

# Package ‘ravetools’

November 8, 2024

**Type** Package

**Title** Signal and Image Processing Toolbox for Analyzing Intracranial Electroencephalography Data

**Version** 0.1.9

**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/>

**License** GPL (>= 2)

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**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 ( $\geq 0.1.3$ ), Rcpp, waveslim ( $\geq 1.8.2$ ), pracma, digest ( $\geq 0.6.29$ ), splines, RNiftyReg ( $\geq 2.7.1$ ), R6 ( $\geq 2.5.1$ ), gsignal ( $\geq 0.3.5$ )

**LinkingTo** Rcpp, RcppEigen

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

**LazyData** true

**NeedsCompilation** yes

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

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

**Repository** CRAN

**Date/Publication** 2024-11-08 17:40:02 UTC

## Contents

band_pass . . . . .	3
baseline_array . . . . .	6
butter_max_order . . . . .	9
check_filter . . . . .	10
collapse . . . . .	11
convolve . . . . .	12
decimate . . . . .	14
design_filter . . . . .	15
design_filter_fir . . . . .	17
design_filter_iir . . . . .	20
detrend . . . . .	22
diagnose_channel . . . . .	23
diagnose_filter . . . . .	25
dijkstras-path . . . . .	27
fast_cov . . . . .	31
fast_quantile . . . . .	32

fill_surface . . . . .	33
filter-window . . . . .	34
filter_signal . . . . .	35
filtfilt . . . . .	36
fir1 . . . . .	37
firls . . . . .	38
freqz2 . . . . .	39
grow_volume . . . . .	39
internal_rave_function . . . . .	40
interpolate_stimulation . . . . .	41
left_hippocampus_mask . . . . .	42
matlab_palette . . . . .	42
mesh_from_volume . . . . .	43
multitaper . . . . .	44
new_matrix4 . . . . .	46
new_quaternion . . . . .	47
new_vector3 . . . . .	47
notch_filter . . . . .	48
parallel-options . . . . .	49
plot_signals . . . . .	50
pwelch . . . . .	51
raw-to-sexp . . . . .	54
rcond_filter_ar . . . . .	57
register_volume . . . . .	58
rgl-call . . . . .	59
shift_array . . . . .	60
vcg_isosurface . . . . .	62
vcg_mesh_volume . . . . .	63
vcg_smooth . . . . .	63
vcg_sphere . . . . .	66
vcg_uniform_remesh . . . . .	66
vcg_update_normals . . . . .	68
wavelet . . . . .	69

**Index****72**

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)
```

---

baseline\_array

*Calculate Contrasts of Arrays in Different Methods*


---

### Description

Provides five methods to baseline an array and calculate contrast.

### Usage

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

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

### Arguments

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

## Details

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

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

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

$$baseline\_array(x, \text{along\_dim} = 3, \text{baseline\_window} = 1 : 100, \text{unit\_dims} = c(1, 2, 4))$$

$\text{along\_dim}=3$  is dimension of time, in this case, it's the third dimension of  $x$ .  $\text{baseline\_indexpoints}=1:100$ , meaning the first 100 time points are used to calculate baseline.  $\text{unit\_dims}$  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  $\text{unit\_dims}=c(1, 4)$ .

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

"percentage"

$$\frac{z - \bar{z}_0}{\bar{z}_0} \times 100\%$$

"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"

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

"sqrt\_zscore"

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

## 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_indepoints = 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 b1 = x[,baseline_window, ]
  #
  baseline_array(x, along_dim = 3,
                 baseline_indepoints = baseline_window,
                 unit_dims = c(1,2,4), method = 'percentage')
}

```



```
}  
range(f1() - f2())  
microbenchmark::microbenchmark(f1(), f2(), times = 10L)
```

---

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 <code>.Machine\$double.eps</code> .

### 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)

```

---

check_filter	<i>Check 'Arma' filter</i>
--------------	----------------------------

---

### Description

Check 'Arma' filter

### Usage

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

### Arguments

b	moving average (MA) polynomial coefficients.
a	auto-regressive (AR) polynomial coefficients.
w	normalized frequency, ranging from 0 to 1, where 1 is 'Nyquist'
r_expected	attenuation in decibel of each w
fs	sample rate, used to infer the frequencies and formatting print message, not used in calculation; leave it blank by default

### Value

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

### Examples

```

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

sample_rate <- 20
nyquist <- sample_rate / 2

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

# filter quality is poor because the AR-coefficients

```

```

# creates singular matrix with unstable inverse,
# this will cause `filtfilt` to fail
check_filter(
  b = filter$b, a = filter$a,

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

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

)

```

collapse

*Collapse array***Description**

Collapse array

**Usage**

collapse(x, keep, ...)

```

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

```

**Arguments**

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

**Value**

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

**Examples**

```

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

# Example 1
x = matrix(1:16, 4)

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

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

ravetools_threads(n_threads = -1)

# Example 3 (performance)

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

# large data big difference
x = array(rnorm(prod(300,200,105)), c(300,200,105,1))
microbenchmark::microbenchmark(
  result = collapse(x, keep = c(3,2)),
  compare = apply(x, c(3,2), mean),
  times = 1L , check = function(v){
    max(abs(range(do.call('-', v)))) < 1e-10
  })

```

**Description**

Use the 'Fast-Fourier' transform to compute the convolutions of two data with zero padding. This function is mainly designed for image convolution. For forward and backward convolution/filter, see [filtfilt](#).

**Usage**

```
convolve_signal(x, filter)
```

```
convolve_image(x, filter)
```

```
convolve_volume(x, filter)
```

**Arguments**

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

**Details**

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

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

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

**Value**

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

**Examples**

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

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

# ---- 2D convolution -----
```

```

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

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

y <- convolve_image(x, kernel)

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

```

---

decimate

*Decimate with 'FIR' or 'IIR' filter*


---

## Description

Decimate with 'FIR' or 'IIR' filter

## Usage

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

## Arguments

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

## Details

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

## Value

Decimated signal

**Examples**

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

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

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

---

design\_filter

*Design a digital filter*


---

**Description**

Provides 'FIR' and 'IIR' filter options; default is 'FIR', see also [design\\_filter\\_fir](#); for 'IIR' filters, see [design\\_filter\\_iir](#).

**Usage**

```
design_filter(
  sample_rate,
  data = NULL,
  method = c("fir_kaiser", "firls", "fir_remez", "butter", "cheby1", "cheby2", "ellip"),
  high_pass_freq = NA,
  high_pass_trans_freq = NA,
  low_pass_freq = NA,
  low_pass_trans_freq = NA,
  passband_ripple = 0.1,
  stopband_attenuation = 40,
  filter_order = NA,
  ...,
  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 <a href="#">design_filter_fir</a> or <a href="#">design_filter_iir</a>
high_pass_trans_freq, low_pass_trans_freq	transition bandwidths, see <a href="#">design_filter_fir</a> or <a href="#">design_filter_iir</a>
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; 'RAVE' may or may not adopt this suggestion depending on the data and numerical feasibility
...	passed to filter generator functions
data_size	used by 'FIR' filter design to determine maximum order, ignored in 'IIR' filters; automatically derived from data

### Value

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

### Examples

```

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

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

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

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

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

```



```

    high_pass_freq = 30
  )

  oldpar <- par(mfrow = c(2, 1),
               mar = c(3.1, 2.1, 3.1, 0.1))
  plot(t, x, type = 'l', xlab = "Time", ylab = "",
       main = "Mixture of 2, 10, and 60Hz", xlim = c(0,1))
  # lines(t, y, col = 'red')
  lines(t, y3, col = 'green')
  lines(t, y2, col = 'blue')
  lines(t, y1, col = 'red')
  legend(
    "topleft", c("Input", "Low: 3Hz", "Pass 8-12Hz", "Pass 30-80Hz"),
    col = c(par("fg"), "red", "blue", "green"), lty = 1,
    cex = 0.6
  )

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

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

```

---

design\_filter\_fir      *Design 'FIR' filter using [firls](#)*

---

## Description

Design 'FIR' filter using [firls](#)

## Usage

```

design_filter_fir(
  sample_rate,
  filter_order = NA,
  data_size = NA,
  high_pass_freq = NA,
  high_pass_trans_freq = NA,
  low_pass_freq = NA,
  low_pass_trans_freq = NA,

```

```

    stopband_attenuation = 40,
    scale = TRUE,
    method = c("kaiser", "firls", "remez")
)

```

### Arguments

sample_rate	sampling frequency
filter_order	filter order, leave NA (default) if undecided
data_size	minimum length of data to apply the filter, used to decide the maximum filter order. For 'FIR' filter, data length must be greater than 3xfilter_order
high_pass_freq	high-pass frequency; default is NA (no high-pass filter will be applied)
high_pass_trans_freq	high-pass frequency band-width; default is automatically inferred from data size. Frequency high_pass_freq - high_pass_trans_freq is the corner of the stop-band
low_pass_freq	low-pass frequency; default is NA (no low-pass filter will be applied)
low_pass_trans_freq	low-pass frequency band-width; default is automatically inferred from data size. Frequency low_pass_freq + low_pass_trans_freq is the corner of the stop-band
stopband_attenuation	allowable power attenuation (in decibel) at transition frequency; default is 40 dB.
scale	whether to scale the filter for unity gain
method	method to generate 'FIR' filter, default is using <a href="#">kaiser</a> estimate, other choices are <a href="#">firls</a> (with hamming window) and <a href="#">remez</a> design.

### Details

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

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

**Value**

'FIR' filter in 'Arma' form.

**Examples**

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

sample_rate <- 500
data_size <- 1000

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

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

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

# ---- Advanced -----

sample_rate <- 500
data_size <- 1000

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

#
print(filter)

diagnose_filter(
  filter$b, filter$a,
```

```

fs = sample_rate,
n = data_size,
cutoffs = c(-1, -40),
vlines = c(3, 8)
)

```

---

design\_filter\_iir      *Design an 'IIR' filter*

---

### Description

Design an 'IIR' filter

### Usage

```

design_filter_iir(
  method = c("butter", "cheby1", "cheby2", "ellip"),
  sample_rate,
  filter_order = NA,
  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. Notice filters with higher orders may become numerically unstable, hence this number is only a suggested number. If the filter is unstable, this function will choose a lower order; leave this input NA (default) if undecided.
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 in 'Arma' form.

### Examples

```
sample_rate <- 500

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

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

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

filter

my_diagnose(filter)

## explicit bandwidths and attenuation (sharper transition)

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

```
)  
  
filter  
  
my_diagnose(filter)  
  
# ---- cheby1 -----  
  
filter <- design_filter_iir(  
  method = "cheby1",  
  low_pass_freq = 12,  
  high_pass_freq = 8,  
  sample_rate = 500  
)  
  
my_diagnose(filter)  
  
# ---- cheby2 -----  
  
filter <- design_filter_iir(  
  method = "cheby2",  
  low_pass_freq = 12,  
  high_pass_freq = 8,  
  sample_rate = 500  
)  
  
my_diagnose(filter)  
  
# ----- ellip -----  
  
filter <- design_filter_iir(  
  method = "ellip",  
  low_pass_freq = 12,  
  high_pass_freq = 8,  
  sample_rate = 500  
)  
  
my_diagnose(filter)
```

---

detrend

*Remove the trend for one or more signals*

---

### **Description**

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

**Usage**

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

**Arguments**

**x** numerical or complex, a vector or a matrix

**trend** the trend of the signal; choices are 'constant' and 'linear'

**break\_points** integer vector, or NULL; only used when trend is 'linear' to remove piecewise linear trend; will throw warnings if trend is 'constant'

**Value**

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

**Examples**

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

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

---

diagnose_channel	<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 srate is too high; will be automatically generated if NULL
srate	sampling rate
name	name of s1, or a vector of two names of s1 and s2 if s2 is provided
try_compress	whether try to compress (decimate) s1 if srate is too high for performance concerns
max_freq	the maximum frequency to display in 'Welch Periodograms'
window, noverlap	see <a href="#">pwelch</a>
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 <a href="#">par</a>
plim	the y-axis limit to draw in 'Welch Periodograms'
nclass	number of classes to show in histogram ( <a href="#">hist</a> )
start_time	the starting time of channel (will only be used to draw signals)



boundary        a red boundary to show in channel plot; default is to be automatically determined by std

xline, yline    distance of axis labels towards ticks

**Value**

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

**Examples**

```
library(ravetools)

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

signal2 <- notch_filter(signal, 2000)

diagnose_channel(signal, signal2, srate = 2000,
  name = c("Raw", "Filtered"), cex = 1)
```

---

diagnose\_filter        *Diagnose digital filter*

---

**Description**

Generate frequency response plot with sample-data simulation

**Usage**

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

**Arguments**

b	the moving-average coefficients of an ARMA model
a	the auto-regressive coefficients of an ARMA filter; default is 1
fs	sampling frequency in Hz
n	number of points at which to evaluate the frequency response; default is 512
whole	whether to evaluate beyond Nyquist frequency; default is false
sample	sample signal of length n for simulation
vlines	additional vertical lines (frequencies) to plot
xlim	frequency limit of frequency response plot; default is "auto", can be "full" or a numeric of length 2
cutoffs	cutoff decibel powers to draw on the frequency plot, also used to calculate the frequency limit when xlim is "auto"

**Value**

Nothing

**Examples**

```

library(ravetools)

# sample rate
srate <- 500

# signal length
npts <- 1000

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

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

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

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

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

# ---- Butter filter -----

```

```

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

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

```

---

dijkstras-path      *Calculate distances along a surface*

---

### Description

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

### Usage

```

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

surface_path(x, target_node)

```

### Arguments

positions	numeric matrix with no NA values. The number of row is the total count of nodes (vertices), and the number of columns represent the node dimension. Each row represents a node.
faces	integer matrix with each row containing indices of nodes. For graphs, faces is a matrix with two columns defining the connecting edges; for '3D' mesh, faces is a three-column matrix defining the face index of mesh triangles.
start_node	integer, row index of positions on where to start calculating the distances. This integer must be 1-indexed and cannot exceed the total number of positions rows
face_index_start	integer, the start of the nodes in faces; please specify this input explicitly if the first node is not contained in faces. Default is NA (determined by the minimal number in faces). The reason to set this input is because some programs use 1 to represent the first node, some start from 0.

```

max_search_distance      numeric, maximum distance to iterate; default is NA, that is to iterate and search
                          the whole mesh
...                      reserved for backward compatibility
x                        distance calculation results returned by dijkstras_surface_distance func-
                          tion
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)

```

---

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	<i>Compute quantiles</i>
---------------	--------------------------

---

## Description

Compute quantiles

## Usage

```
fast_quantile(x, prob = 0.5, na.rm = FALSE, ...)
```

```
fast_median(x, na.rm = FALSE, ...)
```

```
fast_mvquantile(x, prob = 0.5, na.rm = FALSE, ...)
```

```
fast_mvmedian(x, na.rm = FALSE, ...)
```

## Arguments

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

## Value

fast\_quantile and fast\_median calculate univariate quantiles (single-value return); fast\_mvquantile and fast\_mvmedian calculate multivariate quantiles (for each column, result lengths equal to the number of columns).



**Examples**

```

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

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

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

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

```

---

fill\_surface

---

*Fill a volume cube based on water-tight surface*


---

**Description**

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

**Usage**

```

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

```

**Arguments**

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

**Details**

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

**Value**

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

**Author(s)**

Zhengjia Wang

**Examples**

```
# takes > 5s to run example

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

fill_surface(surface, preview = TRUE)
```

---

filter-window

*Filter window functions*

---

**Description**

Filter window functions

**Usage**

hanning(n)

hamming(n)

blackman(n)

blackmannuttall(n)

blackmanharris(n)

flattopwin(n)

bohmanwin(n)

**Arguments**

n                    number of time-points in window

**Value**

A numeric vector of window with length n

**Examples**

hanning(10)

hamming(11)

blackmanharris(21)

---

`filter_signal`*Filter one-dimensional signal*

---

**Description**

The function is written from the scratch. The result has been compared against the 'Matlab' filter function with one-dimensional real inputs. Other situations such as matrix b or multi-dimensional x are not implemented. For double filters (forward-backward), see [filtfilt](#).

**Usage**

filter\_signal(b, a, x, z)

**Arguments**

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

**Value**

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

**Examples**

```
t <- seq(0, 1, by = 0.01)
x <- sin(2 * pi * t * 2.3)
bf <- gsignal::butter(2, c(0.15, 0.3))

res <- filter_signal(bf$b, bf$a, x)
y <- res[[1]]
z <- res[[2]]

## Matlab (2022a) equivalent:
# t = [0:0.01:1];
# x = sin(2 * pi * t * 2.3);
# [b,a] = butter(2,[.15,.3]);
# [y,z] = filter(b, a, x)
```

---

 filtfilt

---

*Forward and reverse filter a one-dimensional signal*


---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

**Examples**

```
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)
```

---

 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**

<code>n</code>	filter order
<code>w</code>	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.
<code>type</code>	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 = "")
```

**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.

**Value**

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

---

freqz2 *Frequency response of digital filter*

---

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

A list of frequencies and corresponding responses in complex vector

---

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 <code>get0</code>
on_missing	default value to return of no function is found

**Value**

Function object if found, otherwise `on_missing`.



---

`interpolate_stimulation`*Find and interpolate stimulation signals*

---

**Description**

Find and interpolate stimulation signals

**Usage**

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

**Arguments**

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

**Value**

Interpolated signal with an attribute of which sample points are interpolated

**Examples**

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

```

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

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

pwelch(x0, 100, 200, 100, plot = 1, col = 'black', ylim = c(-50, 50))
pwelch(x, 100, 200, 100, plot = 2, col = 'red')
pwelch(fitted, 100, 200, 100, plot = 2, col = 'blue')

par(oldpar)

```

---

left\_hippocampus\_mask *Left 'Hippocampus' of 'N27-Collin' brain*

---

### Description

Left 'Hippocampus' of 'N27-Collin' brain

### Usage

left\_hippocampus\_mask

### Format

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

---

matlab\_palette *'Matlab' heat-map plot palette*

---

### Description

'Matlab' heat-map plot palette

### Usage

matlab\_palette()

### Value

vector of 64 colors

---

mesh_from_volume	<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 <a href="#">vcg_uniform_remesh</a>
remesh_voxel_size, remesh_multisample, remesh_automerge	see arguments in <a href="#">vcg_uniform_remesh</a>
smooth	whether to smooth the mesh via <a href="#">vcg_smooth_explicit</a>
smooth_lambda, smooth_delta, smooth_method	see <a href="#">vcg_smooth_explicit</a>

### Value

A 'mesh3d' surface if output\_format is 'rgl', or 'fs.surface' surface otherwise.

## Examples

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

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

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

---

multitaper

*Compute 'multitaper' spectral densities of time-series data*

---

## Description

Compute 'multitaper' spectral densities of time-series data

## Usage

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

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

## Arguments

data_length	length of data
fs	sampling frequency in 'Hz'

frequency_range	frequency range to look at; length of two
time_bandwidth	a number indicating time-half bandwidth product; i.e. the window duration times the half bandwidth of main lobe; default is 5
num_tapers	number of 'DPSS' tapers to use; default is NULL and will be automatically computed from $\text{floor}(2 \times \text{time\_bandwidth} - 1)$
window_params	vector of two numbers; the first number is the window size in seconds; the second number is the step size; default is $c(5, 1)$
nfft	'NFFT' size, positive; see 'Details'
detrend_opt	how you want to remove the trend from data window; options are 'linear' (default), 'constant', and 'off'
data	numerical vector, signal traces

### Details

The original source code comes from 'Prerau' Lab (see 'Github' repository 'multitaper\_toolbox' under user 'preraulab'). The results tend to agree with their 'Python' implementation with precision on the order of at  $1E-7$  with standard deviation at most  $1E-5$ . The original copy was licensed under a Creative Commons Attribution 'NC'-SA' 4.0 International License (<https://creativecommons.org/licenses/by-nc-sa/4.0/>).

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

### Value

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

### Examples

```
# 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()
)
```

---

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

---

## Description

Create a Matrix4 instance for 'Affine' transform

## Usage

```
new_matrix4()
```

```
as_matrix4(m)
```

## Arguments

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

## Value

A Matrix4 instance

## See Also

[new\\_vector3](#), [new\\_quaternion](#)

---

new_quaternion	<i>Create a Quaternion instance to store '3D' rotation</i>
----------------	--

---

**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	<i>Create a Vector3 instance to store '3D' points</i>
-------------	---

---

**Description**

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

**Usage**

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

```
as_vector3(v)
```

**Arguments**

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

**Value**

A Vector3 instance

**See Also**

[new\\_matrix4](#), [new\\_quaternion](#)

**Examples**

```
vec3 <- new_vector3(  
  x = 1:9,  
  y = 9:1,  
  z = rep(c(1,2,3), 3)  
)  
  
vec3[]  
  
# transform  
m <- new_matrix4()  
  
# rotation xy plane by 30 degrees  
m$make_rotation_z(pi / 6)  
  
vec3$apply_matrix4(m)  
  
vec3[]  
  
as_vector3(c(1,2,3))
```

---

notch\_filter

*Apply 'Notch' filter*

---

**Description**

Apply 'Notch' filter

**Usage**

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



**Arguments**

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

**Details**

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

**Value**

filtered signal in time domain (real numerical vector)

**Examples**

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

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

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

---

parallel-options      *Set or get thread options*

---

**Description**

Set or get thread options

**Usage**

```
detect_threads()

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

**Arguments**

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

**Value**

detect\_threads returns an integer of default threads that is determined by the number of CPU cores; ravetools\_threads returns nothing.

**Examples**

```
detect_threads()

ravetools_threads(n_threads = 2)
```

---

plot_signals	<i>Plot one or more signal traces in the same figure</i>
--------------	--

---

**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 <a href="#">plot</a> and <a href="#">par</a>
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)
```

---

pwelch

*Calculate 'Welch Periodogram'*

---

**Description**

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

**Usage**

```
pwelch(  
  x,  
  fs,  
  window = 64,  
  noverlap = window/2,  
  nfft = "auto",  
  window_family = hamming,  
  col = "black",  
  xlim = NULL,  
  ylim = NULL,  
  main = "Welch periodogram",  
  plot = 0,  
  log = c("xy", "", "x", "y"),  
  ...  
)  
  
## S3 method for class ``ravetools-pwelch``  
print(x, ...)  
  
## S3 method for class ``ravetools-pwelch``  
plot(  
  x,  
  log = c("xy", "x", "y", ""),  
  se = FALSE,  
  xticks,  
  type = "l",  
  add = FALSE,  
  col = graphics::par("fg"),  
  col.se = "orange",  
  alpha.se = 0.5,  
  lty = 1,  
  lwd = 1,  
  cex = 1,  
  las = 1,  
  main = "Welch periodogram",  
  xlab,  
  ylab,  
  xlim = NULL,  
  ylim = NULL,  
  xaxs = "i",  
  yaxs = "i",  
  xline = 1.2 * cex,  
  yline = 2 * cex,  
  mar = c(2.6, 3.8, 2.1, 0.6) * (0.5 + cex/2),  
  mgp = cex * c(2, 0.5, 0),  
  tck = -0.02 * cex,  
  grid = TRUE,
```

```

    ...
)

mv_pwelch(
    x,
    margin,
    fs,
    window = 64,
    noverlap = window/2,
    nfft = "auto",
    window_family = hamming
)

```

### Arguments

x	numerical vector or a row-major vector, signals. If x is a matrix, then each row is a channel. For plot function, x is the instance returned by pwelch function.
fs	sample rate, average number of time points per second
window	window length in time points, default size is 64
noverlap	overlap between two adjacent windows, measured in time points; default is half of the window
nfft	number of points in window function; default is automatically determined from input data and window, scaled up to the nearest power of 2
window_family	function generator for generating filter windows, default is <a href="#">hamming</a> . This can be any window function listed in the filter window family, or any window generator function from package gsignal. Default is <a href="#">hamming</a> . For 'iEEG' users, both <a href="#">hamming</a> and <a href="#">blackmanharris</a> are offered by 'EEG-lab'; while <a href="#">blackmanharris</a> offers better attenuation than Hamming windows, it also has lower spectral resolution. <a href="#">hamming</a> 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 <a href="#">hanning</a> window)
col, xlim, ylim, main, type, cex, las, xlab, ylab, lty, lwd, xaxs, yaxs, mar, mgp, tck	parameters passed to <a href="#">plot.default</a>
plot	integer, whether to plot the result or not; choices are 0, no plot; 1 plot on a new canvas; 2 add to existing canvas
log	indicates which axis should be log <sub>10</sub> -transformed, used by the plot function. For 'x' axis, it's log <sub>10</sub> -transform; for 'y' axis, it's 10log <sub>10</sub> -transform (decibel unit). Choices are "xy", "x", "y", and "".
...	will be passed to <code>plot.pwelch</code> or ignored
se	logical or a positive number indicating whether to plot standard error of mean; default is false. If provided with a number, then a multiple of standard error will be drawn. This option is only available when power is in log-scale (decibel unit)
xticks	ticks to show on frequency axis

<code>add</code>	logical, whether the plot should be added to existing canvas
<code>col.se, alpha.se</code>	controls the color and opacity of the standard error
<code>xline, 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 bits, and one raw number uses 8 bits, hence two raw vectors can form one unsigned integer-16. That is, `raw_to_uint16` requires the length of input to be multiple of two. An easy calculation is: the length of x times 8, must be divided by 'N' (see last paragraph for definition).

The returned data uses the closest available R native data type that can fully represent the data. For example, R does not have single float type, hence `raw_to_float` returns double type, which can represent all possible values in float. For `raw_to_uint32`, the potential value range is 0 - (2<sup>32</sup>-1). This exceeds the limit of R integer type (-2<sup>31</sup>) - (2<sup>31</sup>-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(
    "00000000111111110101000011111111",
    "10000000111111110000000010000000"
  )
} else {
  bitstring <- paste0(
    "00000001000000001111111100000001",
    "11111111000010101111111100000000"
  )
}

```



```
)
}

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

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

---

rcond_filter_ar	<i>Computer reciprocal condition number of an 'Arma' filter</i>
-----------------	---

---

### Description

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

### Usage

```
rcond_filter_ar(a)
```

### Arguments

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

### Value

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

### See Also

[check\\_filter](#)

### Examples

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

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

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

# Bad filter (order is too high)
```

```

filter <- butter(50, 0.1, "low")

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

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

```

---

register\_volume      *Imaging registration using 'NiftyReg'*

---

### Description

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

### Usage

```

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

```

### Arguments

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

### 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)

```

---

rgl-call

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


---

**Description**

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

**Usage**

```
rgl_call(FUN, ...)
```

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

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

**Arguments**

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

**Examples**

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

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

  rgl_view({

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

  })
}
```

---

 shift\_array

*Shift array by index*


---

**Description**

Re-arrange arrays in parallel

**Usage**

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

**Arguments**

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

## Details

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

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

## Value

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

## Examples

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

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

```
y <- NA * x
y[1,1:4] = x[1,2:5]
y[2,1:3] = x[2,3:5]
```

```
# Check if z 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)
```

---

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\_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

### **Examples**

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

vcg_mesh_volume(mesh)
```

---

vcg\_smooth              *Implicitly smooth a triangular mesh*

---

### **Description**

Applies smoothing algorithms on a triangular mesh.

**Usage**

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

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

**Arguments**

mesh	triangular mesh stored as object of class 'mesh3d'.
lambda	In <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

### 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\_uniform\_remesh

*Sample a surface mesh uniformly*


---

### Description

Sample a surface mesh uniformly

**Usage**

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

**Arguments**

x	surface
voxel_size	'voxel' size for space 'discretization'
offset	offset position shift of the new surface from the input
discretize	whether to use step function (TRUE) instead of linear interpolation (FALSE) to calculate the position of the intersected edge of the marching cube; default is FALSE
multi_sample	whether to calculate multiple samples for more accurate results (at the expense of more computing time) to remove artifacts; default is FALSE
absolute_distance	whether an unsigned distance field should be computed. When set to TRUE, non-zero offsets is to be set, and double-surfaces will be built around the original surface, like a sandwich.
merge_clost	whether to merge close vertices; default is TRUE
verbose	whether to verbose the progress; default is TRUE

**Value**

A triangular mesh of class 'mesh3d'

**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.

### 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,
  ...
)

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

```

### Arguments

freqs	frequency in which data will be projected on
srate	sample rate, number of time points per second

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

### Value

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

### Examples

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

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

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

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

# power
power <- Mod(coef[])^2

# Power peaks at 5Hz and 10Hz at early stages
# After 1.0 second, 5Hz component fade away
image(power, x = time, y = freq, ylab = "frequency")

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

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

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

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

plot(kern)
```

# Index

## \* datasets

left\_hippocampus\_mask, 42

as\_matrix4 (new\_matrix4), 46  
as\_quaternion (new\_quaternion), 47  
as\_vector3 (new\_vector3), 47

band\_pass, 3  
band\_pass1 (band\_pass), 3  
band\_pass2 (band\_pass), 3  
baseline\_array, 6  
blackman (filter-window), 34  
blackmanharris, 53  
blackmanharris (filter-window), 34  
blackmannuttall (filter-window), 34  
bohmanwin (filter-window), 34  
butter\_max\_order, 9

check\_filter, 10, 57  
collapse, 11  
convolve, 12  
convolve\_image (convolve), 12  
convolve\_signal (convolve), 12  
convolve\_volume (convolve), 12  
cov, 31

decimate, 14  
design\_filter, 15  
design\_filter\_fir, 15, 16, 17  
design\_filter\_iir, 15, 16, 20  
detect\_threads (parallel-options), 49  
detrend, 22, 70  
diagnose\_channel, 23  
diagnose\_filter, 25  
dijkstras-path, 27  
dijkstras\_surface\_distance  
(dijkstras-path), 27

fast\_cov, 31  
fast\_median (fast\_quantile), 32  
fast\_mvmedian (fast\_quantile), 32

fast\_mvquantile (fast\_quantile), 32  
fast\_quantile, 32  
fill\_surface, 33  
filter-window, 34  
filter\_signal, 35  
filtfilt, 13, 35, 36, 57  
fir1, 37  
firls, 17, 18, 38  
flattopwin (filter-window), 34  
freqz2, 39

get0, 40  
grow\_volume, 39

hamming, 53  
hamming (filter-window), 34  
hanning, 53  
hanning (filter-window), 34  
hist, 23, 24

internal\_rave\_function, 40  
interpolate\_stimulation, 41

kaiser, 18

left\_hippocampus\_mask, 42

matlab\_palette, 42  
mesh\_from\_volume, 43  
morlet\_wavelet (wavelet), 69  
multitaper, 44  
multitaper\_config (multitaper), 44  
mv\_pwelch (pwelch), 51

new\_matrix4, 46, 47, 48  
new\_quaternion, 46, 47, 48  
new\_vector3, 46, 47, 47  
niftyreg, 58  
notch\_filter, 48

par, 24, 51



- parallel-options, 49
- plot, 51
- plot.default, 53
- plot.ravetools-pwelch (pwelch), 51
- plot\_signals, 50
- print.ravetools-pwelch (pwelch), 51
- pwelch, 23, 24, 51
  
- ravetools\_threads (parallel-options), 49
- raw-to-sexp, 54
- raw\_to\_float (raw-to-sexp), 54
- raw\_to\_int16 (raw-to-sexp), 54
- raw\_to\_int32 (raw-to-sexp), 54
- raw\_to\_int64 (raw-to-sexp), 54
- raw\_to\_int8 (raw-to-sexp), 54
- raw\_to\_string (raw-to-sexp), 54
- raw\_to\_uint16 (raw-to-sexp), 54
- raw\_to\_uint32 (raw-to-sexp), 54
- raw\_to\_uint8 (raw-to-sexp), 54
- rawToChar, 55
- rcond\_filter\_ar, 57
- register\_volume, 58
- remez, 18
- rgl-call, 59
- rgl\_call (rgl-call), 59
- rgl\_plot\_normals (rgl-call), 59
- rgl\_view (rgl-call), 59
  
- shift\_array, 60
- surface\_path (dijkstras-path), 27
  
- vcg\_isosurface, 62
- vcg\_mesh\_volume, 43, 63
- vcg\_smooth, 63
- vcg\_smooth\_explicit, 43
- vcg\_smooth\_explicit (vcg\_smooth), 63
- vcg\_smooth\_implicit, 43
- vcg\_smooth\_implicit (vcg\_smooth), 63
- vcg\_sphere, 66
- vcg\_uniform\_remesh, 43, 66
- vcg\_update\_normals, 68
  
- wavelet, 69
- wavelet\_cycles\_suggest (wavelet), 69
- wavelet\_kernels (wavelet), 69