

Package ‘ravetools’

September 4, 2026

Type Package

Title Signal and Image Processing Toolbox for Analyzing Intracranial Electroencephalography Data

Version 0.3.2

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](https://doi.org/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, knitr, rmarkdown

VignetteBuilder knitr

LazyData true

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Zhengjia Wang [aut, cre] (ORCID: <https://orcid.org/0000-0001-5629-1116>),
 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 2026-09-04 07:00:02 UTC

Contents

apply_transform3d	4
band_pass	5
baseline_array	7
bpc	10
butter_max_order	14
carla	15
catmull_rom_3d	19
check_filter	21
collapse	22
color_ramp_continuous	24
convolve	25
crp	27
crp_cluster	32

decimate	35
design_filter	36
design_filter_fir	38
design_filter_iir	41
detrend	43
diagnose_channel	44
diagnose_filter	46
dijkstras-path	48
ensure_mesh3d	51
fast_cov	53
fast_quantile	54
fftw_r2c	55
fill_surface	58
filter-window	60
filter_signal	61
filtfilt	62
find_peaks	63
fir1	64
firls	65
freqz2	66
gammatone_fast	66
grow_volume	68
internal_rave_function	69
left_hippocampus_mask	70
matlab_palette	70
mesh_from_volume	70
mriscurvature	72
mriscinflate	74
mrismake_surfaces	76
mrismesh	79
mrissmooth	81
mrissphere	82
multitaper	84
naive_nmf	86
new_matrix4	87
new_quaternion	88
new_vector3	88
notch_filter	89
parallel-options	90
plane_geometry	91
plot.ravetools_bpc	92
plot.ravetools_crp	92
plot.ravetools_crp_cluster	93
plot.ravetools_curve	93
plot_mesh_dotcloud	94
plot_mesh_polygon	98
plot_signals	102
print.ravetools_curve	103

project_plane	104
pwelch	106
raw-to-sexp	109
rcond_filter_ar	112
register_volume	113
register_volume3d	114
resample_3d_volume	118
rgl-call	119
save_registration	120
shift_array	121
stimpulse_interpolate	123
stimulation_signal	126
vcg_average_edge_length	126
vcg_count_edge_defects	127
vcg_detect_collision	127
vcg_fix_defects	132
vcg_isosurface	134
vcg_kdtree_nearest	135
vcg_max_edge_length	137
vcg_mesh_patch	138
vcg_mesh_volume	139
vcg_raycaster	140
vcg_smooth	141
vcg_sphere	144
vcg_subdivide_max_edge_length	145
vcg_subdivision	146
vcg_subset_vertex	147
vcg_uniform_remesh	149
vcg_update_normals	150
wavelet	152
write_ants_transform	154
write_ants_warp	155

Index	157
--------------	------------

apply_transform3d	<i>Apply a linear RAS transform to resample a 3D volume</i>
-------------------	---

Description

Warps a moving volume onto a reference grid given a 4x4 RAS-to-RAS transform (such as the transform returned by [register_volume3d](#)). The transform maps reference (fixed) RAS coordinates to moving RAS coordinates.

Usage

```

apply_transform3d(
    volume,
    vox2ras,
    transform,
    reference_dim = dim(volume),
    reference_vox2ras = vox2ras,
    interpolation = c("trilinear", "nearest", "bspline"),
    na_fill = 0
)

```

Arguments

volume	moving 3D array to resample
vox2ras	the moving volume's voxel-to-RAS 4x4 transform
transform	4x4 RAS-to-RAS transform (fixed to moving)
reference_dim	output dimension (the fixed grid); defaults to dim(volume)
reference_vox2ras	the fixed grid's voxel-to-RAS transform; defaults to vox2ras
interpolation	'trilinear' (default), 'nearest', or 'bspline' (cubic Catmull-Rom)
na_fill	value for out-of-bounds voxels; default 0

Value

The resampled volume on the reference grid, with a 'vox2ras' attribute equal to reference_vox2ras.

See Also

[register_volume3d](#)

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 of length order+1.

Value

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

Examples

```

t <- seq(0, 1, by = 0.0005)
x <- sin(t * 0.4 * pi) + sin(t * 4 * pi) + 2 * sin(t * 120 * pi)

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)

```

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

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

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 *frequency*time*, i.e. signals that share the same session and location also share the same baseline. In this case, we assign `unit_dims=c(1,4)`.

There are 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"

$$10 \times (\log_{10}(z) - \log_{10}(\bar{z}_0))$$

"zscore"

$$\frac{z - \bar{z}_0}{sd(z_0)}$$

"sqrt_zscore"

$$\frac{\sqrt{z} - \sqrt{\bar{z}_0}}{sd(\sqrt{z_0})}$$

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

$$\frac{10 \log_{10}(z) - 10 \log_{10}(\bar{z}_0)}{sd(10 \log_{10}(z_0))}$$

"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)

```

Description

Identifies a small set of canonical temporal response shapes - *basis profile curves* (BPCs) - from the single-trial stimulation-evoked responses recorded at one measurement electrode, where the trials are grouped by stimulation site (or any other condition). Each stimulation group is assigned to the BPC that best explains its trials, and the projection strength of every group is quantified. This is the across-stimulation-site BPC method (see ‘References’); the related [crp_cluster](#) applies the same idea across electrodes.

Usage

```
bpc(
  x,
  groups,
  time = NULL,
  time_window = NULL,
  n_bpc = NULL,
  initial_rank = NULL,
  zeta_threshold = 1,
  null_class = TRUE,
  nmf_max_iters = 10000,
  nmf_tol = c(1e-04, 1e-08),
  verbose = TRUE
)
```

Arguments

x	numeric matrix of single-trial evoked voltages with shape time x trials (rows are timepoints, columns are trials). All trials across all stimulation groups are stacked column-wise.
groups	vector of length ncol(x) (integer, character or factor) giving the stimulation subgroup of each column of x. At least two subgroups are required and each subgroup needs at least two trials.
time	optional numeric vector of length nrow(x) giving the stimulus-aligned time (in seconds) of each row of x. Only used to crop to time_window and as the plotting axis; the method itself does not require it.
time_window	optional numeric c(lo, hi) analysis window (seconds); requires time. An NA extends to the corresponding data edge. NULL (the default) uses all rows.
n_bpc	integer or NULL; if given, fixes the number of basis curves and skips automatic rank selection.
initial_rank	integer or NULL; the starting NMF rank for automatic selection. NULL uses $\min(n - 1L, 10L)$ where n is the number of subgroups.
zeta_threshold	numeric > 0; rank-selection cutoff on ζ (see ‘Details’); smaller values give fewer curves. Defaults to 1, matching the BPC reference.
null_class	logical; if TRUE (the default) subgroups whose winner-take-all loading falls below $1/(2\sqrt{n})$ are left unassigned (NA); if FALSE every subgroup is forced into a curve.

nmf The `naive_nmf` result for the selected rank, with H (raw loadings) and H0 (winner-take-all, thresholded).

n_bpc Integer, the number of basis curves found.

time_window The effective analysis window used (or NULL).

References

The BPC method is described in [doi:10.1371/journal.pcbi.1008710](https://doi.org/10.1371/journal.pcbi.1008710), with a reference Python implementation at https://github.com/MultimodalNeuroimagingLab/bpc_jupyter.

See Also

`crp`, `crp_cluster`, `naive_nmf`

Examples

```
# Three response shapes, several stimulation groups per shape.

set.seed(1)
n_time <- 300L
tt <- seq(-0.2, 1, length.out = n_time)
shapes <- list(
  exp(-((tt - 0.08) / 0.03)^2) - 0.5 * exp(-((tt - 0.18) / 0.04)^2),
  exp(-((tt - 0.38) / 0.03)^2) - 0.5 * exp(-((tt - 0.50) / 0.04)^2),
  exp(-((tt - 0.70) / 0.04)^2)
)

# 3 stimulation groups per shape, 12 trials each
V <- NULL
groups <- NULL
g <- 0L
for (s in seq_along(shapes)) {
  for (rep in seq_len(3L)) {
    g <- g + 1L
    trials <- outer(shapes[[s]], runif(12L, 0.5, 1.5)) +
      matrix(rnorm(n_time * 12L, sd = 0.2), n_time, 12L)
    V <- cbind(V, trials)
    groups <- c(groups, rep(g, 12L))
  }
}

res <- bpc(V, groups, time = tt, time_window = c(0, 1), verbose = TRUE)
res$n_bpc
res$clusters
plot(res)
```

butter_max_order	<i>'Butterworth' filter with maximum order</i>
------------------	--

Description

Large filter order might not be optimal, but at least 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	<i>Common Average Re-referencing by Least Anti-Correlation (CARLA)</i>
-------	--

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 time x trials x channels; if a matrix is provided, it is treated as time x channels (single trial). A filearray (class FileArray) is also accepted, allowing very large inputs to stay on disk instead of in memory. 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 $\max(2, \text{ceiling}(0.1 * \text{nchan_good}))$, 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 $n = 2, \dots, N$, 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 un-referenced 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. Compute the reference yourself as the channel-wise mean (or median) over these channels and subtract it from 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 $\text{length}(\text{order}) \times \text{nboot}$ (or a $\text{length}(\text{order})$ vector for a single trial / $\text{nboot} = 1$) 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](https://doi.org/10.1016/j.jneumeth.2024.110153); the modified CARLA precursor (`virtual_reference = TRUE`) is described in [doi:10.1016/j.jneumeth.2025.110461](https://doi.org/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)

# time x trials x channels
x_full <- array(rnorm(length(tt) * ntrial * nchan, sd = 5),
               dim = c(length(tt), ntrial, nchan))
for (ch in resp_ch) {
  for (k in seq_len(ntrial)) {
    x_full[, k, ch] <- x_full[, k, ch] + ep * runif(1, -0.8, 1.2)
  }
}
for (ch in noise_ch) {
  for (k in seq_len(ntrial)) {
    tmp <- x_full[, k, ch]
    x_full[, k, ch] <- 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, 1L, 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[, k, ch] <- notch_filter(
      x_full[, k, ch], 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! (time x trials)
car_full <- apply(x_full[, , fit$channels, drop = FALSE], c(1, 2), mean)

# old-style: using all channels for CAR
car_old <- apply(x_full, c(1, 2), mean)

x_reref <- sweep(x_full, c(1, 2), car_full, "-")
x_compare <- sweep(x_full, c(1, 2), car_old, "-")

# ---- 5. Inspect: evoked potential is preserved on responsive channels --
# `plot_signals` expects channels x time, so transpose each trial-1 slice.
op <- graphics::par(mfrow = c(2, 4), mar = c(4, 4, 2, 1))
ravetools::plot_signals(
  signals = t(x_full[, 1, ]),
  sample_rate = srate,
  main = "Trial 1 - (raw)")

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

ravetools::plot_signals(
  signals = t(x_reref[, 1, ]),

```

```

    sample_rate = srate,
    main = sprintf("CARLA-ref (n=%d)", length(fit$channels)))

ravetools::plot_signals(
  signals = t(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)

# Trial-average -> time x channels, ready for `matplot(tt, .)`
graphics::matplot(tt, apply(x_full, c(1, 3), mean),
  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
  main = "Trial-averaged (raw)", col = col)

graphics::matplot(tt, apply(x_clean, c(1, 3), mean),
  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
  main = "Notch-filtered", col = col)

graphics::matplot(tt, apply(x_reref, c(1, 3), mean),
  type = "l", lty = 1, xlab = "Time (s)", ylab = "uV",
  main = "CARLA-referenced", col = col)

graphics::matplot(tt, apply(x_compare, c(1, 3), mean),
  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

<code>x</code>	A numeric multi-mode tensor (array), without NA
<code>keep</code>	Which dimension to keep
<code>...</code>	passed to other methods
<code>average</code>	collapse to sum or mean
<code>transform</code>	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
  })
```

color_ramp_continuous *Map continuous values to colors*

Description

Linearly maps a numeric vector onto a color ramp, clamping values outside the given range to the range's endpoints.

Usage

```
color_ramp_continuous(
  values,
  clim = range(values, na.rm = TRUE),
  cmap = grDevices::hcl.colors(11),
  ...
)
```

Arguments

values	numeric vector of values to map to colors
clim	length-two numeric vector giving the value range to map from; values outside [clim[1], clim[2]] are clamped to the nearer endpoint before mapping. Default is range(values, na.rm = TRUE)
cmap	the color ramp to map onto: either a vector of colors (passed to colorRamp to build the ramp function) or a function such as the one returned by colorRamp that takes a numeric vector with elements in [0, 1] and returns an n x 3 (or n x 4, with alpha) matrix of color-channel values in [0, 255]. Default is grDevices::hcl.colors(11)
...	passed to colorRamp when cmap is a vector of colors (for example, alpha = TRUE to build an alpha-aware ramp)

Value

A character vector of 'RRGGBB' (or 'RRGGBBAA') color strings, the same length as values.

Examples

```

x <- rnorm(100)
col <- color_ramp_continuous(x)

plot(x, col = col, pch = 16)

# Change color palettes with vector of colors
col <- color_ramp_continuous(
  x, cmap = c("lightgreen", "white", "pink"))
plot(x, col = col, pch = 16)

# Using colorRamp
col <- color_ramp_continuous(
  x, cmap = colorRamp(c("black", "orangered", "orange")))
plot(x, col = col, pch = 16)

# Using color ramp palette `function(n) { ... }`
col <- color_ramp_continuous(
  x, cmap = hcl.colors, palette = "Blue-Red 3")
plot(x, col = col, pch = 16)

# Set range
col <- color_ramp_continuous(
  x, clim = c(0, 1),
  cmap = c("black", "orangered", "orange"))
plot(x, col = col, pch = 16)

```

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

<code>x</code>	one-dimensional signal vector, two-dimensional image, or three-dimensional volume; numeric or complex
<code>filter</code>	kernel with the same number of dimensions as <code>x</code>

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,
  detect_onset = FALSE,
  onset_search_start = NULL
)
```

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).

<code>remove_artifacts</code>	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’.
<code>artifact_interval</code>	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 .
<code>artifact_p_threshold</code>	numeric, p-value threshold below which a trial is flagged as artifact (provided its mean projection is also below the cohort mean); defaults to $1e-5$.
<code>threshold_quantile</code>	numeric in $(0, 1)$; the fraction of the peak mean projection magnitude used to derive the duration-uncertainty bounds <code>tau_R_lower</code> and <code>tau_R_upper</code> . Defaults to 0.98 as in the manuscript.
<code>time_step</code>	integer, sampling step (in samples) used when sweeping candidate response duration; defaults to 5L, matching the MATLAB <code>t_step</code> . Larger values are faster but smooth the projection profile.
<code>detect_onset</code>	logical; if TRUE, estimate the response <i>onset</i> τ_{onset} via a reverse cumulative-projection scan (see ‘Details’). The forward canonical shape and per-trial weights are left unchanged; only the <i>reported</i> canonical shape C is restricted to the active response support $[\tau_{onset}, \tau_R]$. The onset may fall before <code>t_start</code> (down to <code>onset_search_start</code>). Defaults to FALSE.
<code>onset_search_start</code>	numeric scalar or NULL; the earliest time (in seconds) the onset scan may reach when <code>detect_onset</code> = TRUE. NULL (the default) uses the first loaded time point, so the scan can look across the entire signal before τ_R . Raise it to keep the scan away from an early stimulation artifact. Clamped into $[\min(\text{time}), t_{\text{end}}]$.

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 $p < \text{artifact_p_threshold}$ and mean projection below the cohort mean are dropped, and CRP is re-run.

When `detect_onset = TRUE`, a complementary reverse pass estimates the response *onset*. The retained trials between `onset_search_start` and τ_R are time-reversed, and the same cumulative cross-projection profile is computed growing backward from τ_R . The backward duration that maximizes cross-trial consistency marks τ_{onset} , the time at which trials begin to share structure - useful when the response is delayed by an unknown latency. Because the scan can extend before `t_start`, the onset may be earlier than the analysis window. This pass only locates a time: it re-uses the forward loading, so the canonical shape and per-trial weights are unchanged (the reported `C` is merely sliced to $[\tau_{onset}, \tau_R]$).

Time points whose data are not finite (any NA, NaN or Inf across trials) are dropped before analysis. `t_start`, `t_end` and `onset_search_start` are clamped into the available time range rather than triggering an error.

Value

A named list with the following elements:

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

- `C` Numeric vector, the reported canonical response shape $C(t)$, taken as the slice of `C_full` over its active support (so it shares identical values with `C_full`). By default this is $[t_{start}, \tau_R]$, where it equals the first eigenvector of the linear kernel PCA on V_{tR} (oriented to the mean trace, unit-norm, length T_R). When `detect_onset = TRUE` it is restricted to $[\tau_{onset}, \tau_R]$ (so its norm is then ≤ 1). The matching time axis is `params_times`.
- `C_full` Numeric vector spanning the *entire* loaded time range (every row of time, including any baseline at $t < 0$). Obtained by applying the trial-space loading to the full retained data, so `C_full` coincides exactly with the untrimmed forward `C` on $[t_{start}, \tau_R]$ and extrapolates the shape everywhere else. Time axis is `params_times_full`. Outside $[t_{start}, \tau_R]$ cross-trial consistency is not optimized, so those portions are more variable.
- `loading` Numeric vector of length K (retained trials), the trial-space loading $g = V_{tR}^T C = s_1 v_1$ recovered from the forward decomposition. Applying it to a time $\times$ trials matrix and dividing by $\|g\|^2$ reconstructs the canonical shape over that time range; this is how `C_full` is formed.
- `a1` Numeric vector of length K (number of trials), the per-trial alpha coefficient $\alpha_k = C^T V_k$: 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 .
- `a1_p` Numeric vector of length K , alpha-prime $\alpha_k / \sqrt{T_R}$: `a1` 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, computed over the forward window $[t_{start}, \tau_R]$ against the untrimmed forward canonical shape. Access trial k via `ep[, k]`. (When `detect_onset = TRUE` the reported `C` may be shorter than T_R ; `ep` always stays on the full forward window.)
- `eep_root` Numeric vector of length K , $\|\epsilon_k\| = \sqrt{\epsilon_k^T \epsilon_k}$: 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, time axis for the reported C; length matches C (T_R , or the onset-trimmed support when `detect_onset` = TRUE).

`params_times_full` Numeric vector, time axis for C_full; equals the full loaded time (every row, including the baseline at $t < 0$).

`V_tR` Numeric matrix $T_R \times K$, trial matrix truncated to τ_R - the data actually decomposed. Together with `ep` and `avg_trace_tR` it always spans the full forward window $[t_{start}, \tau_R]$, independently of `detect_onset`.

`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 .

`tR_sample` Integer, row (sample) index of τ_R within the windowed data; `V[seq_len(tR_sample), ]` is the truncated trial matrix `V_tR`.

`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 (H_1 : 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.

`tau_onset`, `tau_onset_lower`, `tau_onset_upper` Numeric scalars, the estimated response onset time (seconds) and its lower/upper threshold-crossing bounds, from the reverse projection scan; all NA unless `detect_onset` = TRUE.

`onset` List with the reverse-scan profile (`onset_tpts` and `mean_proj_profile`, mapped onto original time); NULL unless `detect_onset` = TRUE (also NULL when the search window had too few samples).

`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](https://doi.org/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(-0.5, 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)

# ---- Onset detection (reverse scan) on the same data -----
res_onset <- crp(V, tt, detect_onset = TRUE)
c(tau_onset = res_onset$tau_onset, tau_R = res_onset$tau_R)

```

crp_cluster

Cluster electrodes by their canonical CRP response shape

Description

Groups recording electrodes by the *shape* of their canonical evoked response. Each electrode is summarized by [crp](#) into one amplitude-normalized canonical shape; an electrode-by-electrode similarity matrix is factorized with [naive_nmf](#) to find clusters that share a response shape, and a representative basis profile curve is extracted per cluster. This applies the basis-profile-curve (BPC) approach *across electrodes* rather than across stimulation sites; see ‘References’.

Usage

```

crp_cluster(
  crp_list,
  paired = TRUE,
  time_window = c(0, NA),
  n_clusters = NULL,
  initial_rank = NULL,
  zeta_threshold = 1,
  null_class = TRUE,
  nmf_max_iters = 10000,
  nmf_tol = c(1e-04, 1e-08),
  verbose = TRUE
)

```

Arguments

crp_list	a named list of <code>ravetools_crp</code> objects (one per electrode) from crp ; list names are used as electrode labels. Time axes need not match - the common overlapping domain is used (an error is raised if the domains do not overlap). See ‘Details’.
----------	--

paired	logical; TRUE (the default) builds the trial-level BPC t-statistic similarity and requires the single trials in <code>.data\$V</code> ; FALSE uses the cosine similarity of the <code>C_full</code> curves. See ‘Details’.
time_window	numeric <code>c(lo, hi)</code> analysis window, clipped into the common domain; an NA extends to that domain edge. Defaults to <code>c(0, NA)</code> (stimulus onset to the end of the domain).
n_clusters	integer or NULL; if given, fixes the number of clusters and skips the automatic selection (which otherwise uses <code>zeta_threshold</code>).
initial_rank	integer or NULL; the starting NMF rank for automatic selection. NULL uses <code>min(length(crp_list) - 1L, 10L)</code> .
zeta_threshold	numeric > 0 ; rank-selection cutoff on ζ (see ‘Details’); smaller values give fewer clusters. Defaults to 1, matching the BPC reference.
null_class	logical; if TRUE (the default) electrodes whose winner-take-all loading falls below $1/(2\sqrt{N})$ are left unassigned (NA); if FALSE every electrode is forced into a cluster. See ‘Details’.
nmf_max_iters, nmf_tol	passed to naive_nmf .
verbose	logical; whether to report progress.

Details

1. **Common window.** Electrodes may have different time axes ([crp](#) drops NA samples); the overlapping time domain is used, `time_window` is clipped into it, and each electrode is subset on the fly with no interpolation.
2. **Similarity.** With `paired = TRUE` each entry is the one-sample t-statistic, across an electrode pair’s common trials, of the per-trial CCEP cross-projection of the $1/\alpha$ -rescaled responses; negatives are zeroed, the matrix made symmetric and scaled to a maximum of one. With `paired = FALSE` it is the cosine cross-projection of the `C_full` curves.
3. **Rank selection.** [naive_nmf](#) factorizes the similarity at rank Q (from `initial_rank`); each rank is re-run a few times and the lowest-error fit kept. Q is reduced while ζ - the sum of the upper off-diagonal of the row-normalized $HH^\top$ - exceeds `zeta_threshold`.
4. **Assignment.** Each electrode takes its winner-take-all cluster over the normalized NMF loadings; with `null_class`, loadings below $1/(2\sqrt{N})$ are left unassigned.
5. **Basis curves.** Per cluster, the first linear kernel PCA component of its members’ `C_full` curves (as in [crp](#)).

Value

A named list of class `ravetools_crp_cluster`:

`clusters` Integer vector, the cluster index assigned to each electrode (winner-take-all over the row-normalized NMF loadings). When `null_class = TRUE`, electrodes whose top loading falls below $1/(2\sqrt{N})$ are left unassigned (NA).

`basis_curves` Numeric matrix, $\text{time} \times \text{number of clusters}$; column q is the basis profile curve $B_q(t)$ for cluster q , the first linear-kernel PCA component of its member `C_full` curves, sign-oriented to the cluster mean.

basis_times Numeric vector, the time axis for **basis_curves** (the common overlapping time axis, restricted to **time_window**).

similarity Numeric matrix, the electrode-by-electrode similarity Ξ that was factorized (non-negative, scaled to a maximum of one).

nmf The **naive_nmf** result for the selected rank.

n_clusters Integer, the number of clusters found.

domain Numeric $c(lo, hi)$, the overlapping time domain shared by all electrodes.

paired,time_window The settings used; **time_window** is the effective window after clipping into domain.

References

The BPC method is described in [doi:10.1371/journal.pcbi.1008710](https://doi.org/10.1371/journal.pcbi.1008710); the underlying CRP method in [doi:10.1371/journal.pcbi.1011105](https://doi.org/10.1371/journal.pcbi.1011105).

See Also

[crp](#), [naive_nmf](#)

Examples

Four response shapes; shapes 3 and 4 differ only in amplitude, so they
cluster together once shapes are amplitude-normalized.

```
n_time <- 300L
tt <- seq(-0.2, 1, length.out = n_time)
shapes <- list(
  exp(-((tt - 0.08) / 0.03)^2) - 0.5 * exp(-((tt - 0.18) / 0.04)^2),
  exp(-((tt - 0.38) / 0.03)^2) - 0.5 * exp(-((tt - 0.50) / 0.04)^2),
  exp(-((tt - 0.70) / 0.04)^2),
  exp(-((tt - 0.70) / 0.04)^2) * 2
)

# 4 electrodes per shape, each parameterized with crp()
crp_list <- list()
for (g in seq_along(shapes)) {
  for (e in seq_len(4L)) {
    V <- outer(shapes[[g]], runif(15L, 0.5, 1.5)) +
      matrix(rnorm(n_time * 15L, sd = 0.15), n_time, 15L)
    crp_list[[sprintf("elec_%d_%d", g, e)]] <-
      crp(V, tt, remove_artifacts = FALSE)
  }
}

res <- crp_cluster(crp_list, verbose = TRUE)
res$n_clusters
table(res$clusters)
plot(res)
```

decimate	<i>Decimate with 'FIR' or 'IIR' filter</i>
----------	--

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	<i>Design a digital filter</i>
---------------	--------------------------------

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 <code>design_filter_iir</code> for 'IIR' methods (ignored for 'FIR'). When TRUE (default), filtering is done via 'SOS' form using <code>gsignal::filtfilt</code> ; when FALSE, the <code>ravetools</code> 'ARMA' <code>filtfilt</code> 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	<i>Design 'FIR' filter using firls</i>
-------------------	--

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

<code>data_size</code>	minimum length of data to apply the filter, used to decide the maximum filter order. For 'FIR' filter, data length must be greater than $3 \times \text{filter_order}$
<code>high_pass_freq</code>	high-pass frequency; default is NA (no high-pass filter will be applied)
<code>high_pass_trans_freq</code>	high-pass frequency band-width; default is automatically inferred from data size. Frequency $\text{high_pass_freq} - \text{high_pass_trans_freq}$ is the corner of the stop-band
<code>low_pass_freq</code>	low-pass frequency; default is NA (no low-pass filter will be applied)
<code>low_pass_trans_freq</code>	low-pass frequency band-width; default is automatically inferred from data size. Frequency $\text{low_pass_freq} + \text{low_pass_trans_freq}$ is the corner of the stop-band
<code>stopband_attenuation</code>	allowable power attenuation (in decibel) at transition frequency; default is 40 dB.
<code>scale</code>	whether to scale the filter for unity gain
<code>method</code>	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	<i>Design an 'IIR' filter</i>
-------------------	-------------------------------

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

<code>x</code>	numerical or complex, a vector or a matrix
<code>trend</code>	the trend of the signal; choices are 'constant' and 'linear'
<code>break_points</code>	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	<i>Show channel signals with diagnostic plots</i>
------------------	---

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 srates is too high; will be automatically generated if NULL
srates	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 srates 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),
  vlins = 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 = "#BEBE33", 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 = "#BEBE33", 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)

```

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	<i>Calculate massive covariance matrix in parallel</i>
----------	--

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

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

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

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

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.

Usage

```
fftw_r2c(data, HermConj = 1L, fftwplanopt = 0L, ret = NULL)

fftw_c2c(data, inverse = 0L, fftwplanopt = 0L, ret = NULL)

fftw_c2r(data, HermConj = 1L, fftwplanopt = 0L, ret = NULL)

mvfftw_r2c(data, fftwplanopt = 0L, HermConj = 0L, ret = NULL)

mvfftw_c2c(data, inverse = 0L, fftwplanopt = 0L, ret = NULL)

mvfftw_c2r(data, fftwplanopt = 0L, retrows = 0L, ret = NULL)

fftw_r2c_2d(data, HermConj = 1L, fftwplanopt = 0L, ret = NULL)

fftw_c2c_2d(data, inverse = 0L, fftwplanopt = 0L, ret = NULL)

fftw_r2c_3d(data, HermConj = 1L, fftwplanopt = 0L, ret = NULL)

fftw_c2c_3d(data, inverse = 0L, fftwplanopt = 0L, ret = NULL)
```

Arguments

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

retrows Integer; expected number of rows of the time-domain signal for `mvfftw_c2r` when reconstructing from a one-sided spectrum.

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,
  close_radius = NULL,
  resolution = 256L,
  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
close_radius	radius (in 'voxels' along the finest 'IJK' axis; see IJK2RAS) of the morphological closing (dilate-then-erode) operation used to connect the embedded surface into a water-tight shell <i>and</i> to remove small topological handles/tunnels (for example a hole drilled through an otherwise solid structure) before the surface is filled; default is NULL, which derives a radius from the surface geometry (capped at 15 'voxels'). Handles/tunnels wider than roughly twice this radius (in physical, RAS space) will not be closed, so when the input surface is known to contain larger topological defects, set this to a larger number explicitly. Larger radii cost more time and memory (the closing operates on a dense resolution^3 volume) and - because the operation is performed in a fixed-size volume - must leave enough margin that the dilated shell does not reach the volume boundary; radii that are too large for the working cube will be capped with a warning
resolution	number of 'voxels' along each margin of the working cube volume that the surface is embedded into; default is 256. Larger values increase the spatial precision of the fill and the closing operation (at the cost of memory and computation time, which scale with resolution^3); the value must leave enough room for the surface (transformed into 'IJK' space via IJK2RAS) plus the closing margin to fit inside the cube
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. When the linear part of IJK2RAS encodes voxel sizes that differ across axes (pixdim, i.e. the three 'IJK' axes do not advance by the same physical distance per 'voxel'), the dilate/erode operations underlying close_radius and inflate automatically use a different number of 'voxels' along each axis so that the resulting structuring element stays isotropic in physical (RAS) space
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	<i>Filter one-dimensional signal</i>
---------------	--------------------------------------

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	<i>Forward and reverse filter a one-dimensional signal</i>
----------	--

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	<i>Least-squares linear-phase FIR filter design</i>
-------	---

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	<i>Frequency response of digital filter</i>
--------	---

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	<i>Apply gamma-tone filters to obtain auditory envelopes</i>
----------------	--

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

<code>x</code>	a numeric vector or matrix; if <code>x</code> is a matrix, it should be column-major (each column is a sound track)
<code>sample_rate</code>	sampling frequency
<code>center_frequencies</code>	center frequencies at which the envelopes will be derived; can be either a length of two defining the lower and upper bound, and using <code>n_bands</code> to interpolate automatically, or a length of multiple, with the frequencies specified explicitly
<code>n_bands</code>	number of the center frequencies, can be missing if <code>center_frequencies</code> is explicit and no interpolation is needed; if specified, then the frequencies will be interpolated using equivalent rectangular bandwidth rate ('ERB')
<code>use_hilbert</code>	whether to apply 'Hilbert' transform; default is true, which calculates the magnitude; set to false when only the filter is needed
<code>downsample</code>	whether to down-sample the envelopes after the filters; default is NA (no down-sample).
<code>downsample_before_hilbert</code>	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 <code>downsample</code> 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	<i>Left 'Hippocampus' of 'N27-Collin' brain</i>
-----------------------	---

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	<i>'Matlab' heat-map plot palette</i>
----------------	---------------------------------------

Description

'Matlab' heat-map plot palette

Usage

matlab_palette()

Value

vector of 64 colors

mesh_from_volume	<i>Generate 3D mesh surface from volume data</i>
------------------	--

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)
```

mris_curvature

Estimate per-node curvature of a surface mesh

Description

Estimates, at every vertex of a closed triangular mesh, the local mean curvature, Gaussian curvature, and the two principal curvatures, by fitting an osculating quadratic surface to each vertex's neighborhood.

Usage

```
mris_curvature(mesh, verbose = FALSE)
```

Arguments

mesh	triangular mesh of class 'mesh3d' (or coercible via ensure_mesh3d). The mesh must be closed, manifold, and genus-0 (watertight, no boundary or non-manifold edges, single connected component); mris_curvature raises an error if such defects are detected
verbose	logical; print progress messages. Default FALSE

Details

For each vertex, the function:

1. builds an orthonormal tangent frame (e_1, e_2, n) , where n is the vertex's (already-computed) unit normal;
2. expresses each neighbor's offset from the vertex in this frame as tangential coordinates (u, w) and a height h above the tangent plane along n ;
3. fits, by least squares over the vertex's 2-ring neighborhood, the osculating paraboloid $h = au^2 + buw + cw^2$;
4. derives the mean curvature $H = a + c$, the Gaussian curvature $K = 4ac - b^2$, and the principal curvatures $k_{1,2} = H \pm \sqrt{\max(H^2 - K, 0)}$.

Because the tangent frame is orthonormal and the paraboloid is fitted with zero gradient at the vertex (by construction, since n is the fitted normal direction), these are exactly the eigenvalues of the second fundamental form, i.e. the principal curvatures.

The sign of the result follows the orientation of the per-vertex outward normal: a locally convex ('gyrus-like') patch, where neighbors lie toward the surface's interior relative to the outward normal, has negative mean curvature, while a locally concave ('sulcus-like') patch has positive mean curvature. For example, a sphere of radius r with outward-pointing normals has uniform curvature $H = -1/r$ and $K = 1/r^2$ everywhere.

Vertices whose 2-ring neighborhood is degenerate (fewer than three neighbors, or neighbor offsets that do not span the tangent plane, e.g. nearly collinear) are reported with all four curvature values set to zero.

Value

A named list of four numeric vectors, each of length `ncol(mesh$vb)` (one entry per vertex, in vertex order):

`mean` Mean curvature $H = (k_1 + k_2)/2$.

`gaussian` Gaussian curvature $K = k_1 k_2$.

`k1` First principal curvature ($k_1 \geq k_2$).

`k2` Second principal curvature.

References

Cortical surface-based analysis II: Inflation, flattening, and a surface-based coordinate system. *NeuroImage*, 9(2), 195-207 (1999).

Examples

```
if (is_not_cran()) {

data("left_hippocampus_mask")
mesh <- vcg_isosurface(left_hippocampus_mask)

plot(mesh)

# Fix defects
mesh <- vcg_fix_defects(mesh, verbose = TRUE)

# Smooth
smoothed <- mris_smooth(mesh, verbose = TRUE)

res <- mris_curvature(smoothed)

range(res$k1)

col <- color_ramp_continuous(
  res$k1, clim = c(-1, 1), alpha = TRUE,
  cmap = c("black", "gray", "red"))

plot(smoothed, col = list(col),
     eye = c(-100, 100, 0), up = c(0, 0, 1))

}
```

mris_inflate	<i>Inflate a cortical surface mesh</i>
--------------	--

Description

Iteratively relaxes a closed cortical-surface mesh into a smoother, more compact "inflated" shape, while tracking how far each vertex moves inward along the surface normal as it goes. Returns both the inflated mesh and this per-vertex depth map (sulc).

Usage

```
mris_inflate(
    mesh,
    n_averages = 16L,
    niterations = 10L,
    l_spring_norm = 1,
    l_dist = 0.1,
    momentum = 0.9,
    dt = 0.9,
    desired_rms = 0.015,
    scale_brain = TRUE,
    verbose = FALSE
)
```

Arguments

mesh	triangular mesh of class 'mesh3d', typically a white-matter surface in millimeters. The mesh must be closed, manifold, and genus-0 (watertight, no boundary or non-manifold edges, single connected component) - this is a hard precondition of the underlying algorithm. An open or defective mesh (e.g. a raw surface extracted via vcg_isosurface , which often contains small extraction cracks) will silently inflate into a distorted, "sausage"-shaped result rather than a smooth sphere-like surface, and mris_inflate will raise an error if such defects are detected.
n_averages	starting number of gradient-averaging passes (outer loop); halved each level down to 0. Default 16.
niterations	number of inner iterations per averaging level. Default 10.
l_spring_norm	normalized spring term coefficient. Default 1.0.
l_dist	distance-preservation coefficient (before per-level scaling). Default 0.1.
momentum	momentum coefficient. Default 0.9.
dt	time step. Default 0.9.
desired_rms	target RMS tangent-plane height for early stopping of each inner loop. Default 0.015 (mm).
scale_brain	logical; whether to scale the mesh to the canonical surface area (110,000 mm^2) before inflation. Set FALSE only for non-standard inputs. Default TRUE.
verbose	logical; print per-iteration progress. Default FALSE.

Details

The implementation follows the inflation procedure described in the literature (see **References**):

1. Optionally rescale the mesh to a canonical surface area ($110,000 \text{ mm}^2$, the normalization target used by the reference procedure).
2. Outer loop: a neighborhood-averaging size `n_averages` is halved at each level (16, 8, ..., 0).
3. Inner loop (niterations repetitions per level):

Distance term Restoring force pulling each vertex back towards its original distances to its neighbors, with coefficient `l_dist * sqrt(n_averages)`.

Gradient averaging Smooth the distance-term gradient over `n_averages` neighborhood passes before adding the spring term.

Normalized spring term Laplacian (neighbor-averaging) smoothing scaled by `sqrt(orig_area / current_area)`, with coefficient `l_spring_norm`; added *after* gradient averaging.

Momentum step Update vertex positions using momentum integration with a 1 mm per-step displacement cap.

Depth accumulation Accumulate the normal-projected component of each step into the per-vertex depth map (`sulc`).

4. Center the mesh, rescale it, and zero-mean the depth map (`sulc`).

Value

A named list:

`mesh` Inflated surface as a 'mesh3d' object with `vb`, `it`, and `normals`.

`sulc` Numeric vector of the per-vertex depth values described above (zero-mean, in mesh units).

References

Cortical surface-based analysis II: Inflation, flattening, and a surface-based coordinate system. *NeuroImage*, 9(2), 195-207 (1999).

Examples

```
if (is_not_cran()) {

  data("left_hippocampus_mask")
  mesh <- vcg_isosurface(left_hippocampus_mask)

  # Fix defects
  mesh <- vcg_fix_defects(mesh, verbose = TRUE)

  # Center the mesh
  mesh$vb[1:3, ] <- mesh$vb[1:3, ] - rowMeans(mesh$vb[1:3, ])

  # Inflate the surface while keeping the node distances
  result <- mris_inflate(mesh, n_averages = 4L, niterations = 5L,
    scale_brain = FALSE, verbose = TRUE)
```

```

# Visualize with the sulcal values
pal <- colorRampPalette(c("black", "gray", "red"))(128)
col <- pal[pmax(pmin(round(result$sulc * 10 + 64), 128), 1)]

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

plot(
  mesh, col = col,
  eye = c(0, 100, 0),
  up = c(1, 0, 0))

plot(
  result$mesh, col = col,
  eye = c(0, 100, 0),
  up = c(1, 0, 0))

}

```

mris_make_surfaces

Localize white-matter and pial surfaces from an intensity volume

Description

Starting from a single closed surface mesh (typically a smoothed estimate of the white-matter boundary, in the same physical coordinate space as volume) and a co-registered intensity volume (such as a normalized T1 scan), iteratively deforms the surface in two passes to localize the white/gray-matter and gray-matter/CSF tissue-intensity boundaries, producing a white and a pial surface.

Usage

```

mris_make_surfaces(
  mesh,
  volume,
  white_intensity,
  pial_intensity,
  IJK2RAS = NULL,
  max_thickness = 5,
  step_size = 0.4,
  n_averages = 4L,
  niterations = 10L,
  l_intensity = 1,
  l_spring = 0.5,
  momentum = 0.9,
  dt = 0.5,

```

```

    verbose = FALSE
)

```

Arguments

mesh	triangular mesh of class 'mesh3d' (or coercible via ensure_mesh3d), in the same physical (RAS) coordinate space as volume. As with mris_inflate , the mesh must be closed, manifold, and genus-0 (watertight, no boundary or non-manifold edges, single connected component); mris_make_surfaces raises an error if such defects are detected
volume	a 3-dimensional numeric array of image intensities (such as a normalized T1 volume), co-registered with mesh
white_intensity	target intensity for the white/gray-matter boundary; the value the white-surface pass searches for along each vertex normal. For a normalized T1 volume this is typically close to the midpoint between the white-matter and gray-matter intensities
pial_intensity	target intensity for the gray-matter/CSF boundary, analogous to white_intensity for the pial-surface pass
IJK2RAS	volume IJK (zero-indexed voxel index) to surface tkrRAS transform (a 4x4 matrix); default is NULL, which assumes volume is a conformed volume (LIA orientation, 1mm isotropic 'voxels', centered at the volume midpoint, the same convention fill_surface defaults to) and derives the transform from dim(volume); set this explicitly when volume uses a different orientation or voxel size
max_thickness	half-width, in mm, of the per-vertex normal search window the intensity-target term samples. Default 5
step_size	sampling step, in mm, along the search window. Default 0.4
n_averages	number of 1-ring gradient-averaging passes applied to the intensity-target gradient before the smoothness term is added (mirrors mris_inflate 's gradient-averaging step). Default 4
niterations	number of deformation iterations <i>per pass</i> (white, then pial). Default 10
l_intensity	intensity-target term coefficient. Default 1.0
l_spring	smoothness (1-ring Laplacian spring) term coefficient. Default 0.5
momentum	momentum coefficient. Default 0.9
dt	time step. Default 0.5
verbose	logical; print per-pass progress. Default FALSE

Details

The implementation keeps the two dominant ideas of the surface-placement procedure described in the literature (see **References**):

1. **Intensity-target localization:** for each vertex, sample volume along the vertex's current normal at offsets spanning $\pm \text{max_thickness}$ in steps of `step_size`, and pull the vertex toward the offset whose sampled intensity is closest to a single target value (`white_intensity` for the white-surface pass, `pial_intensity` for the pial-surface pass).

2. **Smoothness:** a 1-ring Laplacian spring keeps the mesh regular while the per-vertex intensity term, which reacts independently to noisy image data, pulls vertices toward the tissue boundary.

Both terms are integrated with the same gradient-averaging and momentum-integration machinery `mris_inflate` and `mris_sphere` use: each inner iteration clears the gradient, adds the intensity-target term, smooths it over `n_averages` passes of 1-ring averaging, adds the locally-acting smoothness term (which is not itself smoothed), then takes a momentum-integration step with a 1 mm per-step displacement cap, and refreshes the vertex normals.

The white-surface pass runs first, for `niterations` iterations; the pial-surface pass then continues from its result, with the momentum velocity reset to rest, toward `pial_intensity` for another `niterations` iterations.

This is a *reduced* port: the literature's procedure is a multi-resolution optimization over roughly seven weighted energy terms (intensity, intensity gradient, smoothness, self-intersection repulsion, curvature, and more), using per-vertex gray/white/CSF intensity statistics derived from a prior segmentation. Reproducing that faithfully is out of scope for this package; `white_intensity` and `pial_intensity` are supplied directly here instead, for example the midpoints between the typical white-matter/gray-matter and gray-matter/CSF intensities of volume.

Value

A named list of two 'mesh3d' surfaces (each with `vb`, `it`, and `normals`):

`white` Surface localized to `white_intensity`.

`pial` Surface localized to `pial_intensity`, continuing from `white`.

References

Cortical surface-based analysis I: Segmentation and surface reconstruction. *NeuroImage*, 9(2), 179-194 (1999).

Examples

```
if (is_not_cran()) {

  data("left_hippocampus_mask")
  n_vox <- length(left_hippocampus_mask)

  volume <- left_hippocampus_mask + runif(n = n_vox, 0, 1)
  vox2ras <- diag(1, 4)
  mesh <- vcg_isosurface(volume, threshold_lb = 0.99)

  plot(mesh)

  # Fix defects
  mesh <- vcg_fix_defects(mesh, verbose = TRUE, merge_tolerance = 1.75)

  res <- mris_make_surfaces(
```

```

    mesh,
    volume,
    pial_intensity = 1.1,
    white_intensity = 1,
    IJK2RAS = vox2ras
)

plot(res$pial)

}

```

mris_remesh

Isotropic re-triangulation of a surface mesh

Description

Re-triangulates a closed surface mesh so that all edges approach a uniform target length, improving triangle regularity and quality without changing the surface topology or shape. The output has a different vertex and face count than the input but closely tracks the original surface geometry.

Usage

```

mris_remesh(
  mesh,
  target_edge_length = NULL,
  niterations = 5L,
  n_smooth = 2L,
  damping = 0.99,
  verbose = FALSE
)

```

Arguments

mesh	triangular mesh of class 'mesh3d' (or coercible via ensure_mesh3d). The mesh must be closed, manifold, and genus-0 (watertight, no boundary or non-manifold edges, single connected component); mris_remesh raises an error if such defects are detected
target_edge_length	numeric; desired uniform edge length in the same units as mesh\$vb. Default NULL uses the current average edge length (quality improvement without size change)
niterations	integer; number of split/collapse/smooth iterations. Default 5
n_smooth	integer; number of tangential-smooth passes per iteration. Default 2
damping	numeric; fraction of the tangential displacement applied at each smooth pass (between 0 and 1). Default 0.99
verbose	logical; print per-iteration vertex and face counts. Default FALSE

Details

Each iteration applies three steps following the procedure described in the **References**:

1. **Edge split**: every edge longer than $\frac{4}{3} \times \text{target_edge_length}$ is split at its midpoint. One-edge splits produce 2 sub-triangles; two-edge splits produce 3; three-edge splits produce 4 (the standard 1-to-4 uniform refinement).
2. **Edge collapse**: every edge shorter than $\frac{4}{5} \times \text{target_edge_length}$ is collapsed to its midpoint. A collapse is skipped if it would flip any surrounding face normal (manifold-safety check).
3. **Tangential smoothing**: n_{smooth} passes of uniform-weight 1-ring Laplacian averaging, with each displacement projected onto the vertex tangent plane (the normal component is removed) before application. This improves vertex regularity while keeping vertices near the original surface.

After all iterations, vertex normals are refreshed.

Unlike `vcg_uniform_remesh`, which uses volumetric resampling and may change topology, this function operates entirely on the mesh surface and preserves the input genus and manifold structure.

Value

A 'mesh3d' object with `vb`, `it`, and normals. Vertex and face counts differ from the input; the surface geometry is preserved.

References

A remeshing approach to multiresolution modeling. *Proceedings of Shape Modelling International*, 49-58 (2003).

Examples

```
sphere <- vcg_sphere()
sphere

vcg_average_edge_length(sphere)
plot(sphere)

remeshed <- mris_remesh(
  sphere,
  target_edge_length = 0.3
)

plot(remeshed)
```

mris_smooth	<i>Smooth a surface mesh</i>
-------------	------------------------------

Description

Smooths a closed triangular surface mesh by repeatedly replacing each vertex with the average of itself and its immediate neighbors (Laplacian/neighbor-averaging smoothing), optionally rescaling the surface back to its original area afterwards.

Usage

```
mris_smooth(
    mesh,
    niterations = 10L,
    npasses = 1L,
    rescale = FALSE,
    verbose = FALSE
)
```

Arguments

mesh	triangular mesh of class 'mesh3d'. Must be closed, manifold, and genus-0 (watertight, no boundary or non-manifold edges, single connected component) - the same hard precondition as mris_inflate (and for the same reason: an open or defective mesh introduces a persistent directional bias into the 1-ring averaging). <code>mris_smooth</code> will raise an error if such defects are detected.
niterations	number of include-self 1-ring averaging rounds applied to the vertex positions in each pass. Default 10.
npasses	number of outer smoothing passes. Default 1.
rescale	logical; whether to rescale the surface back to its original area after each pass. Default FALSE.
verbose	logical; print per-pass progress. Default FALSE.

Details

For each of `npasses` passes, the algorithm:

1. Replaces each vertex position by the include-self mean of itself and its 1-ring (directly-connected) neighbors, repeated `niterations` times.
2. Recomputes vertex normals and total surface area.
3. Optionally rescales the surface back to its original area (only if `rescale = TRUE`).

Value

The smoothed surface as a 'mesh3d' object with `vb`, `it`, and `normals`.

Examples

```

if (is_not_cran()) {

  sphere <- vcg_sphere(sub_division = 4L)

  # roughen the sphere slightly so smoothing has something to do
  sphere$vb[1, ] <- sphere$vb[1, ] * (1 + 0.05 * rnorm(ncol(sphere$vb)))

  # Fix defects
  sphere <- vcg_fix_defects(sphere)

  smoothed <- mris_smooth(sphere, niterations = 5L, verbose = TRUE)

  plot_mesh_polygon(
    list(sphere, smoothed),
    alpha = c(0.3, 0.5),
    col = list("gray", "red"),
    main = "Gray: original; red: smoothed"
  )
}

```

mris_sphere

Project a surface onto a sphere and relax metric distortion

Description

Projects a closed triangular surface mesh radially onto a sphere and then iteratively relaxes the metric distortion this introduces ("unfolding"), so that geodesic distances and face orientations stay close to those of the input surface - a step typically used to prepare an inflated cortical surface for spherical registration.

Usage

```

mris_sphere(
  mesh,
  target_radius = 100,
  n_averages = 64L,
  niterations = 25L,
  l_dist = 1,
  l_area = 1,
  momentum = 0.9,
  dt = 0.05,
  verbose = FALSE
)

```

Arguments

mesh	triangular mesh of class 'mesh3d', typically an inflated surface (see mris_inflate). Must be closed, manifold, and genus-0 - the same hard precondition as mris_inflate . mris_sphere will raise an error if such defects are detected.
target_radius	radius of the target sphere. Default 100.
n_averages	number of gradient-averaging passes applied to the distance-term gradient each iteration (the same neighborhood-averaging mechanism mris_inflate uses). Default 64; reduced from the much larger averaging size used in production pipelines (tuned for cortical surfaces with hundreds of thousands of vertices) to a size more practical for typical 'mesh3d' inputs.
niterations	number of unfolding iterations. Default 25.
l_dist	distance-preservation coefficient. Default 1.0.
l_area	folded-face repulsion coefficient. Default 1.0.
momentum	momentum coefficient. Default 0.9.
dt	time step. Default 0.05.
verbose	logical; print per-iteration progress. Default FALSE.

Details

This is a reduced procedure, not a complete reproduction of any particular reference implementation. The full unfolding procedure described in the literature integrates seven weighted energy terms against a separate reference surface through a multi-resolution, multi-thousand line optimization pipeline, which is out of scope for this package. Instead, this implementation keeps the two dominant terms and integrates them with the same momentum-based machinery [mris_inflate](#) uses:

Distance term (`l_dist`) Restoring force pulling each vertex back towards the input mesh's own original distances to its neighbors.

Area term (`l_area`) Repulsive force that acts only on folded/negative-area faces, pushing their vertices apart - this is the actual "unfolding" mechanism.

Both terms are integrated via momentum integration with gradient averaging and a 1 mm per-step displacement cap, exactly as [mris_inflate](#) does.

A further simplification: the reference procedure relaxes a freshly spherical-projected surface against a separate, previously-loaded white-matter reference surface for the distance term; since this function takes a single mesh as input, the input mesh's own metric (captured before projection) is used as the reference instead - the same convention [mris_inflate](#) already uses, and the practical analogue (the white-matter surface is normally what gets inflated to produce a typical input to this kind of unfolding step).

Value

The projected and relaxed surface as a 'mesh3d' object with `vb`, `it`, and `normals`.

Examples

```

if (is_not_cran()) {

  sphere <- vcg_sphere(sub_division = 3L)

  # deform so there is metric distortion to relax
  sphere$vb[1, ] <- sphere$vb[1, ] * (1 + 0.2 * rnorm(ncol(sphere$vb)))

  # Fix defects
  sphere <- vcg_fix_defects(sphere)

  result <- mris_sphere(
    sphere,
    n_averages = 8L,
    niterations = 10L,
    target_radius = 1,
    verbose = TRUE
  )

  plot_mesh_polygon(list(sphere, result),
    col = list("gray", "red"),
    alpha = c(0.3, 0.9))

}

```

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

<code>data_length</code>	length of data
<code>fs</code>	sampling frequency in 'Hz'
<code>frequency_range</code>	frequency range to look at; length of two
<code>time_bandwidth</code>	a number indicating time-half bandwidth product; i.e. the window duration times the half bandwidth of main lobe; default is 5
<code>num_tapers</code>	number of 'DPSS' tapers to use; default is NULL and will be automatically computed from $\text{floor}(2 \times \text{time_bandwidth} - 1)$
<code>window_params</code>	vector of two numbers; the first number is the window size in seconds; the second number if the step size; default is <code>c(5, 1)</code>
<code>nfft</code>	'NFFT' size, positive; see 'Details'
<code>detrend_opt</code>	how you want to remove the trend from data window; options are 'linear' (default), 'constant', and 'off'
<code>data</code>	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 $1\text{E}-7$ with standard deviation at most $1\text{E}-5$. The original copy was licensed under a Creative Commons Attribution 'NC'-'SA' 4.0 International License (<https://creativecommons.org/licenses/by-nc-sa/4.0/>).

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

Value

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

Examples

```
# 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

<code>m</code>	a matrix or a vector to be converted to the Matrix4 instance; <code>m</code> 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	<i>Set or get thread options</i>
------------------	----------------------------------

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_bpc	<i>Plot Basis Profile Curve results</i>
--------------------	---

Description

S3 plot method for ravetools_bpc objects returned by [bpc](#). Draws a three-panel figure: the basis profile curves, the subgroup-by-subgroup significance matrix Ξ ordered by assignment, and the per-subgroup projection weights.

Usage

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

Arguments

x	a ravetools_bpc object.
...	ignored.

Value

Invisibly returns x.

plot.ravetools_crp	<i>Plot CRP results</i>
--------------------	-------------------------

Description

S3 plot method for objects of class ravetools_crp returned by [crp](#). Produces a three-panel figure:

1. Single-trial traces over the entire loaded time range with the mean, the canonical shape $C(t)$ on its active support (solid) and the full-range extension $C_{full}(t)$ (dashed) overlaid; the analysis window edges and, when present, τ_{onset} are marked.
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} , plus the reverse onset profile and τ_{onset} when detect_onset was used.

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_crp_cluster
```

Plot electrode clustering results

Description

S3 plot method for `ravetools_crp_cluster` objects returned by `crp_cluster`. Draws a two-panel figure: the per-cluster basis profile curves, and the electrode-by-electrode similarity matrix ordered by cluster.

Usage

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

Arguments

`x` a `ravetools_crp_cluster` object.
`...` ignored.

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 <code>ravetools_curve</code> .
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 <code>rgl</code> 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	<i>Render one or more meshes as an orthographic dot cloud in base R</i>
--------------------	---

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 <code>c(0, 1, 0)</code> .
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 <code>par("pin")</code>): 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 <code>vcg_update_normals</code> 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 $1 - \text{mesh_clipping}$ are dropped, leaving only the silhouette/grazing band. <code>mesh_clipping = 1</code> (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 <code>alpha = 0</code> 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 <code>clipping_plane</code> 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 <code>clipping_plane</code> is NULL.
...	additional graphical parameters forwarded to <code>plot.default</code> (new plot) or <code>points</code> (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)
)
```

<code>plot_mesh_polygon</code>	<i>Render one or more meshes as flat-shaded triangles in base R</i>
--------------------------------	---

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 = "",
  main = "",
  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 (<code>\$it</code>) are rendered as sphere instances (one vcg_sphere per vertex, radius <code>cex</code>).
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 <code>c(0, 1, 0)</code> .
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 <code>c("white", "gray30")</code> .
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, main	passed to plot.default when <code>add = FALSE</code> .
zoom	positive numeric magnification applied to the auto-computed axis limits when <code>xlim</code> or <code>ylim</code> is NULL. Values greater than 1 zoom in, values in $(0, 1)$ zoom out. When <code>asp</code> is set, the zoom is plot-region aware (queried via <code>par("pin")</code>): the axis that already fills the device after <code>asp</code> -induced expansion is zoomed by exactly <code>zoom</code> , and the other axis is zoomed less so the data fills more of the plot region while preserving the requested <code>asp</code> . Ignored for any axis whose limit was supplied explicitly, and when <code>add = TRUE</code> . 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 <code>mesh_clipping</code> 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 <code>alpha = 0</code> 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 <code>par("fg")</code> , 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 $\max(\text{shade}, \text{ambient_intensity})$ 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 <code>plot.default</code> (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,
  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	<i>Project plane to a surface</i>
---------------	-----------------------------------

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

<code>x</code>	numerical vector or a row-major vector, signals. If <code>x</code> is a matrix, then each row is a channel. For <code>plot</code> function, <code>x</code> is the instance returned by <code>pwelch</code> function.
<code>fs</code>	sample rate, average number of time points per second
<code>window</code>	window length in time points, default size is 64
<code>noverlap</code>	overlap between two adjacent windows, measured in time points; default is half of the window
<code>nfft</code>	number of points in window function; default is automatically determined from input data and window, scaled up to the nearest power of 2
<code>window_family</code>	function generator for generating filter windows, default is <code>hamming</code> . This can be any window function listed in the filter window family, or any window generator function from package <code>gsignal</code> . Default is <code>hamming</code> . For 'iEEG' users, both <code>hamming</code> and <code>blackmanharris</code> are offered by 'EEG-lab'; while <code>blackmanharris</code> offers better attenuation than Hamming windows, it also has lower spectral resolution. <code>hamming</code> 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 <code>hanning</code> window)
<code>col</code> , <code>xlim</code> , <code>ylim</code> , <code>main</code> , <code>type</code> , <code>cex</code> , <code>las</code> , <code>xlab</code> , <code>ylab</code> , <code>lty</code> , <code>lwd</code> , <code>xaxs</code> , <code>yaxs</code> , <code>mar</code> , <code>mgp</code> , <code>tck</code>	parameters passed to <code>plot.default</code>
<code>plot</code>	integer, whether to plot the result or not; choices are 0, no plot; 1 plot on a new canvas; 2 add to existing canvas
<code>log</code>	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 "".
<code>...</code>	will be passed to <code>plot.pwelch</code> or ignored
<code>se</code>	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)
<code>xticks</code>	ticks to show on frequency axis
<code>add</code>	logical, whether the plot should be added to existing canvas
<code>col.se</code> , <code>alpha.se</code>	controls the color and opacity of the standard error
<code>xline</code> , <code>yline</code>	controls how close the axis labels to the corresponding axes
<code>grid</code>	whether to draw rectangular grid lines to the plot; only respected when <code>add=FALSE</code> ; default is true
<code>margin</code>	the margin in which <code>pwelch</code> 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 concatenates the followings
# 10000000 (0x80), 00000000 (0x00), 01111111 (0x7f), 10000000 (0x80),
# 11111111 (0xFF), 01010000 (0x50), 01111111 (0x7f), 00000000 (0x00)

if(.Platform$endian == "little") {
  bitstring <- paste0(
    "00000000011111110101000011111111",
    "10000000011111110000000010000000"
  )
} else {
  bitstring <- paste0(
    "000000001000000001111111000000001",
    "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	<i>Imaging registration using 'NiftyReg'</i>
-----------------	--

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)

```

register_volume3d	<i>Native 3D volume registration ('rigid', 'affine', or 'SyN')</i>
-------------------	--

Description

Self-contained image registration for 3D volumes, implemented purely in **RcppEigen** (no other external registration library). It mirrors the core behavior of **'ANTs'** `antsRegistration`: a multi-resolution, physical-shift scaled gradient-descent optimizer driving a similarity metric, working entirely in the anatomical RAS (right-anterior-superior) space. Each volume carries its own `vox2ras` (0-indexed voxel index to anatomical RAS) 4×4 transform, so volumes with different sampling, orientation, or field of view are aligned correctly.

Usage

```

register_volume3d(
  source,
  target,
  source_vox2ras = NULL,
  target_vox2ras = NULL,
  source_mask = NULL,
  target_mask = NULL,
  source_points = NULL,
  target_points = NULL,
  points_weight = 0.5,
  weights = NULL,
  type = c("rigid", "affine", "syn", "syn_only"),
  metric = "mattes",
  shrink_factors = c(4, 2, 1),

```

```

smoothing_sigmas = c(2, 1, 0),
iterations = c(1000, 500, 250),
sampling_rate = 0.2,
interpolation = "trilinear",
number_of_bins = 32L,
seed = 1L,
init_transform = NULL,
syn_iterations = c(40, 20, 0),
syn_sigma = 3,
verbose = TRUE
)

```

Arguments

- | | |
|--------------------------------|---|
| source | the moving volume to be aligned, a 3D array (for example a 'CT'); the result transform maps target into this image's space. May also be a list of already co-registered 3D arrays (multiple modalities such as 'T1', 'T2', an atlas constraint); all source channels must share the same dimensions and vox2ras |
| target | the fixed/reference volume to align to, a 3D array (for example a 'MRI'), or a list of co-registered arrays matching source channel-for-channel |
| source_vox2ras, target_vox2ras | 4x4 (or 3x4) matrices mapping the 0-indexed voxel coordinate (column-row-slice, 'C'-style starting from 0, complying with 'NIfTI') to the RAS coordinate system; if NULL, the function looks for a "vox2ras" attribute on the array |
| source_mask, target_mask | optional 3D mask arrays restricting where the metric is evaluated; default NULL (no mask, evaluate everywhere). A target_mask (on the target/fixed grid, same dimensions as target) limits which target voxels drive the registration; a source_mask (on the source/moving grid) drops samples that map outside it. Non-zero voxels are included. Besides focusing the alignment on a region of interest, masks speed things up by skipping background: the linear stage samples only inside the mask, and the deformable stage skips warping voxels outside the (dilated) mask. One mask per grid, shared across channels |
| source_points, target_points | optional N x 3 matrices of corresponding landmark coordinates that add a surface/landmark term to the deformable ("syn"/"syn_only") stage; default NULL (no term). Row i of target_points (in the target/fixed RAS) and row i of source_points (in the source/moving RAS) must be the <i>same</i> anatomical location, for example corresponding cortical-surface vertices from a 'FreeSurfer' spherical registration. The term pulls the warp of each target point onto its source correspondent, recovering cortical folding (gyrification) that the intensity metric blurs over, while the image metric still drives deep-brain subcortical structures. Must be supplied together with equal row counts. Points must be in the same RAS frame as source_vox2ras/target_vox2ras (note 'FreeSurfer' surfaces use surface/TKR RAS, which differs from scanner RAS by c_ras) |
| points_weight | relative weight of the landmark term against the image metric in the deformable stage; default 0.5. Larger values follow the landmarks more closely. The sparse |

	landmark force is attenuated by <code>syn_sigma</code> smoothing, so this typically needs tuning for a given point count and spacing
<code>weights</code>	optional numeric weights, one per source/target pair, controlling each channel's contribution to the deformable cost; default is equal weighting. Weights are normalized internally to sum to 1. Only the deformable (" <code>syn</code> " / " <code>syn_only</code> ") stage is multivariate; the linear stage always uses the first (primary) pair
<code>type</code>	type of transform to estimate; one of ' <code>rigid</code> ' (6 degrees of freedom), ' <code>affine</code> ' (12), ' <code>syn</code> ' (affine followed by a SDR - symmetric diffeomorphic deformation), or ' <code>syn_only</code> ' (deformable stage only — no affine is estimated; <code>init_transform</code> is used directly as the starting affine, useful when you already have a good linear alignment)
<code>metric</code>	similarity metric: " <code>mattes</code> " (Mattes mutual information, the default, best for cross-modal such as ' <code>CT</code> ' - ' <code>MRI</code> '), " <code>cc</code> " (normalized cross-correlation, for same-modality), or " <code>meansquares</code> " (mean squared intensity difference). With multiple channels, supply either a single metric (used for every pair) or a vector with one metric per source/target pair
<code>shrink_factors</code>	integer down-sampling factors, one per resolution level (coarsest first); default <code>c(4, 2, 1)</code>
<code>smoothing_sigmas</code>	Gaussian smoothing applied at each level, in voxels, same length as <code>shrink_factors</code> ; default <code>c(2, 1, 0)</code>
<code>iterations</code>	maximum optimizer iterations per level; default <code>c(1000, 500, 250)</code>
<code>sampling_rate</code>	fraction of fixed voxels sampled to evaluate the metric (speeds up large volumes); default 0.2
<code>interpolation</code>	output interpolation used when warping each modality onto the target grid: ' <code>trilinear</code> ' (default), ' <code>nearest</code> ' (keeps label or segmentation values intact), or " <code>bspline</code> " (cubic Catmull-Rom, higher quality, slower). Like <code>metric</code> , supply a single value (applied to every channel) or one per source/target pair. This affects only the returned warped images, never the optimization's internal sampling (always ' <code>trilinear</code> ', so every channel still produces smooth gradients)
<code>number_of_bins</code>	number of histogram bins for the " <code>mattes</code> " metric; default 32
<code>seed</code>	random seed for the voxel sampler, for reproducibility
<code>init_transform</code>	optional 4x4 initial RAS-to-RAS transform (fixed to moving) to start from
<code>syn_iterations</code> , <code>syn_sigma</code>	deformable stage controls (only used when <code>type = "syn"</code> or " <code>syn_only</code> "): per-level iteration counts and the Gaussian regularization sigma (in voxels) applied to the update field. Both are recycled to the number of levels, so <code>syn_sigma</code> may be a vector to vary the regularization per stage (e.g. <code>c(3, 3, 1)</code> to relax it at the finest level for sharper detail)
<code>verbose</code>	logical; if TRUE (default) print per-level and per-iteration progress to the console, including the current stage, shrink factor, smoothing sigma, cost metric, and step size (linear) or maximum displacement (deformable); useful to monitor convergence on large volumes

Value

A list with:

transform the estimated 4x4 RAS-to-RAS linear transform mapping target (fixed) coordinates to source (moving) coordinates

image the (primary) source resampled onto the target grid

images a list with every source channel resampled onto the target grid using its own interpolation; image is the first element. For single-image input this is a length-1 list

forward_field,inverse_field (only for "syn") the deformation fields

metric_trace the metric value across optimizer iterations

type,metric echoes of the inputs

See Also

[apply_transform3d](#), [resample_3d_volume](#)

Examples

```
# --- synthetic same-modality example -----
nd <- c(50, 50, 50)
vox2ras <- diag(4); vox2ras[1:3, 4] <- -25
blob <- function(cx, cy, cz, s = 6) {
  g <- expand.grid(x = 0:(nd[1]-1), y = 0:(nd[2]-1), z = 0:(nd[3]-1))
  array(exp(-((g$x-cx)^2 + (g$y-cy)^2 + (g$z-cz)^2) / (2*s^2)), nd)
}
target <- blob(25, 25, 25)
source <- blob(28, 23, 26)           # shifted by a known (3, -2, 1) mm

res <- register_volume3d(
  source, target,
  source_vox2ras = vox2ras, target_vox2ras = vox2ras,
  type = "rigid", metric = "cc"
)
res$transform[1:3, 4]               # ~ c(3, -2, 1)

# --- multimodal registration (several co-registered channels) -----
# Two aligned modalities (e.g. a T1 and a T2) jointly drive the deformable
# stage. Each pair may use its own metric, and `weights` set their relative
# influence (normalized internally to sum to 1). All source channels must
# share a grid; likewise all target channels.
t1_target <- blob(25, 25, 25, s = 6)
t2_target <- blob(25, 25, 25, s = 9) # same anatomy, different contrast
t1_source <- blob(27, 24, 26, s = 6) # both channels share the same warp
t2_source <- blob(27, 24, 26, s = 9)

res_mm <- register_volume3d(
  source = list(t1_source, t2_source),
  target = list(t1_target, t2_target),
```

```

source_vox2ras = vox2ras, target_vox2ras = vox2ras,
weights = c(2, 1),           # T1 counts twice as much as T2
metric = c("mattes", "cc"), # one metric per channel
type = "syn"
)
res_mm$transform[1:3, 4]

```

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

Description

Low-level implementation to sample a '3D' volume into a given orientation and shape using nearest-neighbor, trilinear, or cubic B-spline interpolation.

Usage

```

resample_3d_volume(
  x,
  new_dim,
  vox2ras_old,
  vox2ras_new = vox2ras_old,
  na_fill = NA,
  interpolation = c("nearest", "trilinear", "bspline")
)

```

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
interpolation	interpolation method: 'nearest' (default), 'trilinear', or 'bspline' (cubic Catmull-Rom). Trilinear and B-spline are only applied when x is a double volume; other storage types silently fall back to nearest-neighbor.

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

<code>FUN</code>	'rgl' function name
<code>...</code>	passed to 'rgl' function
<code>expr</code>	expression within which 'rgl' functions are called
<code>quoted</code>	whether <code>expr</code> is quoted
<code>env</code>	environment in which <code>expr</code> is evaluated
<code>x</code>	triangular 'mesh3d' object
<code>length, lwd, col</code>	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)

  })
}
```

save_registration	<i>Save or load a registration result in 'ANTS'-compatible files</i>
-------------------	--

Description

`save_registration` writes a `register_volume3d` result to disk as 'ANTS'-style files: an 'ITK' affine `.mat`, the forward/inverse warp 'NIfTI'(s) when present, and a small 'DCF' manifest (the same key-value format as 'DESCRIPTION') recording what each file is plus the registration parameters. `load_registration` reads any of those back: a `.mat` returns a 4×4 RAS matrix, a warp `.nii/.nii.gz` returns the field, and a `.dcf` manifest reassembles the full object.

Usage

```
save_registration(x, path, ...)

## S3 method for class 'ravetools_register_volume3d'
save_registration(x, path, prefix = "registration", compress = TRUE, ...)

## Default S3 method:
save_registration(x, path, prefix = "registration", ...)

load_registration(file, recover_affine_from_header = FALSE)
```

Arguments

- x a ravetools_register_volume3d object (or, for the default method, a 4×4 matrix)
- path output directory, or the manifest (.dcm) path
- ... passed to methods
- prefix file-name prefix; defaults to "registration"
- compress logical; write warp fields as .nii.gz (default) or .nii
- file a .mat, .nii/.nii.gz, or .dcm path
- recover_affine_from_header logical; only used as a last resort when loading. If TRUE and the affine .mat is missing (or a warp file is loaded directly), the affine is recovered from the warp header's hidden encoding. Off by default because that encoding is reduced-precision

Value

save_registration returns the manifest path invisibly. load_registration returns a 4×4 matrix, a field array (with a "vox2ras" attribute), or a ravetools_register_volume3d object, depending on the file type.

See Also

[register_volume3d](#), [write_ants_transform](#), [write_ants_warp](#)

shift_array	<i>Shift array by index</i>
-------------	-----------------------------

Description

Re-arrange arrays in parallel

Usage

```
shift_array(x, along_margin, unit_margin, shift_amount)
```

Arguments

<code>x</code>	array, must have at least matrix
<code>along_margin</code>	which index is to be shifted
<code>unit_margin</code>	which dimension decides <code>shift_amount</code>
<code>shift_amount</code>	shift amount along <code>along_margin</code>

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 and y are the same
z - y
```

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

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

```
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

<code>pulse_info</code>	a list containing number of pulses <code>n_pulse</code> , onset index (<code>onset_index</code> , first time-point is 1), offset index (<code>offset_index</code>); this can be generated by <code>stimpulse_find</code> ; see 'Examples' below
<code>expand_timepoints</code>	point offsets allowed to align the pulses
<code>center</code>	whether to center the pulses by median; default is true
<code>max_offset</code>	maximum (edge) offsets in seconds to interpolate the pulses; default is <code>-0.0002</code> seconds before stimulation onset and <code>0.0005</code> 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	<i>Sample stimulation recording</i>
--------------------	-------------------------------------

Description

Sample stimulation recording

Usage

```
stimulation_signal
```

Format

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

```
vcg_average_edge_length
```

Compute the average edge length of a triangular mesh

Description

Computes the average length of all face edges (each edge is counted once per incident face, so edges shared by two faces are counted twice). Useful as a scale-aware reference length, e.g. to derive a vertex-merge tolerance such as the one used internally by [vcg_fix_defects](#).

Usage

```
vcg_average_edge_length(mesh)
```

Arguments

mesh	triangular mesh of class 'mesh3d'.
------	------------------------------------

Value

A single numeric value: the average edge length, in mesh units.

Examples

```
if (is_not_cran()) {
  sphere <- vcg_sphere()
  vcg_average_edge_length(sphere)
}
```

vcg_count_edge_defects*Count boundary and non-manifold edges of a triangular mesh*

Description

Detects topology defects that prevent a mesh from being a closed, manifold, genus-0 surface, a hard precondition of algorithms such as [mris_inflate](#). An edge is a *boundary* edge when it is referenced by exactly one face (i.e. it bounds a hole), and *non-manifold* when it is referenced by more than two faces.

Usage

```
vcg_count_edge_defects(mesh)
```

Arguments

mesh triangular mesh of class 'mesh3d'.

Value

A named list with elements `boundary_edges` (number of boundary edges), `nonmanifold_edges` (number of non-manifold edges), and `is_closed_manifold` (TRUE when both counts are zero, i.e. the mesh is closed and manifold and ready for [mris_inflate](#)).

Examples

```
if (is_not_cran()) {  
  
  sphere <- vcg_sphere()  
  vcg_count_edge_defects(sphere)  
  
  defective <- vcg_isosurface(left_hippocampus_mask)  
  vcg_count_edge_defects(defective)  
  
}
```

vcg_detect_collision *Detect collisions between two geometries*

Description

Reports, for every element of *y*, whether it comes within radius of *x*, together with the exact minimum distance. Each side may be a point cloud, a set of connected line segments (for example diffusion streamlines or an electrode shaft), or a triangular mesh, so the function covers all nine pairings with one call.

The test is exact: it produces neither false negatives nor false positives, and distance is the true minimum distance rather than a vertex-sampled approximation.

Usage

```
vcg_detect_collision(
  x,
  y,
  mode_x = c("auto", "points", "segments", "mesh"),
  mode_y = c("auto", "points", "segments", "mesh"),
  radius = 0,
  test_level = c("element", "unit", "whole"),
  include_interior = FALSE
)
```

Arguments

<i>x</i>	the geometry that gets indexed; a matrix with <i>n</i> rows and 2 or 3 columns, or a surface (see ensure_mesh3d)
<i>y</i>	the geometry that gets queried; results are aligned to this side. Same accepted types as <i>x</i>
<i>mode_x</i> , <i>mode_y</i>	how to interpret <i>x</i> and <i>y</i> : 'points' treats each row as an isolated point, 'segments' chains consecutive rows together (<i>p1</i> - <i>p2</i> , <i>p2</i> - <i>p3</i> , ...), and 'mesh' uses the triangular faces. The default 'auto' resolves to 'mesh' for a surface that has faces and 'points' otherwise, so a matrix of connected line segments must ask for 'segments' explicitly
<i>radius</i>	distance tolerance; a collision is reported when the minimum distance is at most <i>radius</i> . Defaults to 0, meaning literal contact; a positive tolerance is usually what is wanted
<i>test_level</i>	how exhaustively <i>y</i> is scanned. Answers are always reported per <i>unit</i> of <i>y</i> : a point when <i>mode_y</i> = 'points', a chain when 'segments', a face when 'mesh'. 'element' (the default) measures every element of every unit and reports the closest one; 'unit' stops at the first hit inside each unit and reports that one; 'whole' stops at the first hit anywhere and answers only whether the two geometries collide. Because a point and a face each <i>are</i> their own unit, 'element' and 'unit' differ only when <i>mode_y</i> = 'segments'
<i>include_interior</i>	whether geometry lying strictly inside a closed <i>x</i> surface counts as a collision even when it never comes within <i>radius</i> of the surface itself; requires <i>mode_x</i> = 'mesh'. Default is FALSE

Details

x is indexed once into a uniform spatial grid and y is streamed against it, so put the larger or repeatedly-reused geometry in x and the geometry you want per-unit answers about in y .

Several chains share one matrix and are delimited by rows of NA; any row that is not fully finite is treated as a separator. This lets a whole bundle of streamlines be passed as a single matrix:

```
rbind(
  c(0, 0, 0), c(1, 0, 0), c(2, 0, 0), # first streamline
  c(NA, NA, NA), # separator
  c(0, 5, 0), c(1, 5, 0) # second streamline
)
```

Those separators define the units that results are reported against: with `mode_y = 'segments'` the matrix above is two units, so `hit_unit` has two entries whatever the `test_level`. With `mode_y = 'points'` every row is its own unit, separators included, so `hit_unit` stays aligned with the rows of y and reports NA at each separator.

Separators split the matrix into slots, and *every* slot is a chain, empty ones included: two adjacent separators enclose an empty chain, which reports NA. The two ends are deliberately not symmetric. A separator in the first row opens an empty leading chain, but a separator in the last row closes the chain before it rather than opening a trailing empty one:

matrix	chains
A	1: [A]
A NA B	2: [A] [B]
A NA NA B	3: [A] [] [B]
NA A	2: [] [A]
A NA	1: [A]

That rule exists so a bundle assembled by appending a separator to every chain gives exactly one unit per chain, whichever of them happen to be empty:

```
do.call("rbind", lapply(tracts, function(line) rbind(line, NA)))
```

Since `mode_x` plays no part in that grouping, the index of a segment is always counted from the start of its own chain rather than from the top of the matrix. The `unit` column gives the chain.

Value

A list. `collide` is always present, so a plain yes-or-no question can be asked at any `test_level`:

`collide` a length-one logical: whether x and y collide at all. FALSE for empty input

`hit_unit` logical, one entry per unit of y , omitted when `test_level = 'whole'`. NA marks a unit with nothing to test, and is deliberately not FALSE; see *Why hit_unit reports NA* below

`representation` a data frame with one row per *hit* unit, so `nrow` equals `sum(hit_unit, na.rm = TRUE)` and a collision-free query gives zero rows. When `test_level = 'whole'` it holds at most one row

summary what the query actually was, so a stored result can be read back without the call that made it

summary holds radius, test_level, include_interior, and one entry per side. Each side reports its mode *as resolved* (never 'auto'), the matching unit_type of 'point', 'line', or 'face', and n_units: one per row for 'points' with separator rows included, one per chain for 'segments', one per face for 'mesh'. So summary\$y\$n_units always equals length(hit_unit), and remains the only record of the size of y when test_level = 'whole' drops that vector.

The columns of representation name the collision on both sides:

unit 1-based unit of y: the row itself when mode_y = 'points', the position of the chain when 'segments', the face column when 'mesh'

index 1-based element *within* that unit: always 1 for a point, the position of the segment counted from the start of its own chain for 'segments', and the face vertex nearest x (1, 2, or 3) for 'mesh'

distance the exact distance from that element to x. Distances beyond radius are never computed

x_index 1-based element of x that was hit: a row when mode_x is 'points', the row where the segment starts when 'segments', or a face column when 'mesh'. NA for a hit found by include_interior rather than by proximity

Why hit_unit reports NA

NA means *there was no geometry here to test*. FALSE means *geometry was here and it missed*. Three situations produce the first:

mode_y	the unit reports NA when
'points'	the row is a separator
'segments'	the chain is empty (two adjacent separators)
'segments'	the chain holds one vertex, so carries no segment

mode_y = 'mesh' therefore never reports NA: a face is always testable. (The scan does skip faces flagged as deleted, but a surface is rebuilt from its vertex and face matrices on every call, so no face arrives flagged.)

An empty chain cannot collide with anything, so reporting FALSE would not be wrong in the set-theoretic sense. It is avoided for two reasons. In 'points' mode hit_unit is aligned to the rows of y so that it can be indexed straight back into the input matrix, which means separator rows occupy slots; calling those FALSE asserts something about a point that does not exist. And sum(is.na(hit_unit)) counts the degenerate units that were passed in – empty or single-vertex streamlines in a tract file, deleted faces in a surface – which is usually a data-quality signal worth seeing rather than silently folding into "missed". Once flattened to FALSE that count cannot be recovered.

Most uses never have to deal with it. which already drops NA, so which(hit_unit) gives the colliding units directly, and representation\$unit is that same vector without any coercion. NA only intrudes on sum / any / if without na.rm, and on logical subsetting such as tracts[hit_unit], which yields NULL entries. Where a plain mask really is wanted:

```
mask <- res$hit_unit %in% TRUE      # NA becomes FALSE
```

test_level = 'whole' reports *a* colliding unit, not necessarily the first one in order: the scan stops as soon as any thread finds a collision. collide itself is deterministic, as are the two lower levels, which measure each unit in row order.

Both geometries must already share one coordinate space. Surface coordinates, volume IJK indices, and scanner coordinates are all different spaces, and no transform is applied here.

include_interior relies on ray casting and therefore needs *x* to be watertight with coherently oriented faces; see [vcg_fix_defects](#) to repair a surface that is not.

Multi-threading follows [ravetools_threads](#). The interior test always runs single-threaded.

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)

# A spherical region of interest
roi <- vcg_sphere()

# Two streamlines in one matrix, separated by an NA row: the first passes
# through the sphere, the second stays well clear of it
streamlines <- rbind(
  cbind(seq(-3, 3, by = 0.5), 0, 0),
  c(NA, NA, NA),
  cbind(seq(-3, 3, by = 0.5), 5, 0)
)

result <- vcg_detect_collision(
  x = roi,
  y = streamlines,
  mode_y = "segments",
  radius = 0.1
)
```

```

# Which streamlines touch the sphere, and where does each come closest?
result$hit_unit
result$representation

# What was actually asked, including what `mode = "auto"` resolved to
result$summary

# A bundle assembled by appending a separator to every streamline gives one
# unit per streamline, even where a streamline is empty: here the middle one
# contributes only its own separator, and reports NA rather than vanishing
tracts <- list(
  cbind(seq(-3, 3, by = 0.5), 0, 0),
  matrix(numeric(0), ncol = 3),
  cbind(seq(-3, 3, by = 0.5), 5, 0)
)
bundle <- do.call("rbind", lapply(tracts, function(line) rbind(line, NA)))
vcg_detect_collision(roi, bundle, mode_y = "segments",
  radius = 0.1)$hit_unit

# The same question, giving up on a streamline once it is known to collide.
# `hit_unit` is unchanged; `index` names the first colliding segment instead
# of the closest one, which here happens to be the same segment
vcg_detect_collision(roi, streamlines, mode_y = "segments", radius = 0.1,
  test_level = "unit")$representation

# Does the bundle touch the sphere at all?
vcg_detect_collision(roi, streamlines, mode_y = "segments", radius = 0.1,
  test_level = "whole")$collide

# A point cloud against the same region, asking only for proximity
points <- rbind(c(0, 0, 0), c(1.05, 0, 0), c(10, 10, 10))
vcg_detect_collision(roi, points, radius = 0.1)$hit_unit

# The centre of the sphere is far from its surface, so it only counts as a
# collision when the interior is included
vcg_detect_collision(roi, rbind(c(0, 0, 0)), radius = 0.1)$collide
vcg_detect_collision(roi, rbind(c(0, 0, 0)), radius = 0.1,
  include_interior = TRUE)$collide

```

vcg_fix_defects

Detect and repair defects in a triangular surface mesh

Description

Repairs common defects that prevent a mesh from being a closed, manifold, genus-0 surface - a hard precondition of algorithms such as [mris_inflate](#). Typical sources of such defects are surfaces extracted from volumes via marching-cubes-style algorithms (e.g. [vcg_isosurface](#)), which can leave behind small "cracks": isolated boundary-edge loops bounding tiny holes that are not closed by simple vertex-welding.

Usage

```
vcg_fix_defects(
  mesh,
  merge_tolerance = NA,
  max_hole_size = 100L,
  verbose = FALSE
)
```

Arguments

mesh	triangular mesh of class 'mesh3d'.
merge_tolerance	distance (in mesh units) below which vertices are welded together; default is NA, in which case the tolerance is derived automatically as $1e-4$ times the mesh's average edge length.
max_hole_size	maximum number of boundary edges of a hole that will be triangulated (ear-cutting fill); holes larger than this threshold are left untouched (and will be reported as remaining boundary edges in <code>attr(..., "info")</code>). Default is 100.
verbose	whether to print a short before/after diagnostic report; default is FALSE.

Details

The repair pipeline applies, in order:

1. Remove degenerate and duplicate faces.
2. Weld near-coincident vertices (closes cracks caused by duplicated vertices), using `merge_tolerance` or, by default, a distance derived from the mesh's average edge length.
3. Triangulate ("ear-cut fill") any remaining small boundary loops – i.e. *isolated edges* / genuine small holes that welding alone cannot close, up to `max_hole_size` edges.
4. Remove unreferenced vertices.
5. Re-orient all faces coherently (consistent winding order), and, if the result is a single watertight component, flip normals to point outward (this last step assumes the geometry is meant to be watertight).

Value

A repaired triangular mesh of class 'mesh3d', with an additional attribute "info", a named list reporting what was found and changed: `boundary_edges_before/after`, `nonmanifold_edges_before/after`, `vertices_merged`, `merge_tolerance`, `holes_filled`, `is_oriented`, `is_orientable`, `normals_flipped_outward`, and `is_closed_manifold` (TRUE when the repaired mesh is closed and manifold, i.e. ready for [mris_inflate](#)).

Examples

```
if (is_not_cran()) {
  mesh <- vcg_isosurface(left_hippocampus_mask)
```

```

    repaired <- vcg_fix_defects(mesh, verbose = TRUE)

    attr(repaired, "info")$is_closed_manifold

    # repaired mesh can now be inflated
    inflated <- mris_inflate(repaired, scale_brain = FALSE)

  }

```

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	<i>Find nearest k points</i>
--------------------	------------------------------

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_max_edge_length	<i>Maximum edge length of a triangular mesh</i>
---------------------	---

Description

Returns the length of the longest edge in the mesh.

Usage

```
vcg_max_edge_length(mesh)
```

Arguments

mesh	triangular mesh of class 'mesh3d'.
------	------------------------------------

Value

A single numeric value: the maximum edge length, in mesh units.

See Also

[vcg_average_edge_length](#), [vcg_subdivide_max_edge_length](#)

Examples

```
if (is_not_cran()) {  
  
  sphere <- vcg_sphere()  
  vcg_max_edge_length(sphere)  
  
}
```

vcg_mesh_patch

*Split a mesh into two patches along a geodesic boundary***Description**

Connects a set of surface waypoints with geodesic paths to form a closed boundary loop, then splits the mesh into the two regions created by that loop.

Consecutive waypoints are joined by the geodesic (Dijkstra) shortest path along the mesh surface; the last waypoint connects back to the first.

Usage

```
vcg_mesh_patch(mesh, waypoints, seed_vertex = NULL, max_edge_length = NA)
```

Arguments

mesh	triangular mesh of class 'mesh3d'.
waypoints	numeric matrix with exactly 3 columns (x, y, z) and at least 3 rows; each row is a 3-D coordinate defining a boundary corner. After any mesh subdivision, each coordinate is snapped to the nearest vertex in the (possibly refined) mesh. Consecutive snapped vertices are joined by geodesic paths to form the closed boundary loop.
seed_vertex	integer (optional, 1-based). A vertex known to be inside the desired first patch. When NULL (default) the smaller of the two regions is returned first.
max_edge_length	numeric (optional). When positive and finite, the mesh is refined before patching so that no edge exceeds this length. Global edge subdivision (vcg_subdivision) is applied repeatedly until the average edge length falls below the threshold (fast), then vcg_subdivide_max_edge_length handles any remaining outlier edges. Waypoint coordinates are snapped to vertices after refinement, so finer vertices improve boundary accuracy. Default NA (no refinement).

Value

A length-2 list of mesh3d objects. Each contains:

`$orig_vertex` 1-based integer vector: new vertex index `i` corresponds to column `orig_vertex[i]` of the original mesh\$vb.

The first element is the patch whose centroid is closest to the mean waypoint position; the second is the complementary connected patch. On a multi-manifold mesh, disconnected components not adjacent to the boundary loop appear in neither patch. When the boundary loop does not divide the mesh (degenerate waypoints), the second element is NULL.

Note

All waypoints must lie on the same connected component of the mesh. The mesh should be manifold; run [vcg_fix_defects](#) first if needed.

See Also

[dijkstras_surface_distance](#), [surface_path](#), [vcg_fix_defects](#)

Examples

```
mesh <- vcg_sphere()
mesh <- vcg_uniform_remesh(mesh)

waypoints <- diag(1, 3)
patches <- vcg_mesh_patch(mesh, waypoints)

plot_mesh_polygon(
  patches,
  col = list("red", 'gray'),
  alpha = list(1, 0.5),
  eye = c(10, 10, 10)
)
```

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

<code>x</code>	surface mesh
<code>ray_origin</code>	a matrix with 3 rows or a vector of length 3, the positions of ray origin
<code>ray_direction</code>	a matrix with 3 rows or a vector of length 3, the direction of the ray, will be normalized to length 1
<code>max_distance</code>	positive maximum distance to cast the normalized ray; default is infinity. Any invalid distances (negative, zero, or NA) will be interpreted as unset.
<code>both_sides</code>	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. Vertices that belong to no face carry no connectivity, so they are excluded from the computation and returned at their input positions with zero normal vectors; the remaining vertices are unaffected by their presence.

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 <code>vcg_smooth_implicit</code> , the amount of smoothness, useful only if <code>use_mass_matrix</code> is TRUE; default is 0.2. In <code>vcg_smooth_explicit</code> , 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 <code>use_cot_weight</code> 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_subdivide_max_edge_length
```

Selectively subdivide mesh edges that exceed a length threshold

Description

Up-sample a triangular mesh by iteratively splitting only edges longer than `max_edge_len`. Each long edge is split at its midpoint; the new vertex is connected to the opposite corner of every adjacent face. Iteration stops when no edge exceeds the threshold or `max_iter` passes are exhausted.

This is far cheaper than `vcg_subdivision` when most edges are already short and only a small fraction need splitting.

Usage

```
vcg_subdivide_max_edge_length(mesh, max_edge_len, max_iter = NULL)
```

Arguments

<code>mesh</code>	triangular mesh of class 'mesh3d'.
<code>max_edge_len</code>	maximum allowed edge length (same units as mesh coordinates).
<code>max_iter</code>	maximum number of refinement passes. When NULL (default), derived automatically from the current maximum edge length: $\text{ceiling}(\log_2(\text{current_max} / \text{max_edge_len})) + 1$, the minimum number of bisections needed in the worst case. The loop also terminates early once no edge exceeds the threshold.

Value

An object of class "mesh3d" with all edges at most `max_edge_len` long (provided `max_iter` was sufficient).

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.

Note

The mesh must be manifold. Run `vcg_fix_defects` first if the mesh has boundary edges or non-manifold vertices.

See Also

`vcg_max_edge_length`, `vcg_subdivision`

Examples

```
if (is_not_cran()) {
  sphere <- vcg_sphere()
  cur_max <- vcg_max_edge_length(sphere)
  sphere2 <- vcg_subdivide_max_edge_length(sphere, max_edge_len = cur_max * 0.4)
  vcg_max_edge_length(sphere2) # should be <= cur_max * 0.4
}
```

<code>vcg_subdivision</code>	<i>Sub-divide (up-sample) a triangular mesh</i>
------------------------------	---

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

<code>mesh</code>	triangular mesh stored as object of class 'mesh3d'.
<code>method</code>	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	<i>Sample a surface mesh uniformly</i>
--------------------	--

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

<code>freqs</code>	frequency in which data will be projected on
<code>srate</code>	sample rate, number of time points per second
<code>wave_num</code>	desired number of cycles in wavelet kernels to balance the precision in time and amplitude (control the smoothness); positive integers are strongly suggested
<code>data</code>	numerical vector such as analog voltage signals
<code>precision</code>	the precision of computation; choices are 'float' (default) and 'double'.
<code>trend</code>	choices are 'constant': center the signal at zero; 'linear': remove the linear trend; 'none' do nothing
<code>signature</code>	signature to calculate kernel path to save, internally used
<code>segment_length</code>	optional positive integer; when provided, long signals are processed in overlapping segments of this length (in samples) using batched <code>mvfft_w_c2c</code> 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 <code>segment_length >= length(data)</code> 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(<code>max_kernel_len/2</code>) samples may differ near the global boundaries (both implementations have boundary artifacts there).
<code>...</code>	further passed to <code>detrend</code> ;
<code>frequency_range</code>	frequency range to calculate, default is 2 to 200
<code>cycle_range</code>	number of cycles corresponding to <code>frequency_range</code> . For default frequency range (2 - 200), the default <code>cycle_range</code> 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)

```

write_ants_transform *Read and write an 'ITK'/'ANTs' affine transform*

Description

write_ants_transform stores a 4×4 RAS-to-RAS affine (such as the transform returned by [register_volume3d](#)) as an 'ITK' .mat file (a 'MATLAB' level-5 binary, LPS convention) compatible with antsApplyTransforms. read_ants_transform reads such a file back into a 4×4 RAS matrix. The reader folds a non-zero ITK center of rotation into the translation, so transforms written by 'ANTs' are read correctly.

Usage

```

write_ants_transform(transform, file)

read_ants_transform(file)

```

Arguments

transform	a 4×4 (or 3×4) RAS affine
file	path to the .mat file

Value

write_ants_transform returns file invisibly; read_ants_transform returns a 4×4 RAS matrix.

See Also

[register_volume3d](#), [save_registration](#)

Examples

```
tf <- tempfile(fileext = ".mat")
m <- diag(4); m[1:3, 4] <- c(3, -2, 1)
write_ants_transform(m, tf)
read_ants_transform(tf)
```

write_ants_warp

Read and write an 'ANTs' deformation (warp) field

Description

write_ants_warp stores a dense displacement field (such as the forward_field / inverse_field from [register_volume3d](#)) as an 'ANTs' 5-D warp 'NIfTI' (intent_code 1007, LPS vectors) readable by antsApplyTransforms; read_ants_warp reads it back. Requires the suggested **freesurfer-formats** package.

The field is stored on disk with displacement vectors in LPS (the x and y components are negated relative to the RAS field) and an sform equal to vox2ras, so the file describes the fixed grid's physical space exactly as 'ANTs' expects. Optionally the RAS affine can be hidden in the header text fields (descrip/aux_file); 'ANTs' ignores those, and recovery on read is opt-in.

Usage

```
write_ants_warp(
  field,
  file,
  vox2ras = attr(field, "vox2ras"),
  affine = NULL,
  direction = c("forward", "inverse")
)

read_ants_warp(file, recover_affine = FALSE)
```

Arguments

field	a (nx, ny, nz, 3) array of RAS displacements; the vox2ras attribute is used when vox2ras is not supplied
file	path to the warp .nii / .nii.gz file
vox2ras	the fixed-grid 4×4 voxel-to-RAS matrix

<code>affine</code>	optional 4×4 RAS affine to embed (with reduced precision) in the header; NULL (default) embeds nothing
<code>direction</code>	"forward" or "inverse"; recorded as the header direction flag when affine is embedded
<code>recover_affine</code>	logical; if TRUE, attempt to decode a hidden affine from the header and attach it as the "transform" attribute (with a "direction" attribute). Default FALSE

Value

`write_ants_warp` returns file invisibly; `read_ants_warp` returns the (nx, ny, nz, 3) RAS field with a "vox2ras" attribute (and, if recovered, "transform" / "direction" attributes).

See Also

[register_volume3d](#), [save_registration](#)

Index

* datasets

- left_hippocampus_mask, 70
- stimulation_signal, 126

- apply_transform3d, 4, 117
- as_matrix4 (new_matrix4), 87
- as_quaternion (new_quaternion), 88
- as_vector3 (new_vector3), 88

- band_pass, 5
- band_pass1 (band_pass), 5
- band_pass2 (band_pass), 5
- baseline_array, 7
- blackman (filter-window), 60
- blackmanharris, 108
- blackmanharris (filter-window), 60
- blackmannuttall (filter-window), 60
- bohmanwin (filter-window), 60
- bpc, 10, 92
- butter_max_order, 14

- carla, 15
- catmull_rom_3d, 19, 93, 103
- check_filter, 21, 112
- chol, 65
- collapse, 22
- color_ramp_continuous, 24
- colorRamp, 24
- convolve, 25, 56
- convolve_image (convolve), 25
- convolve_signal (convolve), 25
- convolve_volume (convolve), 25
- cov, 53
- crp, 13, 27, 32–34, 92, 93
- crp_cluster, 11, 13, 32, 93

- decimate, 35
- design_filter, 36
- design_filter_fir, 36, 38
- design_filter_iir, 36, 37, 41

- detect_threads (parallel-options), 90
- detrend, 43, 153
- diagnose_channel, 44
- diagnose_filter, 46
- dijkstras-path, 48
- dijkstras_surface_distance, 139
- dijkstras_surface_distance
(dijkstras-path), 48

- ensure_mesh3d, 51, 58, 60, 72, 77, 79, 104, 128

- fast_cov, 53
- fast_median (fast_quantile), 54
- fast_mvmedian (fast_quantile), 54
- fast_mvquantile (fast_quantile), 54
- fast_quantile, 54
- fft, 56
- fftw-internal (fftw_r2c), 55
- fftw_c2c (fftw_r2c), 55
- fftw_c2c_2d (fftw_r2c), 55
- fftw_c2c_3d (fftw_r2c), 55
- fftw_c2r (fftw_r2c), 55
- fftw_r2c, 55
- fftw_r2c_2d (fftw_r2c), 55
- fftw_r2c_3d (fftw_r2c), 55
- fill_surface, 58, 77
- filter-window, 60
- filter_signal, 61
- filtfilt, 25, 61, 62, 112
- find_peaks, 63
- fir1, 64
- firls, 38, 39, 65
- flattpwin (filter-window), 60
- freqz2, 66

- gammatone_fast, 66
- get0, 69
- grow_volume, 68
- hamming, 108

- hamming (filter-window), 60
- hanning, 108
- hanning (filter-window), 60
- hist, 44, 45
- internal_rave_function, 69
- kaiser, 39
- left_hippocampus_mask, 70
- load_registration (save_registration), 120
- matlab_palette, 70
- mesh_from_volume, 70
- morlet_wavelet (wavelet), 152
- mriscurvature, 72
- mriscinflate, 74, 77, 78, 81, 83, 127, 132, 133
- mrismake_surfaces, 76
- mrismesh, 79
- mrismooth, 81
- mrissphere, 78, 82
- multitaper, 56, 84
- multitaper_config (multitaper), 84
- mv_pwelch (pwelch), 106
- mvfft, 56
- mvfftw_c2c, 153
- mvfftw_c2c (fftw_r2c), 55
- mvfftw_c2r (fftw_r2c), 55
- mvfftw_r2c (fftw_r2c), 55
- naive_nmf, 12, 13, 32–34, 86
- new_matrix4, 87, 88, 89
- new_quaternion, 88, 88, 89
- new_vector3, 88, 88
- niftyreg, 113
- notch_filter, 89
- par, 45, 93, 103
- parallel-options, 90
- plane_geometry, 91
- plot, 94, 103
- plot.default, 96, 99, 100, 108
- plot.ravetools-pwelch (pwelch), 106
- plot.ravetools_bpc, 92
- plot.ravetools_crp, 92
- plot.ravetools_crp_cluster, 93
- plot.ravetools_curve, 20, 93
- plot_mesh_dotcloud, 94, 99, 101
- plot_mesh_polygon, 98
- plot_signals, 102
- points, 96
- polygon, 98
- print.data.frame, 103
- print.ravetools-pwelch (pwelch), 106
- print.ravetools_curve, 20, 103
- project_plane, 104
- pwelch, 44, 45, 56, 106
- qr.solve, 65
- ravetools_threads, 131
- ravetools_threads (parallel-options), 90
- raw-to-sexp, 109
- raw_to_float (raw-to-sexp), 109
- raw_to_int16 (raw-to-sexp), 109
- raw_to_int32 (raw-to-sexp), 109
- raw_to_int64 (raw-to-sexp), 109
- raw_to_int8 (raw-to-sexp), 109
- raw_to_string (raw-to-sexp), 109
- raw_to_uint16 (raw-to-sexp), 109
- raw_to_uint32 (raw-to-sexp), 109
- raw_to_uint8 (raw-to-sexp), 109
- rawToChar, 110
- rcond_filter_ar, 112
- read_ants_transform
 - (write_ants_transform), 154
- read_ants_warp (write_ants_warp), 155
- register_volume, 113
- register_volume3d, 4, 5, 114, 120, 121, 154–156
- remez, 39
- resample_3d_volume, 117, 118
- rgl-call, 119
- rgl_call (rgl-call), 119
- rgl_plot_normals (rgl-call), 119
- rgl_view (rgl-call), 119
- save_registration, 120, 155, 156
- shift_array, 121
- stimpulse_align
 - (stimpulse_interpolate), 123
- stimpulse_extract
 - (stimpulse_interpolate), 123
- stimpulse_find (stimpulse_interpolate), 123
- stimpulse_interpolate, 123
- stimulation_signal, 126

surface_path, [139](#)
surface_path(dijkstras-path), [48](#)

vcg_average_edge_length, [126](#), [137](#)
vcg_count_edge_defects, [127](#)
vcg_detect_collision, [127](#)
vcg_fix_defects, [126](#), [131](#), [132](#), [138](#), [139](#),
[146](#)
vcg_isosurface, [74](#), [97](#), [101](#), [132](#), [134](#)
vcg_kdtree_nearest, [135](#)
vcg_max_edge_length, [137](#), [146](#)
vcg_mesh_patch, [138](#)
vcg_mesh_volume, [70](#), [139](#)
vcg_raycaster, [140](#)
vcg_smooth, [141](#)
vcg_smooth_explicit, [70](#), [71](#)
vcg_smooth_explicit(vcg_smooth), [141](#)
vcg_smooth_implicit, [70](#)
vcg_smooth_implicit(vcg_smooth), [141](#)
vcg_sphere, [98](#), [99](#), [144](#)
vcg_subdivide_max_edge_length, [137](#), [138](#),
[145](#)
vcg_subdivision, [138](#), [145](#), [146](#), [146](#)
vcg_subset_vertex, [147](#)
vcg_uniform_remesh, [70](#), [71](#), [80](#), [149](#)
vcg_update_normals, [95–97](#), [150](#)

wavelet, [152](#)
wavelet_cycles_suggest(wavelet), [152](#)
wavelet_kernels(wavelet), [152](#)
write_ants_transform, [121](#), [154](#)
write_ants_warp, [121](#), [155](#)