Package: MatchingPursuit (via r-universe)

November 1, 2026

Type Package

Title Sparse Time Series Decomposition Using Matching Pursuit and Orthogonal Matching Pursuit

Version 1.4.0

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, imager, raster, graphics, grDevices, utils, digest, EGM, xml2, Matrix

Suggests knitr, rmarkdown, latex2exp, remotes

VignetteBuilder knitr

Depends R (>= 4.0.0)

License GPL-3

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

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Config/pak/sysreqs
cmake libfftw3-dev libgdal-dev gdal-bin libgeos-dev libglpk-dev make libicu-dev libjpeg-dev libpng-dev libtiff-dev libuv1-dev libxml2-dev libproj-dev libsqlite3-dev libx11-dev

Repository https://artur-gramacki.r-universe.dev

Date/Publication 2026-10-31 23:46:19 UTC

RemoteUrl https://github.com/artur-gramacki/matchingpursuit

RemoteRef HEAD

RemoteSha 618efbf2ef9ff92efcce3736b5990d904597d30c

Contents

as_sig . . . 3
clear_cache . . . 4
design_filters . . . 4
edf-methods . . . 6
eeg_montage . . . 8
empi_check . . . 10
empi_execute . . . 11
empi_install . . . 13
empi_locate . . . 14
gabor_atom . . . 14
gabor_projection_fft . . . 16
generate_xml_dict . . . 18
materialize_gabor . . . 20
mp-methods . . . 22
mp_core . . . 24
mp_core_gabor . . . 27
mp_omp_execute . . . 29
mp_omp_gabor_to_mp . . . 33
omp_core . . . 35
omp_core_gabor . . . 38
omp_reference . . . 40
read_atom_params . . . 43
read_csv_signals . . . 44
read_edf_params . . . 45
read_edf_signals . . . 46
read_empi_db . . . 47
read_gabor_dict . . . 48
read_wfdb_signals . . . 52
resample_signal . . . 53
sig-methods . . . 54
signal_to_bin . . . 56
tf_map . . . 57
wfdb-methods . . . 62

Index **65**

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 = NULL,
  notch_order = 2,
  lowpass = NULL,
  lowpass_order = 4,
  highpass = NULL,
  highpass_order = 4,
  bandpass = NULL,
  bandpass_order = 4,
  bandstop = NULL,
  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. `NULL` skips this filter.
`notch_order`	Positive integer specifying the notch filter order.
`lowpass`	Numeric value specifying the low-pass cutoff frequency in Hz. `NULL` skips this filter.
`lowpass_order`	Positive integer specifying the low-pass filter order.
`highpass`	Numeric value specifying the high-pass cutoff frequency in Hz. `NULL` skips this filter.
`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. `NULL` skips this filter.
`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. `NULL` skips this filter.
`bandstop_order`	Positive integer specifying the band-stop filter order.

Value

A list containing the designed Butterworth filter objects (or `NULL` for omitted 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 <- 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())
```

---

edf-methods *Methods for EDF objects*

Description

Methods for printing, summarizing, and plotting objects of class `"edf"`, together with a print method for objects returned by `summary()`.

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,
  ...
)

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

## S3 method for class 'edf'
summary(object, ...)

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

Arguments

x	An object of class "edf" or "summary.edf".
begin	Time point (in seconds) at which to start plotting. If NULL, plotting starts at the beginning of the signal.
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.
rainbow	Logical. If TRUE, individual channels are drawn in different colours.
bg_colour	Background colour.
txt_col	Colour of text elements.
zero_line	Logical. If TRUE, a horizontal line representing zero is displayed for each channel.
main	Plot title. If NULL, the EDF record name is used.
...	Additional arguments. Currently ignored.
object	An object of class "edf".

Details

Objects of class "edf" represent multi-channel signals imported from EDF files, together with sampling information, channel names, a time vector, and the source record name.

Value

`print.edf()` and `print.summary.edf()` return their input object invisibly.

`summary.edf()` returns an object of class `"summary.edf"` containing record information, signal dimensions, sampling information, channel names, and basic descriptive statistics for each channel.

`plot.edf()` is called for its side effect and returns no value.

See Also

`read_edf_signals`

Examples

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

print(x)
summary(x)

plot(
  x,
  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,
  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"
)
```

`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 EMPI program.
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

```
if (interactive()) {
file <- system.file("extdata", "sample1.csv", package = "MatchingPursuit")
signal <- read_csv_signals(file)

out_empi <- empi_execute(
  signal = signal
)

# Suppress standard output and standard error
out_empi <- empi_execute(
  signal = signal,
  ignore.stdout = TRUE,
  ignore.stderr = TRUE
)

# The default EMPI options have been modified; see the EMPI README.md for details.
# The '--full-atoms-in-signal' option restricts the decomposition to atoms
# fully contained within the analyzed signal.
```

```
# The decomposition results are saved to a SQLite database file.
out_empi <- empi_execute(
  signal = signal,
  empi_options = "-o local --full-atoms-in-signal -i 50 --gabor",
  write_to_file = TRUE,
  path = NULL,
  file_name = "my_decomposition.db"
)

plot(out_empi, freq_divide = 4)
}
```

`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

```
sys <- Sys.info()[["sysname"]]
mach <- Sys.info()[["machine"]]

if (sys %in% c("Windows", "Linux") ||
    (sys == "Darwin" && mach == "arm64")) {
  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 computation of projections onto Gabor atoms*

Description

Computes complex projection coefficients between one or more signals and a set of Gabor atoms using FFT-based frequency-domain operations.

Usage

```
gabor_projection_fft(block, signal, sigma_divisor = NULL)
```

Arguments

`block`	A matrix describing a single Gabor dictionary block, typically obtained as a subset of the output of `read_gabor_dict()`. It must contain at least the columns `time_sample`, `window_len`, `fft_size`, and `freq_bin`.
`signal`	A numeric vector, matrix, or data frame containing the signal(s) to be analyzed. For matrices and data frames, each column is treated as a separate signal channel.
`sigma_divisor`	Optional positive numeric value controlling the width of the Gaussian envelope. The envelope scale is calculated as `(window_len + 1) / sigma_divisor`. If `NULL`, a divisor of 3 is used.

Details

For each time position defined in `block`, the corresponding signal segment is multiplied by a normalized Gaussian envelope and transformed using the Fast Fourier Transform. Only Fourier coefficients corresponding to the frequencies specified in the dictionary are retained.

Atom supports may extend beyond the observed signal boundaries. In this case, samples outside the signal are treated as zero. The complete Gaussian envelope is normalized before boundary truncation; the part overlapping the observed signal is not renormalized.

The Gaussian envelope is constructed over the complete atom support and normalized to unit L2 norm before it is applied to the signal. If the atom support extends before the first signal sample

or beyond the last signal sample, only the overlapping signal samples contribute to the projection; values outside the observed signal are implicitly treated as zero.

Consequently, the visible part of a boundary-crossing atom is not renormalized. This preserves the normalization of the complete atom independently of its position relative to the signal boundaries.

Value

A list with two matrices:

`proj_mod_mtx`	Magnitudes of the complex projection coefficients. Rows correspond to atoms in `block` and columns to signal channels.
`fft_bin_mtx`	Complex Fourier coefficients corresponding to the Gabor frequencies. Rows correspond to atoms in `block` and columns to signal channels.

Note

This function is used internally by `mp_core_gabor()` and `omp_core_gabor()` to evaluate Gabor candidate projections. It is also exported to support advanced experiments and methodological testing.

See Also

`read_gabor_dict`, `mp_core_gabor`, `omp_core_gabor`

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 = xml_file,
  full_atoms_in_signal = FALSE,
  sampling_frequency = sampling_frequency,
  duration = duration,
  verbose = TRUE
)

out <- gabor_projection_fft(block, signal)

pmm <- out$proj_mod_mtx
scm <- out$fft_bin_mtx

head(scm)
head(pmm)

# Projection magnitudes are the moduli of the complex coefficients
head(Mod(scm))
```

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

Description

Generates an XML dictionary file containing a set of Gabor atoms.

Usage

```
generate_xml_dict(N, file, max_window_length = c("3N", "N-1"))
```

Arguments

`N`	Integer. Length of the analyzed signal in samples.
`file`	Character string. Path to the XML file that will be created. The output follows the MPTK dictionary XML structure and contains Gabor blocks.
`max_window_length`	Character string specifying the upper limit used when generating window lengths. `"3N"` uses the largest odd integer not exceeding `3*N`, whereas `"N-1"` limits the largest window length to the largest odd integer not exceeding `N-1`. The default is `"3N"`.

Details

The generated dictionary contains multiple Gabor blocks with logarithmically distributed window lengths. The smallest window length is fixed to 17 samples and the maximum window length is determined by `max_window_length`. It can be set to the largest odd integer not exceeding either `3*N` or `N-1`. The `3*N` upper limit is chosen heuristically.

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: determined by `max_window_length`.

3. Number of scales:
$$K = ceil(log2(N)) + 3$$
4. Logarithmic spacing of scales:
$$L_k = 17r^k$$
where
$$r = (L_{max}/17)^{1/(K-1)}$$
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

```
# Generate a dictionary for a 256-sample signal
xml_file <- tempfile(fileext = ".xml")

# Default setting: maximum window length based on 3*N
dict <- generate_xml_dict(
  N = 256,
  file = xml_file
)

dict

# Read the generated Gabor dictionary for a 2-second,
# 128-Hz signal (256 samples)
atoms_dict <- read_gabor_dict(
  xml_file,
  sampling_frequency = 128,
  duration = 2,
  full_atoms_in_signal = FALSE,
  verbose = TRUE
)

head(atoms_dict)
```

```
tail(atoms_dict)

# Alternative setting: limit the maximum window length to N-1
dict <- generate_xml_dict(
  N = 256,
  file = xml_file,
  max_window_length = "N-1"
)

dict
```

`materialize_gabor` *Materialize a Gabor dictionary with fixed or atom-specific phases*

Description

Materialize a Gabor dictionary with fixed or atom-specific phases

Usage

```
materialize_gabor(
  atoms_dict,
  sampling_frequency,
  duration,
  sigma_divisor = 3,
  phases = 0,
  phase_mode = c("grid", "per_atom")
)
```

Arguments

`atoms_dict`	Parameter matrix returned by read_gabor_dict().
`sampling_frequency`	Sampling frequency in Hz.
`duration`	Signal duration in seconds.
`sigma_divisor`	Positive Gaussian-envelope scale divisor; default 3.
`phases`	Numeric vector of phases in radians. In "grid" mode each phase is used for every atom. In "per_atom" mode there must be exactly one phase per dictionary row.
`phase_mode`	Either "grid" (default, original behavior) or "per_atom" (one phase per atom).

Details

A Gabor atom is treated as degenerate when its cosine-modulated Gaussian envelope is (numerically) zero before normalization. In particular, some combinations of frequency, phase, and window length can make the sampled cosine vanish at every sample. The implementation rejects an atom if its norm is less than `1e-10 * sqrt(sum(w^2))`, where `w` is the Gaussian envelope.

A second, distinct case occurs when the atom has no usable samples within the signal interval, for example when its entire support lies outside the interval. Such an atom is rejected by the visible-fragment norm check.

In `phase_mode = "grid"`, invalid atom columns are skipped; in `phase_mode = "per_atom"`, an error is raised to preserve the one-to-one correspondence between dictionary rows and output columns.

Value

Dense numeric matrix with one column per materialized atom. In "grid" mode, degenerate columns are removed, as in the original implementation. In "per_atom" mode, a degenerate atom causes an error so that column positions always match rows of atoms_dict.

Examples

```
fs <- 256
duration <- 2

xml_file <- system.file(
  "extdata", "one_block.xml",
  package = "MatchingPursuit"
)

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

D <- materialize_gabor(
  atoms_dict = atoms_dict,
  sampling_frequency = fs,
  duration = duration,
  phases = 0,
  phase_mode = "grid"
)

# Select four atoms with low frequencies, distributed across the signal
target_time <- seq(0.25, 1.75, length.out = 4)
target_freq <- c(4, 8, 12, 20)

idx <- sapply(seq_along(target_time), function(i) {
  which.min(
    abs(atoms_dict[, "time_sec"] - target_time[i]) +
    abs(atoms_dict[, "freq_hz"] - target_freq[i]) / 10
  )
})

matplot(
  (0:(nrow(D) - 1)) / fs,
   D[, idx, drop = FALSE],
```

```
  type = "l",
  lty = 1,
  xlab = "Time [s]",
  ylab = "Amplitude",
  main = "Four Gabor atoms"
 )
```

---

mp-methods *Methods for Matching Pursuit objects*

Description

Methods for printing, summarizing, and plotting objects of class "mp", together with a print method for objects returned by summary().

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,
  atom_centers = "crosses",
  display_grid = FALSE,
  color = "white",
  palette = "my custom palette",
  plot_signals = TRUE,
  verbose = FALSE,
  ...
)

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

## S3 method for class 'mp'
summary(object, ...)

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

Arguments

x	An object of class "mp" or "summary.mp". Objects of class "mp" are returned by empi_execute() or mp_omp_execute().
channel	Channel to process and display.
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 corresponds to the Nyquist frequency). 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.
atom_centers	"crosses", "numbers", or NULL. Determines how the centres of atoms (represented as so-called blobs) are marked on the T-F map (with small crosses, atom numbers, or no markers).
display_grid	Logical. If TRUE, grid lines are 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	Logical. If TRUE, the original and reconstructed signals are displayed.
verbose	Logical flag indicating whether progress information should be printed.
...	Additional arguments. Currently ignored.
object	An object of class "mp".

Details

Objects of class "mp" represent the result of a Matching Pursuit or Orthogonal Matching Pursuit decomposition. They contain the original signal, its reconstruction, parameters of the selected atoms, individual selected atom waveforms, the corresponding time vector, and sampling frequency.

The plotting method displays a time-frequency (T-F) map to visualize the decomposition. It is a wrapper around tf_map() with out_mode = "plot".

Value

print.mp() and print.summary.mp() return their input object invisibly.

summary.mp() returns an object of class "summary.mp" containing signal dimensions, sampling information, the total number of selected atoms, the number of selected atoms per channel, signal, residual, relative residual, and explained energy information.

plot.mp() is called for its side effect and returns no value.

See Also

tf_map, empi_execute, mp_omp_execute

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)

# Print and summarize the decomposition.
print(out_empi)
summary(out_empi)

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

## End(Not run)
```

---

mp_core *Implements the classical Matching Pursuit 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,
  sparse = FALSE,
  verbose = FALSE
)
```

Arguments

dictionary	A dictionary of atoms. Can be a numeric vector, matrix, data frame, or an object derived from class "Matrix". Atoms are assumed to be stored in columns. Dictionary atoms are internally normalized to unit L2 norm before decomposition. Therefore, atom selection is invariant to non-zero scaling of dictionary columns.

`signal` Can be a numeric vector, matrix, or data frame. 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. If `NULL`, channel 1 is used automatically for a single-channel signal. For multichannel signals, `channel` must be specified.

`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)))`. If `tol` is specified, `n_nonzero_coefs` does not control the stopping criterion. Note that in classical MP, the same dictionary atom may be selected more than once.

`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`.

`sparse` Logical; if `TRUE`, a dense dictionary is converted internally to a sparse matrix before the iterative MP calculations. If the supplied dictionary is already sparse, its sparse representation is preserved regardless of this argument. Defaults to `FALSE`.

`verbose` Logical; flag indicating whether progress information should be printed.

Details

This is a native R implementation of the classical MP algorithm supporting arbitrary matrix-based dictionaries. It provides direct access to the decomposition procedure and can be used independently of the Gabor-specific workflow. For high-performance Matching Pursuit decomposition using the external EMPI backend, see `empi_locate()`, `empi_install()`, `empi_check()`, and `empi_execute()`.

Dictionary atoms are normalized internally to unit L2 norm. For classical MP, the reduction in squared residual norm at iteration k is therefore theoretically equal to the squared MP coefficient. The former definition is used explicitly to maintain a consistent interpretation of `energy` across MP and OMP decompositions.

Value

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

`selected_atoms` Matrix of selected unit-L2-normalized dictionary atoms used in the reconstruction.

`signal` The analyzed signal channel returned as a numeric vector.

`reconstruction` The MP approximation of the signal.

`coefs` Numeric vector of estimated coefficients for selected atoms.

`energy` Numeric vector containing the reduction in squared residual norm associated with each pursuit iteration, $E_k = \|r_{k-1}\|_2^2 - \|r_k\|_2^2$.

`support` Integer vector of selected atom indices at every iteration.

`residual` Final residual vector.

`relative_residual_energy`
: Relative residual energy, $\|r_k\|_2^2 / \|x\|_2^2$, including the initial value before the first iteration and the value after each Matching Pursuit iteration. Values close to zero indicate a better reconstruction.

`n_iters`
: Number of iterations performed by the algorithm.

See Also

`omp_core`, `read_gabor_dict`, `mp_omp_execute`

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 <- mp_core(
  dictionary = dictionary,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 3,
  verbose = TRUE
)

fit$coefs
# [1] 6.274348 2.535423 1.870568

fit$support
# [1] 9 5 7

fit$relative_residual_energy
# [1] 1.0000000 0.2842283 0.1673489 0.1037303

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

mp_core_gabor *Matching Pursuit for a Parameterized Gabor Dictionary*

Description

Performs Matching Pursuit directly on a parameterized Gabor dictionary without materializing the complete dictionary as a time-domain matrix.

Usage

```
mp_core_gabor(
  atoms_dict,
  signal,
  channel = NULL,
  n_nonzero_coefs = NULL,
  tol = NULL,
  sigma_divisor = NULL,
  verbose = FALSE
)
```

Arguments

atoms_dict	Matrix-like object describing candidate Gabor atoms, typically returned by read_gabor_dict(). It must contain the columns block, time_sample, time_sec, freq_bin, freq_hz, window_len, and fft_size.
signal	An object of class "sig", "edf", or "wfdb".
channel	Integer index of the signal channel to analyze. If NULL, channel 1 is used for a single-channel signal; for a multichannel signal, the channel must be specified explicitly.
n_nonzero_coefs	Optional positive integer specifying the maximum number of selected atoms when tol = NULL. If both n_nonzero_coefs and tol are NULL, a default of 50 atoms is used.
tol	Optional stopping tolerance for the relative residual energy. If specified, it overrides n_nonzero_coefs.
sigma_divisor	Optional positive numeric value controlling the width of the Gaussian envelope. If NULL, a value of 3 is used.
verbose	Logical; if TRUE, progress information is printed.

Details

Candidate projections are evaluated blockwise using gabor_projection_fft(). For each candidate, the complex FFT coefficient provides the projections of the current residual onto its cosine and sine components. Precomputed 2 x 2 Gram matrix terms are then used to determine analytically the phase that maximizes the normalized projection.

The complete time-domain dictionary is never stored in memory. A full real-valued waveform is constructed only for the atom selected in the current MP iteration. The selected waveform is normalized to unit L2 norm over the part overlapping the observed signal before the classical MP residual update is applied.

Unlike `mp_core()`, which operates on a fixed matrix dictionary, `mp_core_gabor()` optimizes the phase of every Gabor candidate with respect to the current residual at every MP iteration.

Value

A list containing:

`selected_atoms`	Matrix whose columns are the selected unit-L2-normalized time-domain atoms.
`signal`	Analyzed signal channel as a numeric vector.
`reconstruction`	Signal reconstruction after the performed MP iterations.
`coefs`	MP coefficients of the selected atoms.
`energy`	Decrease in residual energy attributable to each MP iteration.
`support`	Row indices of the selected atoms in `atoms_dict`.
`residual`	Final residual signal.
`relative_residual_energy`	Relative residual energy at iteration 0 and after each completed MP iteration.
`n_iters`	Number of completed MP iterations.
`frequency`	Selected atom frequencies in Hz.
`phase`	Optimized selected atom phases in radians.
`scale`	Selected atom scales in seconds.
`position`	Selected atom centre positions in seconds.
`atom_begin`	Selected atom start positions in seconds.
`window_len`	Selected atom window lengths in seconds.
`atom_begin_sample`	Selected atom start positions in samples.
`window_len_samples`	Selected atom window lengths in samples.
`sampling_frequency`	Sampling frequency in Hz.
`sigma_divisor`	Gaussian envelope divisor actually used.
`n_candidates`	Number of candidate atoms in `atoms_dict`.

See Also

`read_gabor_dict`, `gabor_projection_fft`, `omp_core_gabor`, `mp_core`

Examples

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

xml_file <- system.file(
  "extdata",
  "one_block.xml",
  package = "MatchingPursuit"
)

signal <- read_csv_signals(signal_file, col_names_in_csv = FALSE)

atoms_dict <- read_gabor_dict(
  xml_file = xml_file,
  sampling_frequency = signal$sampling_frequency,
  duration = nrow(signal$signal) / signal$sampling_frequency,
  full_atoms_in_signal = TRUE
)

fit <- mp_core_gabor(
  atoms_dict = atoms_dict,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 10
)
```

---

mp_omp_execute *Matching Pursuit or Orthogonal Matching Pursuit 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 parameterized Gabor dictionary without materializing the complete dictionary as a time-domain matrix.

Usage

```
mp_omp_execute(
  signal,
  mode = NULL,
  dictionary = NULL,
  full_atoms_in_signal = FALSE,
  n_nonzero_coefs = NULL,
```

```
  tol = NULL,
  verbose = FALSE,
  progress = TRUE
)
```

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().
mode	"omp" or "mp". Specifies the algorithm to use for signal decomposition.
dictionary	Character string specifying the path to an XML file containing the dictionary specification. It can be generated using generate_xml_dict() or prepared manually by the user. If NULL, the XML file is generated internally by the function.
full_atoms_in_signal	Logical. If TRUE, only candidate atoms whose complete support lies within the signal are included in the dictionary. If FALSE, candidate atom supports may extend beyond the signal boundaries.
n_nonzero_coefs	Optional positive integer specifying the maximum number of selected atoms for each signal channel when tol = NULL. If both n_nonzero_coefs and tol are NULL, a default of 50 atoms is used.
tol	Optional stopping tolerance for the relative residual energy, between 0 and 1. If specified, it overrides n_nonzero_coefs.
verbose	Logical; if TRUE, progress information is printed during processing.
progress	Logical; if TRUE, displays messages indicating the start and completion of each signal channel. Defaults to TRUE.

Details

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

The XML dictionary specification is first processed by read_gabor_dict(). For each signal channel, the selected algorithm iteratively searches the parameterized Gabor dictionary for the atom best matching the current residual. Candidate projections are evaluated using FFT-based computations, and the phase of each candidate is optimized analytically.

In MP mode, mp_core_gabor() updates the residual by subtracting the contribution of the selected atom at each iteration. In OMP mode, omp_core_gabor() additionally recomputes the coefficients of all selected atoms by least squares using an incremental Cholesky factorization.

The complete time-domain dictionary is never stored in memory. Only the selected Gabor atoms are reconstructed and retained. The decomposition stops when the specified number of atoms has been selected or, if tol is provided, when the relative residual energy reaches the requested tolerance, subject to the algorithm's iteration limit.

The results from all channels are combined into an object of class "mp", which can be visualized using plot() and tf_map().

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 unit-L2-normalized 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, generate_xml_dict, omp_core_gabor, mp_core_gabor

Examples

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

signal <- read_csv_signals(
  file,
  col_names_in_csv = TRUE
)

# +------------------------------------------------------------+
# | Run Matching Pursuit (MP-R backend)                        |
# +------------------------------------------------------------+
fit_mp <- mp_omp_execute(
  mode = "mp",
```

```
  signal = signal,
  n_nonzero_coefs = 10,
  verbose = TRUE
)

plot(fit_mp, channel = 3)

# +-------------------------------------------------------------+
# | Run Orthogonal Matching Pursuit (OMP-R backend)             |
# +-------------------------------------------------------------+
fit_omp <- mp_omp_execute(
  mode = "omp",
  signal = signal,
  n_nonzero_coefs = 10,
  verbose = TRUE
)

plot(fit_omp, channel = 3)

# +-------------------------------------------------------------+
# | Stop decomposition using residual energy tolerance         |
# +-------------------------------------------------------------+
fit_mp_tol <- mp_omp_execute(
  mode = "mp",
  signal = signal,
  tol = 0.1,
  verbose = TRUE
)

plot(fit_mp_tol, channel = 3)

# +-------------------------------------------------------------+
# | Restrict Gabor atoms to the signal boundaries               |
# +-------------------------------------------------------------+
fit_mp_full <- mp_omp_execute(
  mode = "mp",
  signal = signal,
  full_atoms_in_signal = TRUE,
  n_nonzero_coefs = 10,
  verbose = TRUE
)

plot(fit_mp_full, channel = 3)

# +-------------------------------------------------------------+
# | Use an external XML dictionary specification                |
# +-------------------------------------------------------------+
xml_file <- system.file(
  "extdata",
  "sample3.xml",
  package = "MatchingPursuit"
)
```

```
fit_mp_xml <- mp_omp_execute(
  mode = "mp",
  signal = signal,
  dictionary = xml_file,
  n_nonzero_coefs = 10,
  verbose = TRUE
)

plot(fit_mp_xml, channel = 3)
```

`mp_omp_gabor_to_mp` *Convert a Gabor MP/OMP core result to an mp object*

Description

Converts the output of `mp_core_gabor()` or `omp_core_gabor()` into a standardized object of class `"mp"`.

Usage

```
mp_omp_gabor_to_mp(core, sampling_frequency)
```

Arguments

`core`	A list returned by `mp_core_gabor()` or `omp_core_gabor()`. It must contain the selected atoms, signal, coefficients, atom energies, and Gabor parameters.
`sampling_frequency`	Numeric scalar specifying the sampling frequency of the analyzed signal in Hz.

Details

The function transforms a single-channel Gabor decomposition result into the object structure used by `mp_omp_execute()`. The channel identifier in the resulting object is always set to 1. It preserves the original signal, reconstructed signal, selected atoms, atom energies, and time-frequency parameters, including frequency, phase, scale, and position. The reconstruction is obtained by multiplying the matrix of selected atoms by their corresponding coefficients.

The resulting object can be used with functions that accept objects of class `"mp"`, including time-frequency visualization tools such as `tf_map()`.

This function does not perform signal decomposition. It only converts an existing decomposition result into the standard `"mp"` format.

The function is intended exclusively for the Gabor-specific core implementations. It is not designed for the general-purpose `mp_core()` and `omp_core()` functions.

Value

An object of class "mp" containing:

atoms Data frame containing the channel identifier, atom number, energy, envelope type, frequency, phase, scale, and position of each selected Gabor atom.

signal Numeric matrix containing the original signal (samples $\times$ 1).

reconstruction Numeric matrix containing the reconstructed signal (samples $\times$ 1).

selected_atoms List containing the matrix of selected Gabor atoms for the analyzed channel.

time Numeric vector containing the time coordinates of signal samples in seconds.

sampling_frequency Sampling frequency in Hz.

See Also

mp_core_gabor, omp_core_gabor, mp_omp_execute, tf_map

Examples

```
# Generate a Gabor dictionary
sampling_frequency <- 128
duration <- 1
t <- seq(0, duration - 1 / sampling_frequency, by = 1 / sampling_frequency)
y <- sin(2 * pi * 20 * t) + 0.5 * sin(2 * pi * 50 * t)
sig <- as_sig(y, sampling_frequency)

xml_file <- tempfile(fileext = ".xml")
generate_xml_dict(length(y), xml_file)

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

# MP decomposition
fit_mp <- mp_core_gabor(
  atoms_dict = atoms_dict,
  signal = sig,
  channel = 1,
  n_nonzero_coefs = 20,
  verbose = FALSE
)

out_mp <- mp_omp_gabor_to_mp(fit_mp, sampling_frequency)

# OMP decomposition
fit_omp <- omp_core_gabor(
  atoms_dict = atoms_dict,
  signal = sig,
  channel = 1,
  n_nonzero_coefs = 20,
```

```
  verbose = FALSE
)

out_omp <- mp_omp_gabor_to_mp(fit_omp, sampling_frequency)

# Verify the resulting object
class(out_omp)
head(out_omp$atoms)

# Verify reconstruction consistency
max(abs(out_omp$reconstruction[, 1] - fit_omp$reconstruction))
```

`omp_core` *Implements Orthogonal Matching Pursuit algorithm*

Description

Performs Orthogonal Matching Pursuit (OMP) to obtain a sparse representation of a signal using a dictionary of candidate atoms.

Usage

```
omp_core(
  dictionary,
  signal,
  channel = NULL,
  n_nonzero_coefs = NULL,
  tol = NULL,
  sparse = FALSE,
  verbose = FALSE
)
```

Arguments

`dictionary`	A dictionary of atoms. Can be a numeric vector, matrix, data frame, or an object derived from class `"Matrix"`. Atoms are assumed to be stored in columns. Dictionary atoms are internally normalized to unit L2 norm before decomposition. Therefore, atom selection is invariant to non-zero scaling of dictionary columns.
`signal`	Can be a numeric vector, matrix, or data frame. 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. If `NULL`, channel 1 is used automatically for a single-channel signal. For multichannel signals, `channel` must be specified.
`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)))`. If `tol` is specified, `n_nonzero_coefs` does not control the stopping criterion.

`tol` Optional stopping tolerance for the relative residual energy, defined as

$$\|r_k\|_2^2 / \|x\|_2^2.$$

The algorithm stops when this value is less than or equal to `tol`. If specified, `tol` overrides `n_nonzero_coefs`.

`sparse` Logical; if `TRUE`, a dense dictionary is converted internally to a sparse matrix before OMP decomposition. This can substantially improve performance for dictionaries containing many zero values. If the supplied dictionary is already sparse, its sparse representation is preserved regardless of this argument. Defaults to `FALSE`.

`verbose` Logical; flag indicating whether progress information should be printed.

Details

Unlike classical Matching Pursuit, OMP recomputes the coefficients of all previously selected atoms at each iteration by solving a least-squares problem. This makes the residual orthogonal to the subspace spanned by the selected atoms and generally provides a more accurate approximation for a given number of atoms.

The least-squares problem is solved efficiently using incremental Cholesky factorization.

At each iteration, the atom with the largest absolute correlation with the current residual is selected. All coefficients associated with the selected atoms are then recomputed simultaneously by least squares, and the residual is updated.

For OMP, the quantity stored in `energy` is not calculated from the final coefficient of an individual atom. Selected atoms can be strongly correlated, and therefore quantities such as `coefs^2 * colSums(selected_atoms^2)` are not additive contributions to the energy of the reconstruction.

Instead, the energy associated with the atom selected at iteration k is defined as the reduction in residual energy:

$$E_k = \|r_{k-1}\|_2^2 - \|r_k\|_2^2.$$

Thus, `energy[k]` quantifies the reduction in residual energy associated with the k-th OMP iteration, after adding a new atom and re-estimating all active coefficients.

$$\sum_k E_k = \|x\|_2^2 - \|r_K\|_2^2.$$

Value

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

`selected_atoms` Matrix of selected unit-L2-normalized dictionary atoms used in the reconstruction.

`signal`	The analyzed signal channel returned as a numeric vector.
`reconstruction`	OMP reconstruction of the signal
`coefs`	Final least-squares coefficients corresponding to `selected_atoms`.
`energy`	Numeric vector containing the reduction in residual energy produced at each OMP iteration, $\|r_{k-1}\|_2^2 - \|r_k\|_2^2$.
`support`	Indices of the selected atoms in the input dictionary, in selection order.
`residual`	Final residual vector.
`relative_residual_energy`	Relative residual energy after initialization and after each OMP iteration.
`n_iters`	Number of OMP iterations performed.

See Also

`mp_core`, `omp_core_gabor`, `mp_omp_execute`

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,
  verbose = TRUE
)

fit$coefs
# [1] 5.282278 2.637693 2.195920

fit$support
# [1] 9 5 7
```

```
fit$relative_residual_energy
# [1] 1.00000000 0.28422833 0.16609350 0.08795047

# For Gabor-specific decomposition, see omp_core_gabor()
# and mp_omp_execute().
```

`omp_core_gabor` *Orthogonal Matching Pursuit for a parameterized Gabor dictionary*

Description

Performs Orthogonal Matching Pursuit directly on a parameterized Gabor dictionary without materializing the complete dictionary as a time-domain matrix.

Usage

```
omp_core_gabor(
  atoms_dict,
  signal,
  channel = NULL,
  n_nonzero_coefs = NULL,
  tol = NULL,
  sigma_divisor = NULL,
  verbose = FALSE
)
```

Arguments

`atoms_dict`	Matrix-like object describing candidate Gabor atoms, typically returned by `read_gabor_dict()`. It must contain the columns `block`, `time_sample`, `time_sec`, `freq_bin`, `freq_hz`, `window_len`, and `fft_size`.
`signal`	An object of class `"sig"`, `"edf"`, or `"wfdb"`.
`channel`	Integer index of the signal channel to analyze. If `NULL`, channel 1 is used for a single-channel signal; for a multichannel signal, the channel must be specified explicitly.
`n_nonzero_coefs`	Optional positive integer specifying the maximum number of selected atoms when `tol = NULL`. If both `n_nonzero_coefs` and `tol` are `NULL`, a default of 50 atoms is used.
`tol`	Optional stopping tolerance for the relative residual energy. If specified, it overrides `n_nonzero_coefs`.
`sigma_divisor`	Optional positive numeric value controlling the width of the Gaussian envelope. If `NULL`, a value of 3 is used.
`verbose`	Logical; if `TRUE`, progress information is printed.

Details

Candidate selection uses the same parameterized Gabor search as `mp_core_gabor()`. At each OMP iteration, all candidates are evaluated against the current residual and the phase of the newly selected atom is optimized analytically.

Once an atom has been selected, its phase remains fixed. The coefficients of all selected atoms are then recomputed jointly by least squares, as in classical OMP. The least-squares problem is updated efficiently using an incremental Cholesky factorization.

The complete Gabor dictionary is never stored as a time-domain matrix. Only the selected unit-L2-normalized atoms are materialized and retained in the active dictionary.

A single row of `atoms_dict` represents a fixed time position, frequency, and window length, while phase is optimized continuously. Consequently, the same row index may in principle be selected more than once with different phases, provided that the resulting time-domain atoms are linearly independent. Near-linear dependence is detected through the Cholesky update; a rejected candidate is skipped and the next-best candidate is tested.

Value

A list containing:

`selected_atoms`	Matrix whose columns are the selected unit-L2-normalized time-domain atoms.
`signal`	Analyzed signal channel as a numeric vector.
`reconstruction`	OMP reconstruction of the signal.
`coefs`	Final least-squares coefficients corresponding to `selected_atoms`.
`energy`	Reduction in residual energy produced at each OMP iteration.
`support`	Row indices of the selected atoms in `atoms_dict`, in selection order.
`residual`	Final residual signal.
`relative_residual_energy`	Relative residual energy at iteration 0 and after each completed OMP iteration.
`n_iters`	Number of completed OMP iterations.
`frequency`	Selected atom frequencies in Hz.
`phase`	Optimized phases fixed at the time each atom was selected.
`scale`	Selected atom scales in seconds.
`position`	Selected atom centre positions in seconds.
`atom_begin`	Selected atom start positions in seconds.
`window_len`	Selected atom window lengths in seconds.
`atom_begin_sample`	Selected atom start positions in samples.
`window_len_samples`	Selected atom window lengths in samples.
`sampling_frequency`	Sampling frequency in Hz.
`sigma_divisor`	Gaussian envelope divisor actually used.
`n_candidates`	Number of parameter rows in `atoms_dict`.

See Also

mp_core_gabor, read_gabor_dict, gabor_projection_fft, omp_core

Examples

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

xml_file <- system.file(
  "extdata",
  "one_block.xml",
  package = "MatchingPursuit"
)

signal <- read_csv_signals(signal_file, col_names_in_csv = FALSE)

atoms_dict <- read_gabor_dict(
  xml_file = xml_file,
  sampling_frequency = signal$sampling_frequency,
  duration = nrow(signal$signal) / signal$sampling_frequency,
  full_atoms_in_signal = TRUE
)

fit <- omp_core_gabor(
  atoms_dict = atoms_dict,
  signal = signal,
  channel = 1,
  n_nonzero_coefs = 10
)
```

---

omp_reference *Reference implementation of Orthogonal Matching Pursuit*

Description

A straightforward reference implementation of the Orthogonal Matching Pursuit (OMP) algorithm that closely follows its mathematical formulation. The function is intended for reference and illustrative purposes and does not use computational optimizations. The least-squares problem is solved explicitly using the normal-equation formula.

Usage

```
omp_reference(
  dictionary,
  signal,
  n_nonzero_coefs = NULL,
```

```
  tol = NULL,
  verbose = FALSE
)
```

Arguments

dictionary	A dictionary of atoms. Can be a numeric vector, matrix, or data frame. Atoms are assumed to be stored in columns. Dictionary atoms are internally normalized to unit L2 norm before decomposition. Therefore, atom selection is invariant to arbitrary scaling of dictionary columns.
signal	A numeric vector.
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	Optional stopping tolerance for the relative residual energy, defined as $\|r_k\|_2^2 / \|x\|_2^2$. The algorithm stops when this value is less than or equal to tol. If specified, tol overrides n_nonzero_coefs.
verbose	Logical. If TRUE, information about the progress of the algorithm and the orthogonality of the residual is printed.

Details

This implementation is intended for small illustrative examples rather than large-scale computations. In particular, when verbose = TRUE, detailed information is printed at every iteration, including residual and reconstruction vectors. For large signals or a large number of iterations, this may produce a substantial amount of console output.

This implementation operates on a single signal represented by a numeric vector. The signal argument must therefore be a numeric vector, not a matrix containing multiple signals or channels.

Because the least-squares problem is solved explicitly through the normal equations, the implementation may fail for singular or nearly singular selected subdictionaries and is not intended for numerically demanding applications.

Dictionary atoms are internally normalized to unit Euclidean norm before decomposition. The returned coefficients are provided both for the normalized working dictionary and for the original dictionary scaling.

Value

A list containing:

selected_atoms	Indices of the atoms selected by OMP.
coefficients_normalized_dict	Coefficient vector corresponding to the normalized working dictionary.

`coefficients_original_dict`
: Coefficient vector corresponding to the original (unnormalized) dictionary.

`reconstruction_normalized_dict`
: Signal reconstruction obtained using the normalized working dictionary.

`reconstruction_original_dict`
: Signal reconstruction obtained using the original dictionary.

`residual_normalized_dict`
: Residual corresponding to the reconstruction based on the normalized working dictionary.

`residual_original_dict`
: Residual corresponding to the reconstruction based on the original dictionary.

`normalized_reconstruction_error_normalized_dict`
: Normalized reconstruction error (NRE) for the normalized working dictionary, defined as $\|y - \hat{y}\|_2 / \|y\|_2$.

`normalized_reconstruction_error_original_dict`
: Normalized reconstruction error (NRE) for the original dictionary, defined as $\|y - \hat{y}\|_2 / \|y\|_2$.

`orthogonality`
: A list containing, for each OMP iteration, the inner products between the current residual and all atoms selected up to that iteration. Values should be numerically close to zero.

`residual_energy`
: Residual energy after each OMP iteration, defined as $\|r_k\|_2^2$.

`iterations`
: Number of OMP iterations performed.

See Also

`omp_core`, `mp_omp_execute`

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 <- c(4, 2, 3, 5, 1)

out <- omp_reference(
  dictionary = dictionary,
  signal = signal,
  n_nonzero_coefs = 3,
  verbose = TRUE
)
```

```
out
```

`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), ]
```

`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,
  full_atoms_in_signal = 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 candidate atoms.
full_atoms_in_signal	Logical. If TRUE, only candidate atoms whose complete support lies within the signal are included in the dictionary. If FALSE, candidate atom supports may extend beyond the signal boundaries.

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="31"/>
   <param name="windowShift" value="2"/>
   <param name="fftSize" value="64"/>
 </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.

The treatment of atoms near signal boundaries is controlled by full_atoms_in_signal.

If TRUE, only atoms fully contained within the signal support are generated. In this case, atom start positions satisfy

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

where t is the atom start position, N is the signal length, and L is the window length. If the signal is shorter than the window length, no time positions are generated for that block.

If FALSE, atom centres are allowed at positions throughout the signal, and the support of an atom may extend beyond the signal boundaries.

Value

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

block	Block identifier from the XML file.
time_sample	Start position of the atom support (in samples).
time_sec	Start position of the atom support (in seconds).
freq_bin	Frequency bin index.
freq_hz	Frequency in Hertz.
window_len	Window length of the atom support, in samples.
fft_size	FFT size used to define the frequency grid.

Usage in sparse decomposition workflow

The output of read_gabor_dict() is a parameterized description of the Gabor time-frequency grid. It serves as input to mp_core_gabor() and omp_core_gabor(), which perform iterative signal decomposition directly on the parameterized dictionary.

At each iteration, candidate Gabor atoms are evaluated against the current residual using FFT-based projections and analytical phase optimization. Only the selected atoms are reconstructed in the time domain, avoiding materialization of the complete dictionary.

The higher-level mp_omp_execute() function handles dictionary preparation and multi-channel decomposition internally.

EMPI compatibility

XML dictionary definitions exported by EMPI can be read directly by this function. Additional XML elements not used by read_gabor_dict() are ignored. The argument full_atoms_in_signal controls the boundary convention corresponding to the EMPI --full-atoms-in-signal option.

For full details on the EMPI options and their behavior, see the EMPI documentation in README.md.

See Also

generate_xml_dict, mp_core_gabor, omp_core_gabor, mp_omp_execute

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
)

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

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

# +------------------------------------------------------------+
# | Step 3: Compare boundary conventions                       |
# +------------------------------------------------------------+
# Generate only atoms whose complete support lies within
# the signal boundaries.
atoms_full <- read_gabor_dict(
  xml_file = xml_file,
  sampling_frequency = sampling_frequency,
  duration = duration,
  full_atoms_in_signal = TRUE,
  verbose = TRUE
)

# Allow atom support to extend beyond the signal boundaries.
# Atom centres still remain within the signal.
atoms_overstep <- read_gabor_dict(
  xml_file = xml_file,
  sampling_frequency = sampling_frequency,
  duration = duration,
  full_atoms_in_signal = FALSE,
  verbose = TRUE
)

# Allowing boundary overstep increases the number of dictionary atoms.
nrow(atoms_full)
nrow(atoms_overstep)

# With full_atoms_in_signal = TRUE, atom start positions
# are always non-negative.
range(atoms_full[, "time_sample"])

# With full_atoms_in_signal = FALSE, atoms centred near the beginning
# of the signal may have negative start positions.
```

```
range(atoms_overstep[, "time_sample"])
```

`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)
```

---

sig-methods *Methods for signal objects*

Description

Methods for printing, summarizing, and plotting objects of class `"sig"`, together with a print method for objects returned by `summary()`.

Usage

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

## S3 method for class 'sig'
summary(object, ...)

## S3 method for class 'summary.sig'
print(x, ...)

## S3 method for class 'sig'
plot(
```

```
  x,
  begin = NULL,
  end = NULL,
  panel_height = NULL,
  zero_line = TRUE,
  main = NULL,
  mar = c(4, 7, 2, 1),
  col = "black",
  lwd = 1,
  lty = 1,
  ...
)
```

Arguments

x	An object of class "sig" or "summary.sig".
...	Additional arguments. For plot.sig(), these are passed to plot(); otherwise they are currently ignored.
object	An object of class "sig".
begin	Beginning of the displayed interval in seconds.
end	End of the displayed interval in seconds.
panel_height	Vertical distance between signal channels. If NULL, it is determined automatically from the signal amplitude.
zero_line	Logical. If TRUE, draw a horizontal reference line for each channel.
main	Optional plot title.
mar	Numeric vector of length four specifying the plot margins in the form c(bottom, left, top, right).
col	Colour used to draw the signal traces.
lwd	Line width used to draw the signal traces.
lty	Line type used to draw the signal traces.

Details

Objects of class "sig" represent a single- or multi-channel signal together with its sampling frequency and corresponding time vector.

Value

print.sig() and print.summary.sig() return their input object invisibly.

summary.sig() returns an object of class "summary.sig" containing signal dimensions, sampling information, channel names, and basic descriptive statistics for each channel.

plot.sig() is called for its side effect and returns x invisibly.

See Also

read_csv_signals, as_sig

Examples

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

x <- read_csv_signals(
  file,
  col_names_in_csv = TRUE
)

print(x)
summary(x)

plot(x)
plot(x, mar = c(4, 12, 2, 1))
plot(x, mar = c(4, 12, 2, 1), col = "blue")
plot(x, col = "red", lwd = 2, lty = 4)
plot(x, begin = 0, end = 1, main = "Signal")
```

---

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",
  display_scale = "linear",
  db_range = 20,
  freq_divide = NULL,
  increase_factor = 4,
  shortening_factor_x = 2,
  shortening_factor_y = 2,
```

```
  atom_centers = "crosses",
  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 = FALSE
)
```

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" (default), "linear", or "energy". Controls the amplitude assigned to each Gaussian blob. "sqrt" uses sqrt(energy * sampling_frequency); "linear" uses energy * sampling_frequency; "energy" normalizes the integral of each untruncated blob over the whole time-frequency plane to its assigned atom energy.
display_scale	"linear" (default) or "db". Applies dB scaling after all blobs have been summed, for plotting only. Does not modify the returned tf_map or saved RData matrices.
db_range	Positive number specifying the displayed dynamic range in dB (default 20); used only if display_scale = "db".
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	Positive factor narrowing the Gaussian blobs along time; the default of 2 is a visualization choice, not an estimated physical uncertainty.
shortening_factor_y	Positive factor narrowing the Gaussian blobs along frequency; the default of 2 is a visualization choice, not an estimated physical uncertainty.
atom_centers	"crosses", "numbers", or NULL. Determines how the centres of atoms (represented as so-called blobs) are marked on the T-F map (with small crosses, atom numbers, or no markers).

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.

Details

The function also computes basic reconstruction-quality measures for the selected channel, including signal energy, residual energy, and the proportion of explained signal energy.

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.

`signal_energy` Energy of the original signal for the selected channel.

`reconstruction_energy`
Energy of the reconstructed signal for the selected channel.

`residual_energy`
Energy of the reconstruction residual.

`explained_energy`
Proportion of the original signal energy explained by the reconstruction, `1 - sum((signal - reconstruction)^2) / sum(signal^2)`.

`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.

Best practices

- For an exploratory, visually balanced map, start with `mode = "sqrt", display_scale = "linear"`. This reduces the contrast between weak and strong atoms; the map is not an energy-normalized time-frequency density.
- To represent assigned atom energies by the integrals of Gaussian blobs, use `mode = "energy"`. The normalization holds for each complete blob; cropping at time or frequency boundaries can remove part of its displayed integral. For MP/OMP, the interpretation of the supplied atom energies depends on how they were computed upstream.
- Use `display_scale = "db"` to make weak structures more visible. It computes `10 * log10(tf_map / max(tf_map))` and clips values below `-db_range` dB. Try 20 dB for a selective display and 40 dB to reveal more weak structures. `db_range` has no effect with `display_scale = "linear"`.
- `mode = "linear"` is a legacy amplitude rule proportional to atom energy. At fixed shortening factors it differs from `mode = "energy"` only by a global multiplier.
- When comparing MP, OMP, signals, or recordings, keep the mode, Gaussian widths, frequency range, and palette fixed. The dB display is normalized separately to each map's maximum, so colors across different maps do not indicate equal absolute intensities.
- The returned `tf_map` contains the summed, untransformed Gaussian map for the chosen `mode`; it is not a spectrogram. Saved RData matrices are additionally resized and scaled to [0, 1].

Examples

```
# Read signal and XML dictionary
file <- system.file(
  "extdata", "sample1.csv",
  package = "MatchingPursuit"
)

signal <- read_csv_signals(file)

xml_file <- system.file(
  "extdata", "sample1.xml",
  package = "MatchingPursuit"
)

# Perform OMP decomposition
sample1_empi_out <- mp_omp_execute(
  mode = "omp",
  signal = signal,
  dictionary = xml_file,
  n_nonzero_coefs = 16,
  verbose = TRUE
)

# Example 1: Default visualization
# Square-root amplitude scaling and linear display.
out <- tf_map(
  x = sample1_empi_out,
  channel = 1,
  mode = "sqrt",
  display_scale = "linear",
  freq_divide = 4
)

# Example 2: Automatic frequency range
# The displayed frequency range is determined from the
# highest frequency among the selected atoms.
out <- tf_map(
  x = sample1_empi_out,
  channel = 1,
  mode = "sqrt",
  display_scale = "linear",
  freq_divide = NULL
)

# Example 3: Energy-normalized Gaussian blobs
# With a linear display, strong atoms dominate;
# weak atoms may be less visible.
out <- tf_map(
  x = sample1_empi_out,
  channel = 1,
  mode = "energy",
  display_scale = "linear",
```

```
  freq_divide = 4
)

# Example 4: Logarithmic display
# Enhances the visibility of weaker structures.
# Values below -20 dB are mapped to the background color.
out <- tf_map(
  x = sample1_empi_out,
  channel = 1,
  mode = "energy",
  display_scale = "db",
  db_range = 20,
  freq_divide = 4
)

# Example 5: Wider dynamic range and alternative palette
# Larger shortening factors produce more compact blobs.
# Atom numbers identify the selected dictionary elements.
out <- tf_map(
  x = sample1_empi_out,
  channel = 1,
  mode = "energy",
  display_scale = "db",
  db_range = 40,
  freq_divide = 4,
  shortening_factor_x = 4,
  shortening_factor_y = 4,
  palette = "Inferno",
  reverse_palette = FALSE,
  atom_centers = "numbers"
)
```

`wfdb-methods` *Methods for WFDB objects*

Description

Methods for printing, summarizing, and plotting objects of class `"wfdb"`, together with a print method for objects returned by `summary()`.

Usage

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

```
  zero_line = FALSE,
  ...
)

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

## S3 method for class 'wfdb'
summary(object, ...)

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

Arguments

x	An object of class "wfdb" or "summary.wfdb".
begin	Time point (in seconds) at which to start plotting. If NULL, plotting starts at the beginning of the signal.
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 lead panel (in mV).
small_squares	Logical. If TRUE, the small ECG-paper grid (0.04 s by 0.1 mV) is displayed in addition to the large grid.
zero_line	Logical. If TRUE, a horizontal line representing 0 mV is displayed for each lead.
...	Additional arguments. Currently ignored.
object	An object of class "wfdb".

Details

Objects of class "wfdb" represent multi-channel physiological signals imported from WFDB records, together with their sampling frequency, time vector, lead names, and record name.

WFDB (WaveForm DataBase) is a widely used format and software framework 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, containing binary signal samples, and .hea, a header file describing how to interpret the data. Additional annotation files, such as .atr, may also be present and may contain beat labels or rhythm annotations.

The plotting method is designed primarily for ECG signals and displays individual leads in a layout resembling standard ECG paper. The small grid corresponds to 0.04 s by 0.1 mV and the large grid to 0.20 s by 0.5 mV.

Value

print.wfdb() and print.summary.wfdb() return their input object invisibly.

`summary.wfdb()` returns an object of class `"summary.wfdb"` containing record information, signal dimensions, sampling information, lead names, and basic descriptive statistics for each lead.

`plot.wfdb()` is called for its side effect and returns no value.

See Also

`read_wfdb_signals`

Examples

```
# ECG data from the PTB-XL database
file <- system.file("extdata", "00001_lr.hea", package = "MatchingPursuit")
x <- read_wfdb_signals(file)

print(x)
summary(x)

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

Index

as_sig, 3, *55*

clear_cache, 4

design_filters, 4

edf-methods, 6
eeg_montage, 8
empi_check, 10, *12–14*
empi_execute, 11, *11*, *13, 14*, *24*
empi_install, *11, 12*, 13, *14*
empi_locate, *11–13*, 14

gabor_atom, 14
gabor_projection_fft, 16, *28*, *40*
generate_xml_dict, 18, *31*, *50*

materialize_gabor, 20
mp-methods, 22
mp_core, 24, *28*, *37*
mp_core_gabor, *17*, 27, *31*, *34*, *40*, *50*
mp_omp_execute, *24*, *26*, 29, *34*, *37*, *42*, *50*
mp_omp_gabor_to_mp, 33

omp_core, *26*, 35, *40*, *42*
omp_core_gabor, *17*, *28*, *31*, *34*, *37*, 38, *50*
omp_reference, 40

plot.edf *(*edf-methods*)*, 6
plot.mp, *11–14*
plot.mp *(*mp-methods*)*, 22
plot.sig *(*sig-methods*)*, 54
plot.wfdb *(*wfdb-methods*)*, 62
print.edf *(*edf-methods*)*, 6
print.mp *(*mp-methods*)*, 22
print.sig *(*sig-methods*)*, 54
print.summary.edf *(*edf-methods*)*, 6
print.summary.mp *(*mp-methods*)*, 22
print.summary.sig *(*sig-methods*)*, 54
print.summary.wfdb *(*wfdb-methods*)*, 62
print.wfdb *(*wfdb-methods*)*, 62

read_atom_params, 43
read_csv_signals, *3*, 44, *55*
read_edf_params, 45
read_edf_signals, *3*, *8*, 46
read_empi_db, 47
read_gabor_dict, *17*, *19*, *26*, *28*, *31*, *40*, 48
read_wfdb_signals, *3*, 52, *64*
resample, *53*
resample_signal, 53

sig-methods, 54
signal_to_bin, 56
summary.edf *(*edf-methods*)*, 6
summary.mp *(*mp-methods*)*, 22
summary.sig *(*sig-methods*)*, 54
summary.wfdb *(*wfdb-methods*)*, 62

tf_map, *24*, *34*, 57

wfdb-methods, 62