesh <- vcg_isosurface(left_hippocampus_mask)
mesh$normals <- NULL

# Start: examples
new_mesh <- vcg_update_normals(mesh, weight = "angle",
                               pointcloud = c(10, 10))

rgl_view({
  rgl_call("mfrow3d", 1, 2)
  rgl_call("shade3d", mesh, col = 2)

  rgl_call("next3d")
  rgl_call("shade3d", new_mesh, col = 2)
})
}
```

---

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,
  segment_length = 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
segment_length	optional positive integer; when provided, long signals are processed in overlapping segments of this length (in samples) using batched mvfftw_c2c convolutions instead of a single full-length FFT. This dramatically reduces peak memory and FFT cost for long recordings (e.g. multi-hour). Must be strictly greater than the longest wavelet kernel length (the kernel at the lowest frequency); otherwise an error is raised. Default is NULL, which uses the legacy single-shot path. When segment_length >= length(data) the function silently falls back to the legacy path. Results match the legacy path on the interior of the signal up to floating-point error; the first/last ceiling(max_kernel_len/2) samples may differ near the global boundaries (both implementations have boundary artifacts there).
...	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

```
# 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
coef2 <- morlet_wavelet(x, freq, 100, c(2,3), precision = "double")
power2 <- (coef2$real[])^2 + (coef2$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