Package 'MatchingPursuit'

August 19, 2026

Type Package

Title Processing Time Series Data Using the Matching Pursuit Algorithm

Version 1.2.0

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 (Enhanced Matching Pursuit Implementation; EMPI) 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)

BugReports https://github.com/artur-gramacki/MatchingPursuit/issues

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Repository CRAN

Date/Publication 2026-08-19 15:10:02 UTC

Contents

`as_sig` *Convert a Signal to a* `sig` *Object*

Description

Creates an object of class `sig` from signal data already available in R. The function provides a convenient way to prepare signals for subsequent processing and decomposition without importing them from a file.

Usage

```
as_sig(signal, sampling_frequency)
```

Arguments

signal	A numeric vector, matrix, or data frame containing the signal values. For multi-channel signals, individual channels are assumed to be stored in columns.
sampling_frequency	A single positive numeric value specifying the sampling frequency of the signal in Hz.

Details

The time vector is generated automatically from the number of signal samples and the specified sampling frequency. The resulting object has the same basic structure as objects returned by `read_csv_signals()`.

Value

An object of class `sig`, which is a list containing:

- `signal`: A data frame containing the signal values.
- `sampling_frequency`: The sampling frequency in Hz.
- `time`: A numeric vector containing the time coordinates of the signal samples in seconds, starting at 0.

See Also

`read_csv_signals`, `read_edf_signals`, `read_wfdb_signals`

Examples

```
# Single-channel signal
x <- rnorm(1000)
sig <- as_sig(x, sampling_frequency = 100)
str(sig)

# Multi-channel signal
x <- cbind(
  channel1 = rnorm(1000),
  channel2 = rnorm(1000)
)
sig <- as_sig(x, sampling_frequency = 100)
str(sig)
```

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()
}
```

design_filters *Design Butterworth filters*

Description

Designs notch, low-pass, high-pass, band-pass, and band-stop Butterworth filters for a specified sampling frequency.

Usage

```
design_filters(
  sampling_frequency = 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

sampling_frequency	Sampling frequency in Hz.
notch	Numeric vector of length two specifying the lower and upper cutoff frequencies of the notch filter in Hz.
notch_order	Positive integer specifying the notch filter order.
lowpass	Numeric value specifying the low-pass cutoff frequency in Hz.
lowpass_order	Positive integer specifying the low-pass filter order.
highpass	Numeric value specifying the high-pass cutoff frequency in Hz.
highpass_order	Positive integer specifying the high-pass filter order.
bandpass	Numeric vector of length two specifying the lower and upper cutoff frequencies of the band-pass filter in Hz.
bandpass_order	Positive integer specifying the band-pass filter order.
bandstop	Numeric vector of length two specifying the lower and upper cutoff frequencies of the band-stop filter in Hz.
bandstop_order	Positive integer specifying the band-stop filter order.

Value

A list containing the designed Butterworth filter objects:

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 <- design_filters(
  sampling_frequency = 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())
```

`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()`) or data frame with samples in rows and channels in columns. The data frame must have column names corresponding to channel names.
`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 containing all signal channels.
`sampling_frequency`	Sampling frequency.
`time`	Time stamps.
`signal_names`	Names of the signal channels.
`record_name`	Name of the EDF file.

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

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

`empi_check` *Check whether EMPI 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

A character string containing the full path to the EMPI executable if found. If EMPI is not available, invisibly returns `NULL` and displays a message suggesting installation with `empi_install()`.

See Also

`empi_install`, `empi_locate`, `empi_execute`, `plot.mp`

Examples

```
if (interactive()) {
  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	An object of class sig returned by read_csv_signals(), an object of class edf returned by read_edf_signals(), or an object of class wfdb returned by read_wfdb_signals().
empi_options	If NULL, the EMPI program is run with "-o local --gabor -i 50" 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.
...	Additional arguments passed to system() when executing EMPI, such as ignore.stdout = TRUE or ignore.stderr = 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.
signal	Matrix containing the original signal(s).
reconstruction	Matrix containing the reconstructed signal(s).
selected_atoms	List of matrices containing selected atoms for each channel.
time	Time vector corresponding to signal samples.
sampling_frequency	Sampling frequency.

See Also

empi_check, empi_install, empi_locate, plot.mp

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
)

# Default EMPI options have been changed. For details, see the EMPI README.md file.
out_empi <- empi_execute(
  signal = out,
  empi_options = "-o none --full-atoms-in-signal -i 50 --gabor",
  write_to_file = FALSE,
  path = NULL,
  file_name = NULL
)

plot(out_empi, freq_divide = 4)

## 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.

See Also

empi_check, empi_locate, empi_execute, plot.mp

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-arm64.

Usage

```
empi_locate()
```

Value

A list containing:

- `url`: URL of the EMPI binary archive,
- `fname`: archive file name.

See Also

empi_check, empi_install, empi_execute, plot.mp

Examples

```
empi_locate()
```

gabor_atom *Generate a Gabor atom*

Description

Generates a real-valued Gabor atom consisting of a sinusoidal component localized by a Gaussian envelope. Gabor atoms provide simultaneous localization in time and frequency and are commonly used in time-frequency dictionaries for Matching Pursuit decomposition.

Usage

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

Arguments

number_of_samples	Positive integer specifying the number of samples in the generated Gabor atom.
sampling_frequency	Sampling frequency in Hz.
mean	Time position of the center of the Gaussian envelope, in seconds.
phase	Phase of the sinusoidal component, in radians.
sigma	Positive scale parameter controlling the width of the Gaussian envelope, in seconds.
frequency	Frequency of the sinusoidal component in Hz.
normalization	Logical; if TRUE, the resulting Gabor atom is normalized to unit Euclidean norm.

Value

A list containing four numeric vectors of length number_of_samples:

cosine	Cosine wave.
gauss	Gaussian envelope.
gabor	Gabor function.
time	Time vector corresponding to the 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_atom(
  number_of_samples,
  sampling_frequency,
  mean,
  phase,
  sigma,
  frequency,
  normalization
)

# Verify unit-norm normalization
sqrt(sum(out$gabor^2))

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

---

gabor_projection_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_projection_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

This function is primarily intended for internal use by `topk_atoms()`, but it is exported to support advanced experiments and methodological testing.

Examples

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

xml_file <- system.file("extdata", "one_block.xml", package = "MatchingPursuit")
block <- read_gabor_dict(xml_file, sampling_frequency, duration, verbose = TRUE)
my_list <- gabor_projection_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))
```

`generate_xml_dict` *Generate an EMPI-compatible Gabor dictionary*

Description

Generates an XML dictionary file containing a set of Gabor atoms following a reconstruction of the Matching Pursuit Tool Kit (MPTK) dictionary generation strategy.

Usage

```
generate_xml_dict(N, file)
```

Arguments

`N`	Integer. Length of the analyzed signal in samples. The maximum generated window length is `N-1`.
`file`	Character string. Path to the XML file that will be created. The output follows the MPTK dictionary XML structure and contains Gabor blocks.

Details

The generated dictionary contains multiple Gabor blocks with logarithmically distributed window lengths. The smallest window length is fixed to 17 samples and the largest window length is the largest odd integer not exceeding `N - 1`

The window lengths are generated on a logarithmic scale and then quantized to obtain a set of practical window sizes. Window lengths are forced to be odd, which provides an exact temporal centre for symmetric Gaussian/Gabor windows. The window shift is estimated as approximately 6 percent of the window length:

$$windowShift = round(0.06 * windowLen)$$

The FFT size is selected as the smallest power of two satisfying:

$$fftSize >= 2 * windowLen$$

This corresponds to the zero-padding strategy commonly used in MPTK-like Gabor dictionaries.

The dictionary generation procedure is based on the following rules. The rules were derived from an analysis of XML files generated by the EMPI program using the `--dictionary-output` option (see the `read_gabor_dict()` function and the corresponding vignette available on CRAN).

1. Minimum window length: 17 samples.
2. Maximum window length: `N-1` samples.
3. Number of scales:
$$K = ceil(log2(N)) + 3$$
4. Logarithmic spacing of scales:
$$L_k = 17r^k$$
where
$$r = ((N-1)/17)^{1/(K-1)}$$
5. Quantization of window lengths depending on their size.
6. Enforcement of odd window lengths.
7. Window shift proportional to window length.
8. FFT size selected as a power of two.

Value

The function writes an XML dictionary file to `file`. Invisibly returns a data frame containing the generated dictionary parameters:

- `windowLen` - Gabor window length in samples,
- `windowShift` - temporal shift between consecutive atoms,
- `fftSize` - FFT size used for the block.

See Also

`read_gabor_dict`,

Examples

```
## Not run:
# Generate a dictionary for a 4096-sample signal
generate_xml_dict(
  N = 4096,
  file = "dictionary_4096.xml"
)

# Inspect generated parameters without reading the XML file
xml_file <- tempfile(fileext = ".xml")

dict <- generate_xml_dict(
  N = 256,
  file = xml_file
)

dict

atoms_dict <- read_gabor_dict(
  xml_file,
  sampling_frequency = 128,
  duration = 2,
  verbose = TRUE
)

head(atoms_dict)
tail(atoms_dict)

## End(Not run)
```

---

mp_core *Implements the Classical Matching Pursuit (MP) Algorithm*

Description

Computes a sparse representation of a signal using the classical Matching Pursuit (MP) algorithm and a dictionary of atoms.

Usage

```
mp_core(
  dictionary,
  signal,
  channel = NULL,
  n_nonzero_coefs = NULL,
  tol = NULL,
  normalize = 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 expressed as the maximum allowed relative residual energy, $\|r\|_2^2/\|x\|_2^2$. 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.
verbose	Logical; flag indicating whether progress information should be printed.

Details

This is a pure R implementation of the MP algorithm. It is primarily intended for reference and educational purposes and is slower and less numerically precise than the C++ implementation provided with this package. See empi_locate(), empi_install(), empi_check(), and empi_execute() for information about the C++ implementation.

Value

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

selected_atoms	Matrix of selected atoms (dictionary columns) used in the reconstruction.
signal	The original signal reconstructed as a vector.
reconstruction	The MP approximation of the signal.
coefs	Numeric vector of estimated coefficients for selected atoms.
energy	Energy contribution of selected atoms, computed as coefs^2.
support	Integer vector of selected atom indices at every iteration.
residual	Final residual vector.
n_iters	Number of iterations performed by the algorithm.
relative_residual_energy	Fraction of the original signal energy that remains unexplained after each Matching Pursuit iteration. Values close to zero indicate a better reconstruction.

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_gabor_dict, topk_atoms, mp_omp_execute, mp_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 <- mp_core(
  dictionary = dictionary,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 3,
  normalize = TRUE,
  verbose = TRUE
)

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

---

mp_omp_execute *Matching Pursuit (MP) or Orthogonal Matching Pursuit (OMP) decomposition for multi-channel signals*

Description

Performs sparse signal decomposition using either the Matching Pursuit (MP) or Orthogonal Matching Pursuit (OMP) algorithm, as specified by the mode parameter. The decomposition is performed independently for each signal channel using a dictionary of candidate atoms generated by topk_atoms().

Usage

```
mp_omp_execute(
  mode = NULL,
  dictionary,
  signal,
  n_nonzero_coefs = NULL,
  tol = NULL,
  normalize = TRUE,
  fit_intercept = TRUE,
  verbose = FALSE
)
```

Arguments

mode	"omp" or "mp". Specifies the algorithm to use for signal decomposition.
dictionary	A "topk" object returned by topk_atoms(). It contains the candidate atoms and their associated time-frequency parameters.
signal	An object of class sig returned by read_csv_signals(), an object of class edf returned by read_edf_signals(), or an object of class wfdb returned by read_wfdb_signals().
n_nonzero_coefs	Maximum number of atoms selected during the decomposition for each signal channel.
tol	Optional stopping tolerance defined as the maximum allowed relative residual energy. 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. Only used when mode == "omp".
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() or mp_core() independently to each signal channel. For every channel, the selected algorithm (MP or OMP) greedily builds a sparse approximation of the signal using atoms from the supplied "topk" dictionary.

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_gabor_dict()`.
2. Generate a signal-adaptive subset of atoms using `topk_atoms()`.
3. Perform sparse decomposition using `mp_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).
`selected_atoms`	List of matrices containing selected atoms for each channel.
`time`	Time vector corresponding to signal samples.
`sampling_frequency`	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 centre position (seconds).

See Also

`read_gabor_dict`, `topk_atoms`, `omp_core`, `mp_core`, `mp_omp_pipeline`

Examples

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

signal <- read_csv_signals(
  sig_file,
```

```
  col_names_in_csv = TRUE
)

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

# +------------------------------------------------------------+
# | Step 2: Read dictionary definition                         |
# +------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3.xml",
   package = "MatchingPursuit"
)

atoms_dict <- read_gabor_dict(
  xml_file = xml_file,
  sampling_frequency = sampling_frequency,
  duration = duration,
  verbose = TRUE
)

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

# +------------------------------------------------------------+
# | Step 4.1: Run Orthogonal Matching Pursuit                  |
# +------------------------------------------------------------+
fit_omp <- mp_omp_execute(
  mode = "omp",
  dictionary = topk_dict,
  signal = signal,
  n_nonzero_coefs = 20,
  verbose = TRUE,
  fit_intercept = FALSE,
)

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

# +------------------------------------------------------------+
# | Step 4.2: Run Matching Pursuit                             |
# +------------------------------------------------------------+
fit_mp <- mp_omp_execute(
  mode = "mp",
```

```
  dictionary = topk_dict,
  signal = signal,
  n_nonzero_coefs = 20,
  verbose = TRUE
)

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

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

# +------------------------------------------------------------+
# | Execute the complete pipeline (steps 1-4)                  |
# | using a single function                                    |
# +------------------------------------------------------------+
fit <- mp_omp_pipeline(
  mode = "mp",         # or "omp" for Orthogonal Matching Pursuit
  sig_file = sig_file,
  col_names_in_csv = TRUE,
  xml_file = xml_file,
  topk = 5000,
  n_nonzero_coefs = 20,
  verbose = FALSE
)

plot(fit, channel = 3)
```

`mp_omp_pipeline` *Run an MP or OMP decomposition pipeline*

Description

Runs a higher-level Matching Pursuit (MP) or Orthogonal Matching Pursuit (OMP) decomposition workflow for signals stored in CSV format. The function: (1) imports the signal, (2) loads a Gabor dictionary definition from an XML file, (3) selects the most relevant candidate atoms, and (4) performs sparse decomposition using the selected algorithm.

Signal-specific preprocessing, such as filtering, resampling, or EEG montage construction, should be performed separately before using this function.

Usage

```
mp_omp_pipeline(
  mode = NULL,
```

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

Arguments

mode	Character string, either "mp" or "omp", specifying the decomposition algorithm.
sig_file	Path to a CSV file containing the signal data.
col_names_in_csv	Logical; indicates whether the CSV file contains column names in the first row. See read_csv_signals().
xml_file	Path to an XML file defining the Gabor dictionary. See read_gabor_dict().
topk	Positive integer specifying the number of candidate atoms with the highest similarity to the signal retained for MP/OMP decomposition. See topk_atoms().
n_nonzero_coefs	Maximum number of non-zero coefficients in the sparse decomposition. If tol = NULL, the algorithm stops after selecting at most this number of atoms.
tol	Optional numeric tolerance for the stopping criterion. If specified, the algorithm stops when the residual energy falls below this value and n_nonzero_coefs is ignored.
normalize	Logical; if TRUE, dictionary atoms are normalized to unit L2 norm before decomposition.
fit_intercept	Logical; if TRUE, an intercept term is included in the model. Used only when mode = "omp".
verbose	Logical; if TRUE, progress information is printed to the console.

Value

An object of class mp containing the decomposition results. See mp_omp_execute().

See Also

omp_core, mp_core, mp_omp_execute, topk_atoms, read_gabor_dict,

Examples

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

out_mp <- mp_omp_pipeline(
```

```
  mode = "mp",
  sig_file = sig_file,
  col_names_in_csv = TRUE,
  xml_file = xml_file,
  topk = 5000,
  n_nonzero_coefs = 50,
  verbose = TRUE
)

out_omp <- mp_omp_pipeline(
  mode = "omp",
  sig_file = sig_file,
  col_names_in_csv = TRUE,
  xml_file = xml_file,
  topk = 5000,
  n_nonzero_coefs = 50,
  verbose = TRUE
)

plot(out_mp, channel = 2)
plot(out_omp, channel = 2)
```

`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 implementation follows the algorithm used by sklearn.linear_model.OrthogonalMatchingPursuit, including incremental Cholesky updates for solving the least-squares problem.

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:

selected_atoms	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).
coefs	Numeric vector of estimated coefficients for selected atoms.
energy	Energy contribution of selected atoms, computed as coefs^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_gabor_dict`, `topk_atoms`, `mp_omp_execute`, `mp_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
)

fit <- omp_core(
  dictionary = dictionary,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 3,
  fit_intercept = FALSE,
  verbose = TRUE
)

fit$coef
# [1] 5.282278 2.637693 2.195920

fit$support
# [1] 9 5 7

# More realistic example, see omp_execute() examples.

#------------------------------------------------------
```

```
# Comparison with the Python implementation.
# The results are identical.
#-----------------------------------------------------------
# import numpy as np
# from sklearn.linear_model import OrthogonalMatchingPursuit
# from sklearn.preprocessing import normalize

# import sklearn
# print(sklearn.__version__)
# 1.8.0

# A = np.array([
#   [ 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]
# ])
# A = normalize(A, axis = 0)

# y = np.array([
#   [4, 3, 5, 2],
#   [2, 1, 2, 3],
#   [3, 2, 4, 1],
#   [5, 4, 3, 2],
#   [1, 3, 2, 4]
# ])

# omp = OrthogonalMatchingPursuit(n_nonzero_coefs = 3, fit_intercept = False)
# omp.fit(A, y)

# print(omp.coef_)

# [[0.   0.   0.         0.   2.63769257 0.    2.19591982 0.  5.28227763 0.        ]
#  [0.   0.   1.93331753 0.   0.         0.    0.         0.  3.65053608 2.21995968]
#  [0.   0.   0.         0.   2.75657594 0.    0.         0.  6.55405854 0.64133666]
#  [0.   0.   3.3515668  0.   0.         0.    0.         0.  0.98551322 2.81352305]]
```

`plot.edf` *Plots EEG signals stored in an object of class* `edf`

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 = NULL,
  end = NULL,
  panel_height = NULL,
  rainbow = FALSE,
  bg_colour = "white",
  txt_col = "black",
  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. If NULL, plotting starts at the beginning of the signal (0 seconds).
end	Time point (in seconds) at which to stop plotting. If NULL, plotting continues to the end of the signal.
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` returned by `empi_execute()` or `mp_omp_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.
out_empi <- empi_execute(signal = signal)

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

## End(Not run)
```

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

Description

ECG signals are read from files in WFDB format.

Usage

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

Arguments

x	Object of class wfdb (from read_wfdb_signals()).
begin	Time point (in seconds) at which to start plotting. If NULL, plotting starts at the beginning of the signal (0 seconds).
end	Time point (in seconds) at which to stop plotting. If NULL, plotting continues to the end of the signal.
panel_height	Height of each ECG channel panel (in mV). One large ECG-paper square corresponds to 0.5 mV. 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().

Details

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 is the file format used by the PhysioNet project and is commonly used in research datasets.

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.

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.

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")
out <- read_wfdb_signals(file)

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

---

read_atom_params *Read atom parameters from a SQLite database*

Description

Reads the atom parameters from a SQLite database produced by `empi_execute()`.

Usage

```
read_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_filter_resample_montage.db", package = "MatchingPursuit")
out <- read_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 <- read_atom_params(file)
out
```

---

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.

time Time vector corresponding to signal samples.

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
head(signal$time)
tail(signal$time)

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_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	Channel name.
frequency	Channel sampling frequency.
no_of_samples	Number of samples in the channel.
length_sec	Channel duration, in seconds.

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 containing all signal channels.
`sampling_frequency`	Sampling frequency after optional resampling.
`time`	Time stamps after optional resampling.
`signal_names`	Names of the signal channels.
`record_name`	Name of the EDF file.

Examples

```
# Read EDF signals without resampling
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out1  <- read_edf_signals(file, resampling = FALSE)

lapply(out1, class)
out1$sampling_frequency

# Read EDF signals and resample them to 128 Hz
out2 <- read_edf_signals(file, resampling = TRUE, sf_new = 128, verbose = TRUE)

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

`read_empi_db` *Read EMPI decomposition results from a SQLite database*

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(db_file)
```

Arguments

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

Value

An object of class "mp" containing:

atoms A data frame describing the selected atoms.

signal Matrix containing the original signal(s).

reconstruction Matrix containing the reconstructed signal(s).

selected_atoms List of matrices containing selected atoms for each channel.

time Time vector corresponding to signal samples.

sampling_frequency Sampling frequency.

Examples

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

n_channels <- ncol(out$signal)
signal <- out$signal
reconstruction <- out$reconstruction
t <- out$time
sampling_frequency <- out$sampling_frequency

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

len <- length(signal[, 1])
lab <- seq(t[1], t[len] + 1 / sampling_frequency, 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)
```

`read_gabor_dict` *Read a Gabor dictionary from an XML file*

Description

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

Usage

```
read_gabor_dict(xml_file, sampling_frequency, duration, verbose = FALSE)
```

Arguments

`xml_file`	Path to the XML file containing the dictionary definition.
`sampling_frequency`	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_gabor_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 a typical native R workflow, the output of `read_gabor_dict()` is passed to `topk_atoms()`, and the resulting `"topk"` object is then used by `mp_omp_execute()` or the lower-level `mp_core()` and `omp_core()` functions.

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_gabor_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, mp_omp_execute, omp_core, mp_core, mp_omp_pipeline, generate_xml_dict

Examples

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

sample3 <- read_csv_signals(
  sig_file,
  col_names_in_csv = TRUE
)

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

# +-------------------------------------------------------------------+
# | Read dictionary definition                                        |
# +-------------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3.xml",
  package = "MatchingPursuit"
)

atoms_dict <- read_gabor_dict(
  xml_file,
  sampling_frequency,
  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_EMPI.xml"
# )

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

# +-----------------------------------------------------------------+
# | Please compare the sample3.xml and sample3_EMPI.xml           |
# | files and find out which fields in the latter file are not    |
# | used in the read_gabor_dict() function.                       |
# +-----------------------------------------------------------------+
con <- file(xml_file, open = "r")
cat(readLines(con, n = 22), sep = "\n")
close(con)

xml_file_2 <- system.file(
  "extdata",
  "sample3_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_wfdb_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 is the file format used by the PhysioNet project and is commonly used in research datasets.

Usage

```
read_wfdb_signals(file)
```

Arguments

`file`	Path to the WFDB 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 `wfdb`. The returned value is a list containing:

signal Matrix of signals stored in the WFDB file.

sampling_frequency Sampling frequency.

time Time vector corresponding to signal samples.

lead_names Names of the WFDB leads (channels).

record_name Name of the file.

Note

The function `EGM::read_wfdb()` from version 0.2.0 of the `EGM` package does not support multi-frequency signals. Consequently, records containing different numbers of samples per frame, as indicated by the `16x2`, `16x4`, and `16x1` specifications below, cannot be read correctly.

```
multi_freq_test 3 100 1000
multi_freq_test.dat 16x2 200.0(0)/mV 16 0 0 258 0 ECG
multi_freq_test.dat 16x4 400.0(0)/mmHg 16 0 400 57824 0 ABP
multi_freq_test.dat 16x1 100.0(0)/pm 16 0 0 18204 0 RESP
```

Examples

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

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

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

---

resample_signal *Resample a signal (upsampling or downsampling)*

Description

Resamples one or more dimensional numeric signals using `signal::resample()`.

Usage

```
resample_signal(signal, p, q, d = 5)
```

Arguments

signal	A numeric vector, numeric matrix, or data frame containing only numeric columns. For two-dimensional objects, rows correspond to time samples and columns correspond to signal channels.
p	A positive integer specifying the interpolation factor.
q	A positive integer specifying the decimation factor.
d	A positive integer specifying the filter delay. The default is 5.

Details

The new sampling frequency is determined by the ratio p/q:

$$f_{\text{new}} = f_{\text{old}} \frac{p}{q}.$$

For matrices and data frames, resampling is performed independently for each column. Rows are interpreted as time samples and columns as individual signal channels.

The function uses resample internally. The resampling process includes interpolation, low-pass filtering, and decimation.

Value

A numeric vector, matrix, or data frame containing the resampled signal. The output type matches the input type. Column names are preserved for matrices and data frames.

Examples

```
# Numeric vector
signal <- sin(2 * pi * 5 * seq(0, 1, length.out = 400))
signal_resampled <- resample_signal(signal, p = 1, q = 4)

old.par <- par("mfrow", "mai")
par(mfrow = c(2, 1))
par(mai = c(0.9, 0.5, 0.3, 0.4))

plot(signal, type = "o")
plot(signal_resampled, type = "o")

par(old.par)

# Numeric matrix: samples in rows, channels in columns (256Hz, 10sec., 5 channels)
signal <- matrix(rnorm(2560 * 5), nrow = 2560, ncol = 5)
colnames(signal) <- paste0("channel_", seq_len(ncol(signal)))

# Resample to 64Hz
signal_64 <- resample_signal(signal, p = 1, q = 4)
```

```
dim(signal_64)

# Data frame
signal_df <- as.data.frame(signal)
signal_df_64 <- resample_signal(signal_df, p = 1, q = 4)
names(signal_df_64)
```

---

signal_to_bin *Convert multichannel signals to binary format*

Description

Converts a numeric matrix or data frame containing one or more signals to the binary format required by EMPI. Rows correspond to samples and columns to channels. Values are stored as 4-byte floating-point numbers using little-endian byte order.

For multichannel signals, samples are written in time order, with all channel values for a given time point stored consecutively: first all channels at `t = 0`, then all channels at `t =` Δt, and so on.

Usage

```
signal_to_bin(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 binary 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

A `raw` vector containing the binary representation of the signal. If `write_to_file = TRUE`, a `.bin` file is additionally created.

Note

The `.bin` files generated by this function are not intended for direct user manipulation. They are used internally by `empi_execute()`. The external program *Enhanced Matching Pursuit Implementation* (EMPI) 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 <- signal_to_bin(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",
  reverse_palette = 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 centres 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".
reverse_palette	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.
signal	Original signal.
sampling_frequency	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)

# '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,
  topk = NULL,
  sigma_divisor = NULL,
  verbose = FALSE
)
```

Arguments

atoms_dict	A matrix describing Gabor atoms (e.g. output of read_gabor_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	An object of class sig returned by read_csv_signals(), an object of class edf returned by read_edf_signals(), or an object of class wfdb returned by read_wfdb_signals().
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(), mp_core() or to mp_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_gabor_dict, mp_omp_execute omp_core mp_core

Examples

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

signal <- read_csv_signals(
  sig_file,
  col_names_in_csv = TRUE
)

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

# +------------------------------------------------------------+
# | Step 2: Read dictionary                                    |
# +------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3.xml",
  package = "MatchingPursuit"
)

atoms_dict <- read_gabor_dict(
  xml_file,
  sampling_frequency,
  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,
  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 <- mp_omp_execute(
  mode = "omp",
  dictionary = out_topk_atoms,
  signal = signal,
  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$signal,
  channel = 1,
  n_nonzero_coefs = 50
)

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

Index