Package 'MatchingPursuit'

May 7, 2026

Type Package

Title Processing Time Series Data Using the Matching Pursuit Algorithm

Version 1.0.1

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

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

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

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

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

Suggests knitr, rmarkdown, latex2exp, remotes

VignetteBuilder knitr

Depends R (>= 3.5.0)

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Repository CRAN

Date/Publication 2026-04-14 09:10:15 UTC

Contents

`atom.params` *Reading the atom parameters*

Description

Returns a data frame with atom parameters read from a SQLite file.

Usage

```
atom.params(db.file)
```

Arguments

`db.file`	The SQLite file created after executing the `empi.execute()` function.

Value

Data frame with all the atom parameters saved in a given SQLite file. The file can be generated using the `empi.execute()` function.

Examples

```
# The file contains data with 18 channels.
file <- system.file("extdata", "EEG.db", package = "MatchingPursuit")
out <- atom.params(file)
out[which(out$channel_id == 1), ]
out[which(out$channel_id == 18), ]

# This file contains data with only 1 channel.
```

```
file <- system.file("extdata", "sample1.db", package = "MatchingPursuit")
out <- atom.params(file)
out
```

`clear.cache` *Clear MatchingPursuit Cache*

Description

Deletes all files in the MatchingPursuit cache directory.

Usage

```
clear.cache()
```

Value

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

Examples

```
if (interactive()) {
  clear.cache()
}
```

`eeg.montage` *Performs bipolar, reference or average EEG montage*

Description

An EEG montage refers to the specific arrangement of EEG electrodes and the way their signals are displayed relative to each other during the interpretation of an electroencephalogram. The same EEG recording can look very different depending on the montage use. The function implements the three most frequently used montage methods in practice, i.e. 1) Bipolar Montage, 2) Referential (Monopolar) Montage and 3) Average Reference Montage.

Usage

```
eeg.montage(
  eeg.data,
  montage.type = c("average", "reference", "bipolar"),
  ref.channel = NULL,
  bipolar.pairs = NULL
)
```

Arguments

`eeg.data`	Must be a data frame: rows = samples, columns = channels.The data frame must have correct column names (channel names).
`montage.type`	A character string representing 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 find out what names the individual channels have in the analysed EEG set, it is worth executing the `read.edf.params()` function.

Value

A data frame with final montage (rows = samples, columns = channels).

Examples

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

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(signal, montage.type = c("bipolar"), bipolar.pairs = pairs)
signal.ref.mont <- eeg.montage(signal, montage.type = c("reference"), ref.channel = "O1")
signal.avg.mont <- eeg.montage(signal, montage.type = c("average"))

head(signal.bip.mont)
head(signal.ref.mont)
head(signal.avg.mont)
```

---

empi.check *Checks if EMPI external software is installed*

Description

The EMPI program is installed using the `empi.install()` function and is stored in the cache directory. This function checks whether the EMPI program is still there (the user has free access to the cache directory and can, for example, delete it at any time).

Usage

```
empi.check()
```

Value

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

Examples

```
empi.check()
```

---

empi.execute *Launches the empi program*

Description

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

Usage

```
empi.execute(
  signal,
  empi.options = NULL,
  write.to.file = FALSE,
  path = NULL,
  file.name = NULL
)
```

Arguments

signal	List returned from read.csv.signals() function. The list stores the signal in a data frame along with its sampling frequency. The data should have logical column names (channel names). The elements of the list must be named "signal" and "sampling.rate".
empi.options	If NULL, the EMPI program runs with "-o local --gabor -i 50" parameters. Otherwise, user can specify any command-line options. See README.md file after downloading the EMPI program using empi.install() function.
write.to.file	If TRUE, a SQLite database file will be created and saved in the path directory or, (if path=NULL), in the cache directory. This file stores the results of signal decomposition using the MP algorithm.
path	Directory in which to save the SQLite database file. If it is NULL, the file will be saved in the cache directory.
file.name	The name of the file to generate if write.to.file=TRUE.

Details

The EMPI program (source and binary files for various 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. If write.to.file=TRUE is specified, the results are also written to a SQLite file on disk in the path directory.

Examples

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

empi.out <- empi.execute (
  signal = signal,
  empi.options = NULL,
  write.to.file = FALSE,
  path = NULL,
  file.name = NULL
)
## End(Not run)
```

empi.install *Installs the EMPI external program*

Description

Downloads **Enhanced Matching Pursuit Implementation** external program (or EMPI for short) binary 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 x64, macOS arm64), downloads the appropriate archive from the official repository, verifies its integrity using a checksum, and extracts it.

Value

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

Examples

```
if (interactive()) {
  empi.install()
}
```

empi.locate *Get required external software localization*

Description

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

Usage

```
empi.locate()
```

Value

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

Examples

```
empi.locate()
```

`empi2tf` *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 created map can be: 1) displayed on the screen, 2) saved in `.png` file, or 3) saved as an `.RData` object.

Usage

```
empi2tf(
  db.file = NULL,
  db.list = NULL,
  channel,
  mode = "sqrt",
  freq.divide = 1,
  increase.factor = 1,
  shortening.factor.x = 2,
  shortening.factor.y = 2,
  display.crosses = TRUE,
  display.atom.numbers = FALSE,
  display.grid = FALSE,
  crosses.color = "white",
  palette = "my custom palette",
  rev = TRUE,
  out.mode = "plot",
  path = NULL,
  file.name = NULL,
  size = c(512, 512),
  draw.ellipses = FALSE,
  plot.signals = TRUE,
  write.atoms = FALSE
)
```

Arguments

`db.file`	The SQLite file created after executing the `empi.execute()` function. In this case, the `db.list` parameter must be `NULL`.
`db.list`	The list created after executing the `empi.execute()` function. In this case, the `db.file` parameter must be `NULL`.
`channel`	Channel from the SQLite file to process.

mode "sqrt", "log", or "linear". It 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 in the T-F map should be decreased. For example, if the sampling frequency is f=256Hz, the maximum frequency in the T-F map will be f/2/freq.divide (f/2 is the Nyquist rule).

increase.factor
Factor of increasing the number of pixels in the f-axis, the most sensible are non-negative integers (e.g. 2, 4, 5, 8).

shortening.factor.x
Usually, for better visualization of atoms, a value of 2 will be appropriate.

shortening.factor.y
Usually, for better visualization of atoms, a value of 2 will be appropriate.

display.crosses
Whether small crosses should be displayed in the canters of atoms.

display.atom.numbers
Whether atom numbers should be displayed in the canters of atoms.

display.grid Whether to draw grid lines.

crosses.color Colour of small crosses.

palette Palette from the list returned by hcl.pals() function or the string "my custom palette".

rev rev param in 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 file file.name (as png file).
- "RData" - saves the T-F map of size in the file.name (as R's matrix), resampling is performed using the function imager::resize() function.
- "RData2" - saves the T-F map of size in the file.name (as R's matrix), resampling is performed using the function using 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 \code{RData} file (if \code{out.mode="RData"} or \code{out.mode="RData2")

size png file size in pixels (if out.mode="file") or size of the T-F matrix (if out.mode="RData" of out.mode="RData2").

draw.ellipses Only for testing. User can set it to TRUE to see the effect. Works properly 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 into Atoms.pdf file (to the cache directory or user specified one - depending on path variable)

Value

Depending on the `out.mode` parameter the function returns:

- Time-Frequency map plotted on the screen
- Time-Frequency map saved in a .png file
- Time-Frequency map saved as .RData file

Regardless of the above, the function returns the following:

- all the Gabor functions
- reconstructed signal
- original signal
- sampling frequency
- grid size in t axis
- grid size in f axis
- epoch size in samples
- length of the signal in seconds
- time-frequency map
- time-frequency map after resampling (if `out.mode="RData"` or if `out.mode="RData2"`, otherwise, `NULL` is returned)
- channel number processed
- frequency divide

Examples

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

out <- empi2tf(
  db.file = file,
  channel = 1,
  mode = "sqrt",
  freq.divide = 4,
  increase.factor= 4,
  display.crosses = TRUE,
  display.atom.numbers = FALSE,
  out.mode = "plot",
)
```

`filters.coeff` *A wrapper function for* `signal::butter()` *function*

Description

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

Usage

```
filters.coeff(
  fs = 256,
  notch = c(49, 51),
  notch.order = 2,
  lowpass = 30,
  lowpass.order = 4,
  highpass = 1,
  highpass.order = 4,
  bandpass = c(0.5, 40),
  bandpass.order = 4,
  bandstop = c(0.5, 40),
  bandstop.order = 4
)
```

Arguments

`fs`	Sampling rate.
`notch`	Vector of two frequencies for notch filter.
`notch.order`	Notch filter order.
`lowpass`	Low-pass filter frequency.
`lowpass.order`	Low-pass filter order.
`highpass`	High-pass filter frequency.
`highpass.order`	High-pass filter order.
`bandpass`	Vector of two frequencies for band-pass filter.
`bandpass.order`	Band-pass filter order.
`bandstop`	Vector of two frequencies for band-stop filter.
`bandstop.order`	Band-stop filter order.

Value

List with parameters of individual filters.

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
out <- read.edf.signals(file, resampling = FALSE)
signal <- out$signals
sampling.rate <- out$sampling.rate

fc <- filters.coeff(
  fs = sampling.rate,
  notch = c(49, 51),
  lowpass = 40,
  highpass = 1,
  bandpass = c(0.5, 40),
  bandstop = c(10, 50)
)

print(fc)

signal::freqz(fc$notch, Fs = sampling.rate)
signal::freqz(fc$lowpass, Fs = sampling.rate)
signal::freqz(fc$highpass, Fs = sampling.rate)
signal::freqz(fc$bandpass, Fs = sampling.rate)
signal::freqz(fc$bandstop, Fs = sampling.rate)

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

signal.filt <- signal

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

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

---

gabor.fun *Gabor function implementation*

Description

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

Usage

```
gabor.fun(
  number.of.samples,
  sampling.frequency,
  mean,
  phase,
  sigma,
  frequency,
  normalization = TRUE
)
```

Arguments

number.of.samples	How many samples should the generated atom consist of?
sampling.frequency	Sampling frequency.
mean	Time position.
phase	Phase.
sigma	Scale / width of the Gaussian window.
frequency	Frequency of the sinusoid.
normalization	If TRUE, norm of the generated atom equals 1.

Value

List of 4 vectors with cosine, gauss, gabor and time waveforms of size number.of.samples.

Examples

```
number.of.samples <- 512
sampling.frequency <- 256.0
mean <- 1
phase <- pi
sigma <- 0.5
frequency <- 5.0
normalization = TRUE

out <- gabor.fun(
  number.of.samples,
  sampling.frequency,
  mean,
  phase,
  sigma,
  frequency,
  normalization
)

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

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

`read.csv.signals` *Reads and checks if the csv file has the correct structure*

Description

Reads and checks if the csv file has the correct structure

Usage

```
read.csv.signals(file, col.names = NULL)
```

Arguments

`file`	File to be read and check. The first line of the file must contain two numbers: the sampling rate in Hz (`freq`) and the signal length in seconds (`sec`). The function checks whether the file actually contains `round(freq*sec)` samples. The two numbers must by separated by one or more whitespace characters.
`col.names`	Vector with column names. If not specified, default names will be created.

Value

A list is returned with: 1) data frame where rows = samples for all channels, columns = channels, 2) sampling rate.

Examples

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

# The first line of the file must contain two numbers:
# a) the sampling rate 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.rate
```

read.edf.params *Reads a selected EDF or EDF+ file and returns signal parameters*

Description

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

Usage

```
read.edf.params(file)
```

Arguments

file	The path to the EDF / EDF+ file to be read.

Value

A data frame is returned containing the most basic parameters of the EDF / EDF+ file.

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 all signals data*

Description

The function reads a selected EDF or EDF+ file. Also resampling can be done (upsampling or downsampling).

Usage

```
read.edf.signals(
  file,
  resampling = FALSE,
  f.new = NULL,
  from = NULL,
  to = NULL,
  verbose = FALSE
)
```

Arguments

file	The path to the EDF / EDF+ file to be read.
resampling	If TRUE the frequency of all signals will be upsampling or downsampling, depending on the actual sampling rate of subsequent channel.
f.new	A new frequency (used for upsampling or downsampling).
from	Loading a signal from (given as a second).
to	Loading a signal to (given as a second).
verbose	Flag to print out progress information.

Details

If resampling=TRUE, the frequency of all signals will be upsampled or downsampled, depending on the actual sampling rate of the individual channels and the set value of the f.new parameter. The EDF standard assumes that each channel can be sampled at a different rate. Therefore, it may happen that some channels are upsampled and others are downsampled. The function does not provide the functionality to independently change the sampling rate for each channel.

Value

A list is returned with: 1) data frame with all signals stored in the given edf file, 2) complete result returned by the edf::read.edf() function, 3) sampling rate of the data after possible resampling (upsampled or downsampled), 4) time stamps of the data after possible resampling (upsampled or downsampled).

Examples

```
file <- system.file("extdata", "EEG.edf", package = "MatchingPursuit")
sigs1  <- read.edf.signals(file, resampling = FALSE)

lapply(sigs1, class)
sigs1$sampling.rate

sigs2 <- read.edf.signals(file, resampling = TRUE, f.new = 128, verbose = TRUE)

lapply(sigs2, class)
sigs2$sampling.rate
```

---

read.empi.db.file — *Reads data from a SQLite file created by the Matching Pursuit algorithm*

Description

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

Usage

```
read.empi.db.file(db.file)
```

Arguments

db.file	SQLite file.

Value

- Detailed parameters of all the generated atoms
- Original input signal(s)
- Reconstructed signal(s), as the sum of generated atoms
- Generated Gabor atoms
- time stamps
- sampling rate

Examples

```
## Not run:
file <- system.file("extdata", "EEG.db", package = "MatchingPursuit")
out <- read.empi.db.file(file)

n.channels <- ncol(out$original.signal)
original.signal <- out$original.signal
reconstruction <- out$reconstruction
t <- out$t
f <- out$f

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

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

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

len <- length(original.signal[, 1])
lab <- seq(t[1], t[len] + 1 / f, 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)

## End(Not run)
```

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

Description

Saves the given data (signals) in binary form. Input signal(s) must be a data frame: rows = samples for all channels, columns = channels. The function is used internally in the `empi.execute()` function. The binary data are floating-point values in the byte order of the current machine (no byte-order conversion is performed). For multichannel signals, first come the samples for all channels at `t=0`, then for all channels at `t=`Δ`t` and so forth. In other words, the signal should be written in column-major order (rows = channels, columns = samples).

Usage

```
sig2bin(data, write.to.file = FALSE)
```

Arguments

`data`	Data frame with the input signal(s).
`write.to.file`	If `TRUE` the bin file will be created and saved in the cache directory.

Value

Input signal returned as the `raw`. If `write.to.file=TRUE`, the `.bin` file will additionally be created and saved in the current directory.

Note

The user does not work directly with `.bin` files. Binary files are used only in the `empi.execute()` function. The external program (*Enhanced Matching Pursuit Implementation*, or EMPI for short) executed inside this function requires binary data as input. Moreover, the ability to convert text files to binary form may be useful if someone wants to work with EMPI independently of the R environment.

Examples

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

signal.bin <- sig2bin(data = out$signal, write.to.file = FALSE)

# We have 3 channels. The first 4 time points.
head(out$signal, 4)

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

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

Index