Package 'MatchingPursuit'

June 24, 2026

Type Package

Title Processing Time Series Data Using the Matching Pursuit Algorithm

Version 1.1.0

Author Artur Gramacki [aut, cre] (ORCID:
<https://orcid.org/0000-0002-1610-9743>),
Jarosław Gramacki [ctb] (ORCID:
<https://orcid.org/0000-0001-5032-1353>),
Piotr T. Różański [ctb] (ORCID:
<https://orcid.org/0000-0002-0457-6731>)

Maintainer Artur Gramacki <a.gramacki@gmail.com>

Description Provides tools for analysing and decomposing time series data using the Matching Pursuit (MP) algorithm, a greedy signal decomposition technique that represents complex signals as a linear combination of simpler functions (called atoms) selected from a redundant dictionary. Support for the Orthogonal Matching Pursuit (OMP) variant of the classical MP algorithm is also provided. For more details see Mallat and Zhang (1993) <doi:10.1109/78.258082>, Pati et al. (1993) <doi:10.1109/ACSSC.1993.342465>, Elad (2010) <doi:10.1007/978-1-4419-7011-4> and Różański (2024) <doi:10.1145/3674832>.

SystemRequirements external tool (installed via empi.install() function). The package uses the implementation of the Matching Pursuit algorithm by Piotr T. Różański, available at https://github.com/develancer/empi.

Imports edf, signal, RSQLite, DescTools, imager, raster, graphics, grDevices, utils, digest, EGM, xml2

Suggests knitr, rmarkdown, latex2exp, remotes

VignetteBuilder knitr

Depends R (>= 3.5.0)

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Repository CRAN

Date/Publication 2026-06-24 07:40:45 UTC

Contents

atom_params . . . 2
clear_cache . . . 3
eeg_montage . . . 4
empi_check . . . 5
empi_execute . . . 6
empi_install . . . 7
empi_locate . . . 8
filters_coeff . . . 9
gabor_fun . . . 10
gabor_proj_fft . . . 12
omp_core . . . 13
omp_execute . . . 15
plot.ecg . . . 18
plot.edf . . . 20
plot.mp . . . 21
read_csv_signals . . . 23
read_dict . . . 24
read_ecg_signals . . . 27
read_edf_params . . . 28
read_edf_signals . . . 29
read_empi_db_file . . . 31
run_omp_pipeline . . . 32
sig2bin . . . 34
tf_map . . . 35
topk_atoms . . . 38

Index **42**

`atom_params` *Read atom parameters from a SQLite database*

Description

Reads atom parameters stored in a SQLite database created by `empi_execute()` function.

Usage

```
atom_params(db_file)
```

Arguments

`db_file`	A character string giving the path to a SQLite database file.

Value

A data frame containing the atom parameters stored in the database:

`channel_id`	Channel identifier.
`atom_number`	Atom number.
`energy`	Energy of the atom.
`frequency`	Frequency of the atom.
`phase`	Phase of the atom.
`scale`	Scaling factor.
`position`	Position of the atom in time.

Examples

```
# Example database containing data from 18 channels
file <- system.file("extdata", "EEG_bipolar_filtered.db", package = "MatchingPursuit")
out <- atom_params(file)
out[which(out$channel_id == 1), ]
out[which(out$channel_id == 18), ]

# Example database containing data from a single channel
file <- system.file("extdata", "sample1.db", package = "MatchingPursuit")
out <- atom_params(file)
out
```

---

clear_cache *Clear MatchingPursuit Cache*

Description

Deletes all files in the MatchingPursuit cache directory.

Usage

```
clear_cache()
```

Value

Logical scalar. Returns `TRUE` if all files were successfully removed, and `FALSE` otherwise. The return value is invisible.

Examples

```
if (interactive()) {
  clear_cache()
}
```

eeg_montage *Performs bipolar, reference or average EEG montage*

Description

An EEG montage refers to the arrangement of EEG electrodes and the way their signals are displayed relative to one another during electroencephalogram interpretation. The same EEG recording may appear very different depending on the montage used. This function implements the three montage methods most commonly used in practice: 1) Bipolar Montage, 2) Referential (Monopolar) Montage, and 3) Average Reference Montage.

Usage

```
eeg_montage(
  x,
  montage_type = c("average", "reference", "bipolar"),
  ref_channel = NULL,
  bipolar_pairs = NULL
)
```

Arguments

x	Object of class edf (from read_edf_signals()).
montage_type	A character string specifying the montage type. • "average" - each electrode is referenced to the average of all electrodes • "reference" - each active electrode is compared to a single common reference electrode • "bipolar" - each channel compares two adjacent electrodes
ref_channel	Name of the reference channel for "reference" montage.
bipolar_pairs	List of electrodes pairs for "bipolar" montage. See example below.

Details

To check the channel names in the analysed EEG recording, use the read_edf_params() function.

Value

An object of class edf, which is a list with fields:

signal	Data frame with all signal channels.
sampling_frequency	Data frame with all signals stored in the EDF file.
time_stamps	Sampling rate after optional resampling.
signal_names	Time stamps after optional resampling.
record_name	Signal names.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out <- read_edf_signals(file, resampling = FALSE, from = 0, to = 10)

read_edf_params(file)

# The classical double banana montage.
pairs <- list(
  c("Fp2", "F4"),
  c("F4",  "C4"),
  c("C4",  "P4"),
  c("P4",  "O2"),
  c("Fp1", "F3"),
  c("F3",  "C3"),
  c("C3",  "P3"),
  c("P3",  "O1"),
  c("Fp2", "F8"),
  c("F8",  "T4"),
  c("T4",  "T6"),
  c("T6",  "O2"),
  c("Fp1", "F7"),
  c("F7",  "T3"),
  c("T3",  "T5"),
  c("T5",  "O1"),
  c("Fz",  "Cz"),
  c("Cz",  "Pz")
)

signal_bip_mont <- eeg_montage(out, montage_type = c("bipolar"), bipolar_pairs = pairs)
signal_ref_mont <- eeg_montage(out, montage_type = c("reference"), ref_channel = "O1")
signal_avg_mont <- eeg_montage(out, montage_type = c("average"))

head(signal_bip_mont$signal)
head(signal_ref_mont$signal)
head(signal_avg_mont$signal)
```

`empi_check` *Checks if EMPI external software is installed*

Description

The EMPI program is installed using the `empi_install()` function and stored in the cache directory. This function checks whether the EMPI program is still available there (users have full access to the cache directory and may remove its contents at any time).

Usage

```
empi_check()
```

Value

If the EMPI program is found, its full path is returned. Otherwise, a message is displayed, prompting the user to install it using the `empi_install()` function.

Examples

```
empi_check()
```

`empi_execute` *Launches the empi program*

Description

Runs the EMPI program for the given data (signal).

Usage

```
empi_execute(
  signal,
  empi_options = NULL,
  write_to_file = FALSE,
  path = NULL,
  file_name = NULL
)
```

Arguments

`signal`	List containing the signal in a data frame together with its sampling frequency. The data frame should have meaningful column names (channel names). The list must contain elements named `"signal"` and `"sampling_frequency"`.
`empi_options`	If `NULL`, the EMPI program is run with `"-o local --gabor -i 50 --cpu-workers 8"` parameters. Otherwise, the user may specify any command-line options. See the `README.md` file after downloading the EMPI program using the `empi_install()` function.
`write_to_file`	If `TRUE`, a SQLite database file will be created and saved in the `path` directory or, if `path = NULL`, in the cache directory. This file stores the results of signal decomposition using the MP algorithm
`path`	Directory in which the SQLite database file will be saved. If `NULL`, the file will be saved in the cache directory.
`file_name`	Name of the file to create if `write_to_file = TRUE`.

Details

The EMPI program (source code and binary files for multiple operating systems) can be downloaded from https://github.com/develancer/empi. Details are presented in the journal paper: Różański, P. T. (2024). *empi: GPU-Accelerated Matching Pursuit with Continuous Dictionaries*. ACM Transactions on Mathematical Software, Volume 50, Issue 3, Article No. 17, pp. 1-17, doi:10.1145/3674832.

Value

Results of signal decomposition using the MP algorithm. An object of class `mp` is returned. If `write_to_file = TRUE`, the results are also written to a SQLite file in the `path` directory.

`atoms`	A data frame describing the selected atoms.
`original_signal`	Matrix containing the original signal(s).
`reconstruction`	Matrix containing the reconstructed signal(s).
`gabors`	List of matrices containing selected atoms for each channel.
`t`	Time vector corresponding to signal samples.
`sf`	Sampling frequency.

Examples

```
## Not run:
file <- system.file("extdata", "sample1.csv", package = "MatchingPursuit")
out <- read_csv_signals(file)

out_empi <- empi_execute(
  signal = out,
  empi_options = NULL,
  write_to_file = FALSE,
  path = NULL,
  file_name = NULL
)

plot(out_empi)

## End(Not run)
```

`empi_install` *Installs the EMPI external program*

Description

Downloads the **Enhanced Matching Pursuit Implementation** (EMPI) external program compatible with the current operating system and stores it in the package cache directory.

Usage

```
empi_install()
```

Details

The function detects the operating system (Windows, Linux, macOS arm64), downloads the appropriate archive from the official repository, verifies its integrity using a checksum, and extracts it.

Value

The function downloads the EMPI program in a version compatible with the operating system used (Windows, Linux, MacOS-x64, MacOS-arm64) and stores it in the package cache directory.

Examples

```
if (interactive()) {
  empi_install()
}
```

---

empi_locate *Get required external software localization*

Description

Returns **Enhanced Matching Pursuit Implementation** binary locations for the following operating systems: Windows, Linux, macOS-x64, macOS-arm64.

Usage

```
empi_locate()
```

Value

List with URL of the EMPI binaries and zip file name.

Examples

```
empi_locate()
```

`filters_coeff` *A wrapper function for* `signal::butter()` *function*

Description

Implements notch, low-pass, high-pass, band-pass, and band-stop filters with specified frequency ranges and Butterworth filter order.

Usage

```
filters_coeff(
  sf = 256,
  notch = c(49, 51),
  notch_order = 2,
  lowpass = 30,
  lowpass_order = 4,
  highpass = 1,
  highpass_order = 4,
  bandpass = c(0.5, 40),
  bandpass_order = 4,
  bandstop = c(0.5, 40),
  bandstop_order = 4
)
```

Arguments

`sf`	Sampling frequency.
`notch`	Vector of two frequencies for notch filter.
`notch_order`	Notch filter order.
`lowpass`	Low-pass filter frequency.
`lowpass_order`	Low-pass filter order.
`highpass`	High-pass filter frequency.
`highpass_order`	High-pass filter order.
`bandpass`	Vector of two frequencies for band-pass filter.
`bandpass_order`	Band-pass filter order.
`bandstop`	Vector of two frequencies for band-stop filter.
`bandstop_order`	Band-stop filter order.

Value

List with parameters of individual filters.

`notch`	Notch filter used to remove a specific narrow frequency band.
`lowpass`	Low-pass filter that attenuates high-frequency components.

highpass	High-pass filter that attenuates low-frequency components.
bandpass	Band-pass filter that retains frequencies within a selected range.
bandstop	Band-stop filter that removes frequencies within a selected range.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out <- read_edf_signals(file, resampling = FALSE)
signal <- out$signal
sampling_frequency <- out$sampling_frequency

fc <- filters_coeff(
  sf = sampling_frequency,
  notch = c(49, 51),
  lowpass = 40,
  highpass = 1,
  bandpass = c(0.5, 40),
  bandstop = c(10, 50)
)

print(fc)

signal::freqz(fc$notch, Fs =  sampling_frequency)
signal::freqz(fc$lowpass, Fs =  sampling_frequency)
signal::freqz(fc$highpass, Fs =  sampling_frequency)
signal::freqz(fc$bandpass, Fs =  sampling_frequency)
signal::freqz(fc$bandstop, Fs =  sampling_frequency)

plot(signal[, 1], type = "l", panel.first = grid())

signal_filt <- signal

for (m in 1:ncol(signal)) {
  signal_filt[, m] = signal::filtfilt(fc$notch, signal_filt[, m]); # 50Hz notch filter
 signal_filt[, m] = signal::filtfilt(fc$lowpass, signal_filt[, m]); # Low pass IIR Butterworth
 signal_filt[, m] = signal::filtfilt(fc$highpass, signal_filt[, m]); # High pass IIR Butterwoth
}

plot(signal_filt[, 1], type = "l", panel.first = grid())
```

---

gabor_fun *Gabor function implementation*

Description

A Gabor function is a sinusoidal wave localized by a Gaussian envelope. In signal processing, it is widely used as a basic building block for representing signals localized in both time and frequency. The Matching Pursuit algorithm uses a redundant dictionary of so-called *Gabor atoms*. These atoms are particularly suitable because they: 1) provide optimal time–frequency localization, 2) represent oscillatory signals well, 3) enable adaptive time-frequency decomposition.

Usage

```
gabor_fun(
  number_of_samples,
  sampling_frequency,
  mean,
  phase,
  sigma,
  frequency,
  normalization = TRUE
)
```

Arguments

number_of_samples	Number of samples in the generated atom.
sampling_frequency	Sampling frequency.
mean	Time position of the Gaussian envelope.
phase	Phase of the sinusoidal component.
sigma	Scale parameter controlling the width of the Gaussian window.
frequency	Frequency of the sinusoidal component.
normalization	If TRUE, the resulting atom is normalized to have unit norm.

Value

A list containing four numeric vectors of length number_of_samples:

cosine	Cosine wave.
gauss	Gaussian envelope.
gabor	Gabor function.
t	Time vector corresponding to signal samples.

Examples

```
number_of_samples <- 512
sampling_frequency <- 256.0
mean <- 1
phase <- pi
sigma <- 0.5
frequency <- 5.0
normalization = TRUE

out <- gabor_fun(
  number_of_samples,
  sampling_frequency,
  mean,
  phase,
```

```
  sigma,
  frequency,
  normalization
)

# If normalization = TRUE, norm of atom = 1, we can check it
crossprod(out$gabor)

plot(out$t, out$gabor, type = "l", xlab = "t", ylab = "gabor", panel.first = grid())
```

`gabor_proj_fft` *FFT-based fast computation of inner products between a signal and Gabor atoms*

Description

This function computes inner products between a windowed signal and a set of Gabor atoms using FFT-based frequency-domain operations. Instead of explicitly constructing and shifting atoms in the time domain, it extracts selected Fourier coefficients corresponding to Gabor frequencies. The resulting values provide both complex projection coefficients and their magnitudes.

Usage

```
gabor_proj_fft(block, signal)
```

Arguments

`block`	See the vignette for a description of the structure of blocks.
`signal`	A numeric vector, matrix, or data frame representing the signal(s) to be analyzed. Each column is treated as a separate channel.

Value

A list containing two matrices computed from windowed FFT segments of the signal:

`proj_mod_mtx`	Magnitudes of selected Gabor atom inner products (absolute values of projection coefficients).
`fft_bin_mtx`	Complex Fourier coefficients used to compute inner products with Gabor atoms.

Note

Users do not work directly with this function. It is used internally in the `topk_atoms()` function. However, it can be used by users for their own experiments and tests.

Examples

```
signal <- as.matrix(rnorm(256))
sf <- 256
duration <- 1

xml_file <- system.file("extdata", "one_block_dict.xml", package = "MatchingPursuit")
block <- read_dict(xml_file, sf, duration, verbose = TRUE)
my_list <- gabor_proj_fft(block, signal)

pmm <- my_list$proj_mod_mtx
scm <- my_list$fft_bin_mtx

head(scm)
head(pmm)

# Of course it gives 'pmm'
head(Mod(scm))
```

`omp_core` *Implements Orthogonal Matching Pursuit (OMP) algorithm*

Description

This function implements the Orthogonal Matching Pursuit (OMP) algorithm to compute a sparse representation of a signal using a dictionary of atoms. This is an efficient implementation with incremental Cholesky factorization for efficient least-squares solving. The function works very similarly to the Python implementation of OMP available in the scikit-learn library.

Usage

```
omp_core(
  dictionary,
  signal,
  channel = NULL,
  n_nonzero_coefs = NULL,
  tol = NULL,
  normalize = TRUE,
  fit_intercept = TRUE,
  verbose = FALSE
)
```

Arguments

`dictionary`	A dictionary of atoms. Can be a matrix, data frame, or any object coercible to a matrix. Atoms are assumed to be stored in columns. Alternatively, a `"topk"` object returned by `topk_atoms()`.

signal	A signal matrix or an object coercible to a matrix. Signals are assumed to be stored in columns. The signal length (number of rows) must match the atom length.
channel	Index of the signal (channel) to decompose.
n_nonzero_coefs	Maximum number of non-zero coefficients in the sparse representation. If tol = NULL, the algorithm stops after selecting at most n_nonzero_coefs atoms. If both n_nonzero_coefs and tol are NULL, the default value is max(1, floor(0.1 * ncol(dictionary))). Ignored when tol is specified.
tol	Stopping tolerance defined as the maximum allowed squared residual norm ($\|r\|^2$). The algorithm stops when the residual energy falls below this value. If specified, it overrides n_nonzero_coefs.
normalize	Logical; if TRUE, dictionary atoms are normalized to unit ℓ_2 norm before decomposition.
fit_intercept	Logical; if TRUE, the signal and dictionary atoms are centered before decomposition and an intercept term is estimated.
verbose	Logical; flag indicating whether progress information should be printed.

Details

Unlike classical Matching Pursuit, OMP recomputes all selected coefficients at each iteration by solving a least-squares problem, which generally yields more accurate sparse approximations for a given number of atoms.

Value

A list containing the result of the Orthogonal Matching Pursuit decomposition with the following elements:

gabors	Matrix of selected atoms (dictionary columns) used in the reconstruction.
original_signal	The original signal reconstructed as a vector (including intercept if fit_intercept = TRUE).
reconstruction	The OMP approximation of the signal including intercept (if applicable).
coef	Numeric vector of estimated coefficients for selected atoms.
energy	Energy contribution of selected atoms, computed as coef^2 * colSums(selected_atoms^2).
intercept	Estimated intercept term (0 if fit_intercept = FALSE).
support	Integer vector of selected atom indices.
residual	Final residual vector.
n_iters	Number of iterations performed by the algorithm.

If dictionary is a "topk" object, the result additionally contains:

frequency	Frequencies of selected atoms.
phase	Phases of selected atoms.
scale	Scales of selected atoms.
position	Positions of selected atoms.

See Also

read_dict, topk_atoms, omp_execute, run_omp_pipeline

Examples

```
dictionary <- matrix(
c(
  1.0,  0.9,  0.1,  1.0, -0.2,  0.3,  0.7, -0.5,  1.2,  0.4,
  0.2,  1.0,  0.8, -0.3,  1.0, -0.6,  0.5,  0.9, -0.1,  0.8,
  0.0,  0.1,  1.0,  0.5,  0.7,  1.1, -0.4,  0.2,  0.6, -0.7,
  0.9, -0.2,  0.4,  1.3,  0.1,  0.0,  0.8, -0.9,  0.5,  1.0,
 -0.3,  0.6,  1.1, -0.4,  0.2,  0.7, -0.8,  1.0,  0.3,  0.9),
nrow = 5, byrow = TRUE
)

signal <- matrix(
c(
  4, 3, 5, 2,
  2, 1, 2, 3,
  3, 2, 4, 1,
  5, 4, 3, 2,
  1, 3, 2, 4),
nrow = 5, byrow = TRUE
)

# set 'verbose = TRUE' to see the progress

fit <- omp_core(
  dictionary = dictionary,
  signal = signal,
  channel = 3,
  n_nonzero_coefs = 3,
  verbose = FALSE
)

fit$coef
fit$support

# More realistic example, see omp_execute() examples.
```

`omp_execute` *Orthogonal Matching Pursuit decomposition for multi-channel signals*

Description

Performs sparse signal decomposition using the Orthogonal Matching Pursuit (OMP) algorithm. The decomposition is carried out independently for each signal channel using a dictionary of candidate atoms generated by `topk_atoms()`.

Usage

```
omp_execute(
  dictionary,
  signal,
  sf,
  n_nonzero_coefs,
  tol = NULL,
  normalize = TRUE,
  fit_intercept = TRUE,
  verbose = FALSE
)
```

Arguments

dictionary	A "topk" object returned by topk_atoms(). It contains the candidate atoms and their associated time-frequency parameters.
signal	A matrix, data frame, or object coercible to a matrix containing the signal(s) to decompose. Signals are assumed to be stored in columns.
sf	Sampling frequency of the signal in Hertz.
n_nonzero_coefs	Maximum number of atoms selected by the OMP algorithm for each signal channel.
tol	Optional stopping tolerance defined as the maximum allowed squared residual norm. If specified, it overrides n_nonzero_coefs.
normalize	Logical; if TRUE, atoms are normalized to unit ℓ_2 norm before decomposition.
fit_intercept	Logical; if TRUE, an intercept term is estimated by centering both the signal and the dictionary atoms before decomposition.
verbose	Logical; if TRUE, progress information is printed during processing.

Details

The returned object is of class "mp" and can be visualized using plot() and tf_map().

The function applies omp_core() independently to each signal channel. For every channel, OMP greedily selects atoms from the supplied "topk" dictionary and computes a sparse approximation of the signal.

The resulting object follows the same structure as Matching Pursuit outputs, enabling direct generation of time-frequency maps and visualizations using ‘tf_map()‘ or ‘plot()‘ functions.

Typical workflow:

1. Read a dictionary using read_dict().
2. Generate a signal-adaptive subset of atoms using topk_atoms().
3. Perform sparse decomposition using omp_execute().
4. Visualize the result using plot() or tf_map().

Value

An object of class "mp" containing:

atoms	A data frame describing the selected atoms.
original_signal	Matrix containing the original signal(s).
reconstruction	Matrix containing the reconstructed signal(s).
gabors	List of matrices containing selected atoms for each channel.
t	Time vector corresponding to signal samples.
sf	Sampling frequency.

The atoms data frame contains:

- channel_id — signal channel identifier,
- atom_number — atom index within the channel,
- energy — atom energy contribution,
- envelope — envelope type,
- frequency — atom frequency (Hz),
- phase — atom phase (radians),
- scale — atom scale (seconds),
- position — atom center position (seconds).

See Also

read_dict, topk_atoms, omp_core, run_omp_pipeline

Examples

```
# +-------------------------------------------------------------+
# | Step 1: Read signal                                         |
# +-------------------------------------------------------------+
sig_file <- system.file(
  "extdata",
  "sample3.csv",
  package = "MatchingPursuit"
)

sample3 <- read_csv_signals(
  sig_file,
  col_names_in_csv = TRUE
)

sf <- sample3$sampling_frequency
signal <- sample3$signal
duration <- nrow(signal) / sf

# +-------------------------------------------------------------+
```

```
# | Step 2: Read dictionary definition                          |
# +------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3_dict.xml",
   package = "MatchingPursuit"
)

atoms_dict <- read_dict(
  xml_file = xml_file,
  sf = sf,
  duration = duration,
  verbose = TRUE
)

# +------------------------------------------------------------+
# | Step 3: Generate signal-adaptive atom dictionary           |
# +------------------------------------------------------------+
topk_dict <- topk_atoms(
  atoms_dict = atoms_dict,
  signal = signal,
  sf = sf,
  topk = 5000,
  verbose = TRUE
)

# +------------------------------------------------------------+
# | Step 4: Run Orthogonal Matching Pursuit                    |
# +------------------------------------------------------------+
fit <- omp_execute(
  dictionary = topk_dict,
  signal = signal,
  sf = sf,
  n_nonzero_coefs = 50,
  verbose = TRUE
)

# Inspect results
class(fit)
head(fit$atoms)

# +------------------------------------------------------------+
# | Step 5: Time-frequency map                                 |
# +------------------------------------------------------------+
plot(fit, channel = 3)
```

`plot.ecg` *The function displays ECG signals in a layout corresponding to standard paper ECG printouts*

Description

A typical ECG paper layout was used, with a small grid of 0.04 s × 0.1 mV and a large grid of 0.20 s × 0.5 mV.

Usage

```
## S3 method for class 'ecg'
plot(
  x,
  begin,
  end,
  panel_height = 3,
  small_squares = TRUE,
  zero_line = FALSE,
  ...
)
```

Arguments

x	Object of class ecg (from read_ecg_signals()).
begin	Time point (in seconds) at which to start plotting.
end	Time point (in seconds) at which to stop plotting.
panel_height	Number of large squares to display (according to standard ECG paper): • small grid: 0.04 sec. x 0.1 mV • large grid: 0.20 sec. x 0.5 mV
small_squares	If TRUE, the small grid is also displayed.
zero_line	If TRUE, a horizontal line representing 0 mV is displayed.
...	Currently ignored. Required for compatibility with the generic plot().

Value

No return value, called to visualize an ECG graph.

Examples

```
# ECG data comes from https://physionet.org/content/ptb-xl/1.0.3/
file <- system.file("extdata", "00001_lr.hea", package = "MatchingPursuit")
dir <- dirname(file)
name <- tools::file_path_sans_ext(basename(file))

out <- read_ecg_signals(file)

plot(
  x = out,
  begin = 0,
  end = 10,
  panel_height = 1,
  zero_line = FALSE,
```

```
  small_squares = TRUE
)
```

---

plot.edf *The function displays EEG signals*

Description

Signals are displayed one below another and may be shown in different colours for improved readability.

Usage

```
## S3 method for class 'edf'
plot(
  x,
  begin,
  end,
  panel_height = NULL,
  rainbow = TRUE,
  bg_colour = "black",
  txt_col = "white",
  zero_line = TRUE,
  main = NULL,
  ...
)
```

Arguments

x	Object of class edf (from read_edf_ signals()).
begin	Time point (in seconds) at which to start plotting.
end	Time point (in seconds) at which to stop plotting.
panel_height	Controls the vertical spacing between individual signals. If NULL, the value is chosen automatically so that all signals are clearly visible and do not overlap.
rainbow	If TRUE, individual channels are drawn in different colours.
bg_colour	Background colour.
txt_col	Colour of text elements (axis labels and title).
zero_line	If TRUE, a horizontal line representing 0 mV is displayed.
main	The text shown as the plot title.
...	Currently ignored. Required for compatibility with the generic plot().

Value

No return value, called to visualize an EEG graph.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out  <- read_edf_signals(file, resampling = FALSE)

plot(
  x = out,
  begin = 0,
  end = 10,
  panel_height = NULL,
  rainbow = TRUE,
  bg_colour = "black",
  txt_col = "white",
  zero_line = TRUE,
  main = "EEG signals stored in the EEG.edf file"
)

plot(
  x = out,
  begin = 0,
  end = 10,
  panel_height = NULL,
  rainbow = FALSE,
  bg_colour = "white",
  txt_col = "black",
  zero_line = TRUE,
  main = "EEG signals stored in the EEG.edf file"
)
```

`plot.mp` *Plots a time-frequency (T-F) map to visualize EMPI decomposition*

Description

This function is a wrapper around `tf_map()` with `out_mode = "plot"`.

Usage

```
## S3 method for class 'mp'
plot(
  x,
  channel = 1,
  mode = "sqrt",
  freq_divide = NULL,
  increase_factor = 8,
  shortening_factor_x = 2,
  shortening_factor_y = 2,
  display_crosses = TRUE,
  display_atom_numbers = FALSE,
  display_grid = FALSE,
```

```
  color = "white",
  palette = "my custom palette",
  plot_signals = TRUE,
  ...
)
```

Arguments

x	An object of class mp created by empi_execute().
channel	Channel from the SQLite file to process.
mode	"sqrt", "log", or "linear". Determines the intensity with which the so-called blobs are displayed on the T-F map.
freq_divide	Specifies how many times the displayed frequency range in the T-F map should be reduced. At high sampling rates, and when a low-pass filter with a cut-off frequency much lower than the sampling frequency is used, a large part of the T-F map may contain no blobs. If the sampling frequency is f, the maximum frequency in the T-F map will be ceiling(f / 2 / freq_divide) (f / 2 follows the Nyquist rule). If NULL, it is determined from the atom with the highest frequency fmax according to freq_divide = (f / 2) / fmax.
increase_factor	Factor controlling the increase in the number of pixels along the frequency axis. Non-negative integers such as 2, 4, 5, or 8 are typically appropriate.
shortening_factor_x	Usually, a value of 2 provides better visualization of atoms.
shortening_factor_y	Usually, a value of 2 provides better visualization of atoms.
display_crosses	Whether small crosses should be displayed at the centres of atoms.
display_atom_numbers	Whether atom numbers should be displayed at the centres of atoms.
display_grid	Whether grid lines should be drawn.
color	Color of the small crosses and atom numbers
palette	Palette from the list returned by hcl.pals() or the string "my custom palette".
plot_signals	Whether the original and reconstructed signals should also be displayed.
...	Currently ignored. Required for compatibility with the generic plot().

Value

No return value, called to visualize the empi decomposition.

Examples

```
## Not run:
file <- system.file("extdata", "sample1.csv", package = "MatchingPursuit")
signal <- read_csv_signals(file, col_names = "ch1")
```

```
# Execute the MP algorithm.
mp_class <- empi_execute(signal = signal)

# Plot a time-frequency map based on MP atoms.
plot(mp_class)

## End(Not run)
```

`read_csv_signals` *Reads and validates a CSV file structure*

Description

Reads and validates a CSV file structure

Usage

```
read_csv_signals(file, col_names = NULL, col_names_in_csv = FALSE)
```

Arguments

`file`	File to be read and checked. The first line of the file must contain two numbers: the sampling frequency in Hz (`freq`) and the signal length in seconds (`sec`). The function verifies whether the file contains exactly `round(freq * sec)` samples. The two numbers must be separated by one or more whitespace characters.
`col_names`	Optional character vector of column names. If not specified, default names are created.
`col_names_in_csv`	Logical value. If `TRUE`, the second line of the file is assumed to contain column names.

Value

A list containing:

`signal`	Data frame containing all signals (rows = samples, columns = channels).
`sampling_frequency`	Sampling frequency.

Examples

```
file <- system.file("extdata", "sample1.csv", package = "MatchingPursuit")

# The first line of the file must contain two numbers:
# a) the sampling frequency in Hz
# b) the signal length in seconds
out <- read.csv(file, header = FALSE)
```

```
head(out)

signal <- read_csv_signals(file, col_names = "signal_1")
head(signal$signal)
signal$ sampling_frequency

file <- system.file("extdata", "sample2.csv", package = "MatchingPursuit")
signal <- read_csv_signals(file, col_names = c("signal_1"))
head(signal$signal)
signal$sampling_frequency

# Now, the csv file contains signal names in the second line
file <- system.file("extdata", "sample3.csv", package = "MatchingPursuit")
signal <- read_csv_signals(file, col_names_in_csv = TRUE)
head(signal$signal)
signal$ sampling_frequency
```

`read_dict` *Read dictionary of Gabor atoms from XML file*

Description

The function parses an XML file describing a multiscale Gabor dictionary.

Usage

```
read_dict(xml_file, sf, duration, verbose = FALSE)
```

Arguments

`xml_file`	Path to the XML file containing the dictionary definition.
`sf`	Sampling frequency (in Hz) of the signal associated with the dictionary.
`duration`	Duration of the signal (in seconds) used to determine the number of valid time positions.
`verbose`	Logical; if `TRUE`, prints progress information about parsed blocks and generated atoms.

Details

Each <block> in the XML file defines a time-frequency scale of atoms using three parameters:

- `windowLen` — length of the analysis window (in samples),
- `windowShift` — step size between consecutive windows,
- `fftSize` — FFT size defining frequency resolution.

The function assumes an XML structure containing `param` nodes with `name` and `value` attributes. An example XML file is shown below. For simplicity, the example contains only one block; in practice, dictionary files usually contain multiple blocks.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dict>
 <block>
   <param name="windowLen" value="30"/>
   <param name="windowShift" value="72"/>
   <param name="fftSize" value="32"/>
 </block>
</dict>
```

Each block generates a grid of atoms over time and frequency bins, forming a multiresolution Gabor dictionary. Smaller windows provide better time resolution, while larger windows improve frequency resolution.

This implementation assumes a *finite signal support model* and restricts dictionary generation to atoms fully contained within the signal support. In particular, only atoms satisfying:

$$0 \leq t \leq N - L$$

are generated, where t is the atom start position, N is the signal length, and L is the window length. Atoms that would extend beyond the left or right boundary of the signal are **not included in the dictionary**. If the signal duration is shorter than the window length, no time positions are generated for that block.

Value

A matrix where each row describes a Gabor atom with the following columns:

`block`	Block identifier from the XML file.
`time_sample`	Time position of the atom (in samples).
`time_sec`	Time position of the atom (in seconds).
`freq_bin`	Frequency bin index.
`freq_hz`	Frequency in Hertz.
`window_len`	Window length used for the atom.
`fft_size`	FFT size used for the atom.

Usage in sparse decomposition pipeline

The output of `read_dict()` is a low-level dictionary of atom parameters (time-frequency grid description). It serves as an input to `topk_atoms()`, which:

- evaluates complex Gabor atoms,
- computes phase-invariant cross-correlations with the signal,
- selects the best `topk` atoms per channel,
- constructs a full real-valued atom representation with optimal phase.

The resulting `"topk"` object contains precomputed atoms and metadata that are directly consumed by `omp_core()` for sparse decomposition. In summary, the processing pipeline is: `read_dict()` $\rightarrow$ `topk_atoms()` $\rightarrow$ `omp_core()`.

Exporting dictionaries from the EMPI program

The EMPI program can export dictionary definitions as an XML file containing atom parameters. This file may include additional elements that are not used in this package, these are safely ignored by the `read_dict()` function. This feature enables direct comparison between the EMPI implementation of the Matching Pursuit (MP) algorithm and the Orthogonal Matching Pursuit (OMP) algorithm implemented in `omp_core()`. It should be noted that EMPI includes several advanced optimization strategies that are not present in the current OMP implementation. To ensure comparability of results, EMPI is executed with the parameters `-o none` and `--full-atoms-in-signal`. Their exact meaning is described in the EMPI documentation (see `README.md`).

See Also

`topk_atoms`, `omp_execute` `omp_core`, `run_omp_pipeline`

Examples

```
# +-------------------------------------------------------------+
# | Read signal                                                 |
# +-------------------------------------------------------------+
sig_file <- system.file(
  "extdata",
  "sample3.csv",
  package = "MatchingPursuit"
)

sample3 <- read_csv_signals(
  sig_file,
  col_names_in_csv = TRUE
)

sf <- sample3$sampling_frequency
duration <- nrow(sample3$signal) / sf

# +-------------------------------------------------------------------+
# | Read dictionary definition                                        |
# +-------------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3_dict.xml",
  package = "MatchingPursuit"
)

atoms_dict <- read_dict(
  xml_file,
  sf,
  duration,
  verbose = TRUE
)

# +-------------------------------------------------------------------+
# | Running the EMPI program with the                                 |
```

```
# | --dictionary-output option allows you to save                |
# | (in XML format) data about the dictionary used.              |
# +--------------------------------------------------------------+

#
# Uncomment to run empi_execute() function
#
# dest_dir <- tools::R_user_dir("MatchingPursuit", "cache")

# opts <- paste0(
#  "-o none --gabor -i 50 --full-atoms-in-signal --dictionary-output ",
#  dest_dir,
#   "/sample3_dict_EMPI.xml"
# )

# out_sample3 <- empi_execute(
#   signal = sample3,
#   empi_options = opts
# )

# +--------------------------------------------------------------+
# | Please compare the sample3_dict.xml and sample3_dict_EMPI.xml |
# | files and find out which fields in the latter file are not   |
# | used in the read_dict() function.                            |
# +--------------------------------------------------------------+
con <- file(xml_file, open = "r")
cat(readLines(con, n = 22), sep = "\n")
close(con)

xml_file_2 <- system.file(
  "extdata",
  "sample3_dict_EMPI.xml",
  package = "MatchingPursuit"
)

con <- file(xml_file_2, open = "r")
for (i in 1:35) {
  cat(readLines(con, n = 1), sep = "\n")
}
close(con)
```

`read_ecg_signals` *Reads WFDB-compatible signal and header files*

Description

WFDB (WaveForm DataBase) is a standard file format for storing, reading, and analyzing physiological time-series signals. It is widely used for signals such as ECG, EEG, blood pressure, respiration, and other biomedical waveforms. It was developed by PhysioNet and is commonly used in research datasets.

Usage

```
read_ecg_signals(file)
```

Arguments

file	Path to the ECG record to be read.

Details

A WFDB record typically consists of two main files: .dat - binary signal samples (waveform values), and .hea - a header file describing how to interpret the data. In some cases, additional annotation files such as .atr may be present, containing beat labels or rhythm annotations.

Value

An object of class ecg. The returned value is a list containing:

signal	Matrix of signals stored in the ECG file.
sampling_frequency	Sampling frequency.
time_stamps	Time vector corresponding to signal samples.
lead_names	Names of the ECG leads (channels).
record_name	Name of the file.

Examples

```
# ECG data comes from https://physionet.org/content/ptb-xl/1.0.3/
file <- system.file("extdata", "00001_lr.hea", package = "MatchingPursuit")
dir <- dirname(file)
name <- tools::file_path_sans_ext(basename(file))

out <- read_ecg_signals(file)
head(out$signal)
out$sampling_frequency
out$lead_names

plot(out, begin = 0, end = 10, panel_height = 1.5)
```

---

read_edf_params *Reads a selected EDF or EDF+ file and returns signal parameters*

Description

Reads a selected EDF or EDF+ file and returns basic signal parameters (channel names, sampling frequency of each channel, number of samples per channel, and signal duration in seconds). Additional information stored in EDF+ files (such as interrupted recordings or time-stamped annotations) is not used by the package and is therefore not read.

Usage

```
read_edf_params(file)
```

Arguments

file	Path to the EDF / EDF+ file to be read.

Value

A data frame containing the basic parameters of the EDF / EDF+ file:

channel_name	Name of the given channel.
frequency	Sampling frequency of the given channel.
no_of_samples	Number of samples in the given channel.
length_sec	Length in seconds of the given channel.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
read_edf_params(file)
```

---

read_edf_signals *Reads a selected EDF or EDF+ file and returns signal data*

Description

The function reads a selected EDF or EDF+ file. Optionally, resampling can be performed (upsampling or downsampling).

Usage

```
read_edf_signals(
  file,
  resampling = FALSE,
  sf_new = NULL,
  from = NULL,
  to = NULL,
  verbose = FALSE
)
```

Arguments

file	Path to the EDF / EDF+ file to be read.
resampling	If TRUE, all signals are resampled (either upsampled or downsampled), depending on the original sampling rates of the channels.
sf_new	Target sampling frequency used for upsampling or downsampling.
from	Starting time of the signal to be loaded (in seconds).
to	Ending time of the signal to be loaded (in seconds).
verbose	Logical flag indicating whether progress information should be printed.

Details

If resampling = TRUE, signals are resampled according to the target frequency specified by f.new. Since the EDF standard allows different sampling rates per channel, some channels may be upsampled while others are downsampled. The function does not support independent resampling of individual channels.

Value

An object of class edf, which is a list with fields:

signal	Data frame with all signal channels.
sampling_frequency	Data frame with all signals stored in the EDF file.
time_stamps	Sampling rate after optional resampling.
signal_names	Time stamps after optional resampling.
record_name	Signal names.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out1  <- read_edf_signals(file, resampling = FALSE)

lapply(out1, class)
out1$ sampling_frequency

out2 <- read_edf_signals(file, resampling = TRUE, sf_new = 128, verbose = TRUE)

lapply(out2, class)
out2$ sampling_frequency
```

`read_empi_db_file` *Reads data from a SQLite file created by the Matching Pursuit algorithm*

Description

Reads data from a SQLite file (`.db`) created by the Matching Pursuit algorithm. The reconstructed signal(s) and Gabor function(s) are also returned.

Usage

```
read_empi_db_file(db_file)
```

Arguments

`db_file`	SQLite file.

Value

An object of class `"mp"` containing:

`atoms`	A data frame describing the selected atoms.
`original_signal`	Matrix containing the original signal(s).
`reconstruction`	Matrix containing the reconstructed signal(s).
`gabors`	List of matrices containing selected atoms for each channel.
`t`	Time vector corresponding to signal samples.
`sf`	Sampling frequency.

Examples

```
file <- system.file("extdata", "EEG_bipolar_filtered.db", package = "MatchingPursuit")
out <- read_empi_db_file(file)

n_channels <- ncol(out$original_signal)
original_signal <- out$original_signal
reconstruction <- out$reconstruction
t <- out$t
sf <- out$sf

old.par <- par("mfrow", "pty", "mai")

par(mfrow = c(2, 1))
par(pty = "m")
par(mai = c(0.9, 0.5, 0.3, 0.4))

plot(
  original_signal[,1], type = "l", col = "blue",
```

```
  main = paste("channel: ", 1, " / " , n_channels, " (original signal)",  sep = ""),
  xaxt = "n", ylab = "", xlab = "time [sec]"
)

len <- length(original_signal[, 1])
lab <- seq(t[1], t[len] + 1 / sf, length.out = 11)
axis(side = 1, las = 1, cex.axis = 0.9, at = seq(0, len, length.out = 11), labels = lab)

plot(
  reconstruction[,1], type = "l", col = "blue",
  main = paste("channel: ", 1, " / " , n_channels, " (reconstructed signal)",  sep = ""),
  xaxt = "n", ylab = "", xlab = "time [sec]"
)

axis(side = 1, las = 1, cex.axis = 0.9, at = seq(0, len, length.out = 11), labels = lab)

par(old.par)
```

`run_omp_pipeline` *Run the complete OMP decomposition pipeline*

Description

Executes the full Orthogonal Matching Pursuit (OMP) workflow: reads a signal from a CSV file, loads a dictionary definition from an XML file, selects the most relevant atoms, and performs sparse signal decomposition using OMP.

Usage

```
run_omp_pipeline(
  sig_file,
  col_names_in_csv,
  xml_file,
  topk,
  n_nonzero_coefs,
  tol = NULL,
  normalize = TRUE,
  fit_intercept = TRUE,
  verbose = FALSE
)
```

Arguments

`sig_file`	Path to a CSV file containing the signal data. See `read_csv_signals()` function.
`col_names_in_csv`	Logical. Indicates whether the CSV file contains column names in the first row. See `read_csv_signals()` function.

xml_file	Path to an XML file defining the dictionary atoms. See read_dict() function.
topk	Integer. Number of atoms with the highest correlation to the signal retained for OMP fitting. See topk_atoms() function.
n_nonzero_coefs	Integer. Maximum number of non-zero coefficients to include in the sparse decomposition.
tol	Optional numeric tolerance for the stopping criterion. If NULL, the algorithm stops after selecting n_nonzero_coefs atoms.
normalize	Logical. If TRUE, dictionary atoms are normalized before fitting.
fit_intercept	Logical. If TRUE, an intercept term is included in the model.
verbose	Logical. If TRUE, progress information is printed to the console.

Value

An object of class "mp" containing:

atoms	A data frame describing the selected atoms.
original_signal	Matrix containing the original signal(s).
reconstruction	Matrix containing the reconstructed signal(s).
gabors	List of matrices containing selected atoms for each channel.
t	Time vector corresponding to signal samples.
sf	Sampling frequency.

See Also

omp_core, read_dict, omp_execute topk_atoms

Examples

```
sig_file <- system.file("extdata", "sample3.csv", package = "MatchingPursuit")
xml_file <- system.file("extdata", "sample3_dict.xml", package = "MatchingPursuit")

# set 'verbose = TRUE' to see the progress

out <- run_omp_pipeline(
  sig_file = sig_file,
  col_names_in_csv = TRUE,
  xml_file = xml_file,
  topk = 5000,
  n_nonzero_coefs = 50,
  verbose = TRUE
)

plot(out, channel = 3)
```

sig2bin *Reads input signal(s) from a data frame and returns them in binary format*

Description

Saves the given data (signals) in binary form. The input signal(s) must be a data frame: rows correspond to samples for all channels, and columns correspond to channels. The function is used internally by `empi_execute()`. The binary data consist of floating-point values in the byte order of the current machine (no byte-order conversion is performed).

For multichannel signals, samples are written in time order: first all channels at `t = 0`, then all channels at `t=`Δ`t`, and so on. In other words, the signal is stored in column-major order (rows = channels, columns = samples).

Usage

```
sig2bin(data, write_to_file = FALSE, path = NULL, file_name = NULL)
```

Arguments

`data`	Data frame containing the input signal(s).
`write_to_file`	If `TRUE`, a `.bin` file is created and saved in the `path` directory or, if `path = NULL`, in the cache directory.
`path`	Directory in which the SQLite database file will be saved. If `NULL`, the file will be saved in the cache directory.
`file_name`	Name of the file to create if `write_to_file = TRUE`.

Value

Input signal returned as `raw`. If `write_to_file = TRUE`, a `.bin` file is additionally created and saved in the current directory.

Note

Users do not work directly with `.bin` files. Binary files are used only in `empi_execute()`. The external program *Enhanced Matching Pursuit Implementation* (EMPI), executed inside this function, requires binary input data. This conversion utility may also be useful for users who wish to run EMPI outside of the R environment.

Examples

```
file <- system.file("extdata", "sample3.csv", package = "MatchingPursuit")
out <- read_csv_signals(file, col_names_in_csv = TRUE)

signal_bin <- sig2bin(data = out$signal, write_to_file = FALSE)

# We have 3 channels. The first 4 time points.
```

```
head(out$signal, 4)

# The same elements of the signal in binary (floats are stored in 4 bytes).
head(signal_bin, 48)

# After decoding to numeric.
# Of course we get the same values as in out$signal.
readBin(signal_bin[1:4], what = "numeric", size = 4, endian = "little")
readBin(signal_bin[5:8], what = "numeric", size = 4, endian = "little")
readBin(signal_bin[41:44], what = "numeric", size = 4, endian = "little")
readBin(signal_bin[45:48], what = "numeric", size = 4, endian = "little")
```

`tf_map` *Creates a time-frequency map using atoms from the Matching Pursuit algorithm*

Description

Creates a time-frequency map using atoms from the Matching Pursuit algorithm. The resulting map can be: 1) displayed on the screen, 2) saved as a `.png` file, or 3) saved as an `.RData` object.

Usage

```
tf_map(
  x = NULL,
  channel,
  mode = "sqrt",
  freq_divide = NULL,
  increase_factor = 1,
  shortening_factor_x = 2,
  shortening_factor_y = 2,
  display_crosses = TRUE,
  display_atom_numbers = FALSE,
  display_grid = FALSE,
  color = "white",
  palette = "my custom palette",
  rev = TRUE,
  out_mode = "plot",
  path = NULL,
  file_name = NULL,
  size = c(512, 512),
  draw_ellipses = FALSE,
  plot_signals = TRUE,
  write_atoms = FALSE,
  verbose = TRUE
)
```

Arguments

x	An object of class mp or a path to a SQLite file created by empi_execute().
channel	Channel from the SQLite file to process.
mode	"sqrt", "log", or "linear". Determines the intensity with which the so-called blobs are displayed on the T-F map.
freq_divide	Specifies how many times the displayed frequency range in the T-F map should be reduced. At high sampling rates, especially when a low-pass filter with a cut-off frequency much lower than the sampling frequency is used, a large part of the T-F map may contain no blobs. If the sampling frequency is f, the maximum frequency displayed in the T-F map will be ceiling(f / 2 / freq_divide) (f / 2 follows the Nyquist rule). If NULL, it is determined from the atom with the highest frequency fmax according to freq_divide = (f / 2) / fmax.
increase_factor	Factor controlling the increase in the number of pixels along the frequency axis. Non-negative integers such as 2, 4, 5, or 8 are usually appropriate.
shortening_factor_x	Usually, a value of 2 provides better atom visualization.
shortening_factor_y	Usually, a value of 2 provides better atom visualization.
display_crosses	Whether small crosses should be displayed at the centres of atoms.
display_atom_numbers	Whether atom numbers should be displayed in the canters of atoms.
display_grid	Whether grid lines should be drawn.
color	Color of the small crosses or atom numbers.
palette	Palette from the list returned by the hcl.pals() function or the string "my custom palette".
rev	Value of the rev argument passed to the hcl.colors() function.
out_mode	One of the following: • "plot" - draws a T-F map on the screen. • "file" - saves a T-F map to the file file_name (as a png file). • "RData" - saves the T-F map of size size to file_name (as an R matrix); resampling is performed using the imager::resize() function. • "RData2" - saves the T-F map of size size to file_name (as an R matrix); resampling is performed using the raster::resample() function.
path	Path where png, RData, or pdf files will be written. If NULL, files will be written to the cache directory.
file_name	Name of the png file (if out_mode = "file") or name of the RData file (if out_mode = "RData" or out_mode = "RData2").
size	Size of the png file in pixels (if out_mode = "file") or size of the T-F matrix (if out_mode = "RData" or out_mode = "RData2").
draw_ellipses	Intended for testing only. Can be set to TRUE to display the effect. Works correctly only if out_mode = "plot".

plot_signals	Whether the original and reconstructed signals should also be displayed.
write_atoms	If TRUE, writes all atom plots to the Atoms.pdf file (in the cache directory or in a user-specified directory, depending on path).
verbose	Logical flag indicating whether progress information should be printed.

Value

Depending on the out_mode parameter, the function:

- displays the time-frequency map on the screen
- saves the time-frequency map as a .png file
- saves the time-frequency map as a .RData file

Regardless of the output mode, the function also returns:

gabor_functions	All Gabor functions.
reconstruction	Reconstructed signal.
original_signal	original signal.
sf	sampling frequency.
grid_size_t	Grid size along the time axis.
grid_size_f	Grid size along the frequency axis.
epochSize	Epoch size in samples.
number_of_secs	Signal length in seconds.
tf_map	Time-frequency map.
tf_map_resampled	Resampled time-frequency map (if out_mode = "RData" or out_mode = "RData2"; otherwise NULL).
channel	Processed channel number.
freq_divide	Frequency division factor.

Examples

```
file <- system.file("extdata", "sample1.db", package = "MatchingPursuit")
empi_class <- read_empi_db_file(file)

# 'freq_divide' is set arbitrarily
out <- tf_map(
  x = empi_class,
  channel = 1,
  mode = "sqrt",
  freq_divide = 4,
  increase_factor= 4,
  display_crosses = TRUE,
  display_atom_numbers = FALSE,
  out_mode = "plot",
```

```
)

# 'freq_divide' is determined based on the atom with the highest frequency
out <- tf_map(
  x = empi_class,
  channel = 1,
  mode = "sqrt",
  increase_factor= 4,
  display_crosses = TRUE,
  display_atom_numbers = FALSE,
  out_mode = "plot",
)
```

`topk_atoms` *Select best Gabor atoms based on phase-invariant similarity*

Description

This function constructs a sparse, signal-dependent Gabor dictionary by selecting the most relevant atoms from a precomputed atom dictionary.

Usage

```
topk_atoms(
  atoms_dict,
  signal,
  sf,
  topk = NULL,
  sigma_divisor = NULL,
  verbose = FALSE
)
```

Arguments

`atoms_dict`	A matrix describing Gabor atoms (e.g. output of `read_dict()`). Each row represents a candidate atom and must contain the following columns: `block`, `time_sec`, `freq_hz`, and `window_len`. Other columns are ignored.
`signal`	A numeric vector, matrix, or data frame representing the signal(s) to be analyzed. Each column is treated as a separate channel.
`sf`	Sampling frequency (Hz) of the signal.
`topk`	Number of best atoms to select per signal. If `NULL`, defaults to `ceiling(0.05 * nrow(atoms_dict))`.
`sigma_divisor`	Optional parameter controlling the width of the Gaussian window. Larger values produce narrower windows. If `NULL`, a default heuristic is used.
`verbose`	Logical; if `TRUE`, progress information is printed during both similarity computation and atom generation.

Details

In the first step, phase-invariant similarities between complex Gabor atoms and the input signal are computed using cross-products. In the second step, the top-ranked atoms are reconstructed with optimal phase alignment and converted into real-valued time-domain signals. The resulting object is used as input to `omp_core()` or to `omp_execute()`. This second function is a wrapper around the first function. It is prepared in such a way that an object of class `mp` is created as output. This allows it to be passed to the `tf_map()` function, which creates a time-frequency map.

Value

An object of class `"topk"`, a list containing:

`inner_products`	Matrix of phase-invariant similarities between all atoms in the dictionary and signal channels.
`topk_indices`	Matrix of indices of the selected top-k atoms for each channel.
`atoms`	List of matrices containing reconstructed real-valued atoms (one matrix per signal channel, where columns represent individual atoms).
`frequency`	Matrix of frequencies (Hz) of the selected atoms for each channel.
`phase`	Matrix of optimal phase values used for atom reconstruction.
`scale`	Matrix of Gaussian window scales (normalized sigma in seconds) for each atom.
`position`	Matrix of time positions (centers of atoms in seconds) for each atom.
`atom_begin`	Matrix of start times of each atom (in seconds).
`window_len`	Matrix of window lengths (in seconds).

See Also

`read_dict`, `omp_execute` `omp_core`

Examples

```
# +-------------------------------------------------------------+
# | Step 1: Read signal                                         |
# +-------------------------------------------------------------+
sig_file <- system.file(
  "extdata",
  "sample3.csv",
  package = "MatchingPursuit"
)

sample3 <- read_csv_signals(
  sig_file,
  col_names_in_csv = TRUE
)

sf <- sample3$sampling_frequency
signal <- sample3$signal
duration <- nrow(sample3$signal) / sf
```

```
# +------------------------------------------------------------+
# | Step 2: Read dictionary                                    |
# +------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3_dict.xml",
  package = "MatchingPursuit"
)

atoms_dict <- read_dict(
  xml_file,
  sf,
  duration,
  verbose = TRUE
)

head(atoms_dict)
tail(atoms_dict)
nrow(atoms_dict)

# +------------------------------------------------------------+
# | Step 3: Select top-k atoms most similar to the signal      |
# +------------------------------------------------------------+
out_topk_atoms <- topk_atoms(
  atoms_dict = atoms_dict,
  signal = signal,
  sigma_divisor = NULL,
  sf = sf,
  topk = 5000,
  verbose = TRUE
)

class(out_topk_atoms)

# +------------------------------------------------------------+
# | Step 4.1                                                   |
# | Apply OMP to obtain a sparse representation of the signal  |
# +------------------------------------------------------------+
# | Output: object of class 'mp'                              |
# | Processes: all signal channels (3 in this example)         |
# +------------------------------------------------------------+
fit_1 <- omp_execute(
  dictionary = out_topk_atoms,
  signal = signal,
  sf = sf,
  n_nonzero_coefs = 50
)

class(fit_1)

# +------------------------------------------------------------+
# | Step 4.2                                                   |
# | Apply OMP to obtain a sparse representation of the signal  |
```

```
# +-------------------------------------------------------------+
# | Output: list with atom parameters                           |
# | Processes: one selected channel                             |
# +-------------------------------------------------------------+
fit_2 <- omp_core(
  dictionary = out_topk_atoms,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 50
)

# +-------------------------------------------------------------+
# | Step 5: Plot time-frequency representation                  |
# +-------------------------------------------------------------+
plot(fit_1, channel = 3)
```

Index

atom_params, 2

clear_cache, 3

eeg_montage, 4
empi_check, 5
empi_execute, 6
empi_install, 7
empi_locate, 8

filters_coeff, 9

gabor_fun, 10
gabor_proj_fft, 12

omp_core, 13, *17*, *26*, *33*, *39*
omp_execute, *15*, 15, *26*, *33*, *39*

plot.ecg, 18
plot.edf, 20
plot.mp, 21

read_csv_signals, 23
read_dict, *15*, *17*, 24, *33*, *39*
read_ecg_signals, 27
read_edf_params, 28
read_edf_signals, 29
read_empi_db_file, 31
run_omp_pipeline, *15*, *17*, *26*, 32

sig2bin, 34

tf_map, 35
topk_atoms, *15*, *17*, *26*, *33*, 38