Package: heartbeatr (via r-universe)

May 17, 2026

Type Package

Title A Workflow to Process Data Collected with PULSE Systems

Version 1.0.0

Description Given one or multiple paths to files produced by a PULSE multi-channel or a PULSE one-channel system (<https://electricblue.eu/pulse>) from a single experiment: [1] check pulse files for inconsistencies and read/merge all data, [2] split across time windows, [3] interpolate and smooth to optimize the dataset, [4] compute the heart rate frequency for each channel/window, and [5] facilitate quality control, summarising and plotting. Heart rate frequency is calculated using the Automatic Multi-scale Peak Detection algorithm proposed by Felix Scholkmann and team. For more details see Scholkmann et al (2012) <doi:10.3390/a5040588>. Check original code at <https://github.com/ig248/pyampd>. ElectricBlue is a non-profit technology transfer startup creating research-oriented solutions for the scientific community (<https://electricblue.eu>).

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Imports av, cli, dplyr, ggplot2, lubridate, magrittr, purrr, readr, stringr, tibble, tidyr, transformr

Depends R (>= 3.5.0)

RoxygenNote 7.3.3

VignetteBuilder knitr

NeedsCompilation no

Author Rui Seabra [aut, cre] (ORCID: <https://orcid.org/0000-0002-0240-3992>), Fernando Lima [aut] (ORCID: <https://orcid.org/0000-0001-9575-9834>)

Maintainer Rui Seabra <ruisea@gmail.com>

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libavfilter-dev libicu-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev libx11-dev

Repository https://ruiseabra.r-universe.dev

Date/Publication 2025-09-18 09:00:02 UTC

RemoteUrl https://github.com/cran/heartbeatr

RemoteRef HEAD

RemoteSha 780301036a963b7a109a2724490ac311320005fd

Contents

find_peaks . . . 2
is.pulse . . . 3
PULSE . . . 3
PULSE_by_chunks . . . 9
pulse_choose_keep . . . 12
pulse_data . . . 13
pulse_doublecheck . . . 14
pulse_example . . . 16
pulse_find_peaks_all_channels . . . 16
pulse_find_peaks_one_channel . . . 18
pulse_halve . . . 19
pulse_heart . . . 20
pulse_interpolate . . . 21
pulse_normalize . . . 23
pulse_optimize . . . 25
pulse_plot . . . 27
pulse_plot_one . . . 29
pulse_plot_raw . . . 30
pulse_read . . . 32
pulse_smooth . . . 33
pulse_split . . . 34
pulse_summarise . . . 37

Index **40**

`find_peaks` *Find peaks of waves in raw PULSE data*

Description

`heartbeatr-package` Find peaks of waves in raw PULSE data

Usage

```
find_peaks(t, y)
```

Arguments

t	time
y	val

Value

A numeric vector indicating the indexes where peaks were detected.

---

is.pulse *heartbeatr utility function*

Description

heartbeatr-package utility function

Usage

```
is.pulse(path)
```

Arguments

path	file path

Value

A logical indicating if the path supplied corresponds to a valid PULSE file or not.

PULSE *Process PULSE data from a single experiment (STEPS 1-6)*

Description

ALL STEPS EXECUTED SEQUENTIALLY

- step 1 – pulse_read()
- step 2 – pulse_split()
- step 3 – pulse_optimize()
- step 4 – pulse_heart()
- step 5 – pulse_doublecheck()
- step 6 – pulse_choose_keep()
- extra step – pulse_normalize()
- extra step – pulse_summarise()
- visualization – pulse_plot() and pulse_plot_raw()

This is a wrapper function that provides a shortcut to running all 6 steps of the PULSE multi-channel data processing pipeline in sequence, namely `pulse_read()` » `pulse_split()` » `pulse_optimize()` » `pulse_heart()` » `pulse_doublecheck()` » `pulse_choose_keep()`.

Please note that the `heartbeatr` package is designed specifically for PULSE systems commercialized by the non-profit co-op ElectricBlue (https://electricblue.eu/pulse) and is likely to fail if data from any other system is used as input without matching file formatting.

`PULSE()` takes a vector of `paths` to PULSE csv files produced by a PULSE system during **a single experiment** (either multi-channel or one-channel, but never both at the same time) and automatically computes the heartbeat frequencies in all target channels across use-defined time windows. The entire workflow may take less than 5 minutes to run on a small dataset (a few hours of data) if `params` are chosen with speed in mind and the code is run on a modern machine. Conversely, large datasets (spanning several days) may take hours or even days to run. In extreme situations, datasets may be too large for the machine to handle (due to memory limitations), and it may be better to process batches at a time (check `PULSE_by_chunks()` and consider implementing a parallel computing strategy).

Usage

```
PULSE(
  paths,
  window_width_secs = 30,
  window_shift_secs = 60,
  min_data_points = 0.8,
  interpolation_freq = 40,
  bandwidth = 0.2,
  doublecheck = TRUE,
  lim_n = 3,
  lim_sd = 0.75,
  raw_v_smoothed = TRUE,
  correct = TRUE,
  discard_channels = NULL,
  keep_raw_data = TRUE,
  subset = 0,
  subset_seed = NULL,
  subset_reindex = FALSE,
  process_large = FALSE,
  show_progress = TRUE,
  max_dataset_size = 20
)
```

Arguments

`paths`	character vectors, containing file paths to CSV files produced by a PULSE system during a single experiment.
`window_width_secs`	numeric, in seconds, defaults to `30`; the width of the time windows over which heart rate frequency will be computed.

`window_shift_secs`
: numeric, in seconds, defaults to `60`; by how much each subsequent window is shifted from the preceding one.

`min_data_points`
: numeric, defaults to `0.8`; decimal from 0 to 1, used as a threshold to discard incomplete windows where data is missing (e.g., if the sampling frequency is `20` and `window_width_secs = 30`, each window should include `600` data points, and so if `min_data_points = 0.8`, windows with less than `600 * 0.8 = 480` data points will be rejected).

`interpolation_freq`
: numeric, defautls to `40`; value expressing the frequency (in Hz) to which PULSE data should be interpolated. Can be set to `0` (zero) or any value equal or greater than `40` (the default). If set to zero, no interpolation is performed.

`bandwidth`
: numeric, defaults to `0.2`; the bandwidth for the Kernel Regression Smoother. If equal to `0` (zero) no smoothing is applied. Normally kept low (`0.1 - 0.3`) so that only very high frequency noise is removed, but can be pushed up all the way to 1 or above (especially when the heartbeat rate is expected to be slow, as is typical of oysters, but double check the resulting data). Type `?ksmooth` for additional info.

`doublecheck`
: logical, defaults to `TRUE`; should `pulse_doublecheck()` be used? (it is rare, but there are instances when it should be disabled).

`lim_n`
: numeric, defaults to `3`; minimum number of peaks detected in each time window for it to be considered a "keep".

`lim_sd`
: numeric, defaults to `0.75`; maximum value for the sd of the time intervals between each peak detected for it to be considered a "keep"

`raw_v_smoothed`
: logical, defaults to `TRUE`; indicates whether or not to also compute heart rates before applying smoothing; this will increase the quality of the output but also double the processing time.

`correct`
: logical, defaults to `TRUE`; if `FALSE`, data points with `hz` values likely double the real value are flagged **BUT NOT CORRECTED**. If `TRUE`, `hz` (as well as `data`, `n`, `sd` and `ci`) are corrected accordingly. Note that the correction is not reversible!

`discard_channels`
: character vectors, containing the names of channels to be discarded from the analysis. `discard_channels` is forced to lowercase, but other than that, the **exact** names must be provided. Discarding unused channels can greatly speed the workflow!

`keep_raw_data`
: logical, defaults to `TRUE`; If set to `FALSE`, `$data` is set to `FALSE` (i.e., raw data is discarded), dramatically reducing the amount of disk space required to store the final output (usually, by two orders of magnitude). HOWEVER, note that it won't be possible to use `pulse_plot_raw()` anymore!

`subset`
: numerical, defaults to `0`; the number of time windows to keep from the entire dataset (or the number of entries to reject if set to a negative value); smaller subsets make the entire processing quicker and facilitate the execution of trial runs to optimize parameter selection before processing the entire dataset.

`subset_seed` numerical, defaults to `NULL`; only used if `subset` is different from `0`; `subset_seed` controls the seed used when extracting a subset of the available data; if set to `NULL`, a random seed is selected, resulting in rows being selected randomly; alternatively, the user can set a specific seed in order to always select the same rows (important when the goal is to compare the impact of different parameter combinations using the exact same data points).

`subset_reindex` logical, defaults to `FALSE`; only used if `subset` is different from `0`; after extracting a subset of the available data, should rows be re-indexed (i.e., `.$i` made fully sequential); re-indexed rows make using `pulse_plot_raw()` easier, but row identity doesn't match anymore with row identity before subsetting.

`process_large` logical, defaults to `FALSE`; If set to `FALSE`, if the dataset used as input is large (i.e., combined file size greater than 20 MB, which is equivalent to three files each with a full hour of PULSE data), `PULSE` will not process the data and instead suggest the use of `PULSE_by_chunks()`, which is designed to handle large datasets; If set to `TRUE`, `PULSE` will proceed with the attempt to process the dataset, but the system's memory may become overloaded and R may never finish the job.

`show_progress` logical, defaults to `FALSE`. If set to `TRUE`, progress messages will be provided.

`max_dataset_size`
numeric, defaults to `21`. Corresponds to the maximum combined size (in Mb) that the dataset contained by the files in `paths` can be when `process_large` is set to `FALSE`. If that is the case, data processing will be aborted with a message explaining the remedies possible. This is a fail-safe to prevent `PULSE` from being asked to process a dataset that is larger than the user's machine can handle, a situation that typically leads to a stall (R doesn't fail, it just keeps trying without any progress being made). A conservative value of `21` will allow only a little more than 3 hours-worth of data to be processed (a PULSE csv file with 1 hour of data typically takes up to 7 Mb). If the machine has a large amount of RAM available, a higher value can be used. Alternatively, consider using the function `PULSE_by_chunks()` instead.

Value

A tibble with nrows = (number of channels) * (number of windows in `pulse_data_split`) and 13 columns:

- `i`, the order of each time window
- `smoothed`, logical flagging smoothed data
- `id`, PULSE channel IDs
- `time`, time at the center of each time window
- `data`, a list of tibbles with raw PULSE data for each combination of channel and window, with columns `time`, `val` and `peak` (`TRUE` in rows corresponding to wave peaks)
- `hz`, heartbeat rate estimate (in Hz)
- `n`, number of wave peaks identified
- `sd`, standard deviation of the intervals between wave peaks
- `ci`, confidence interval (hz ± ci)

- keep, logical indicating whether data points meet N and SD criteria
- d_r, ratio of consecutive asymmetric peaks
- d_f, logical flagging data points where heart beat frequency is likely double the real value

One experiment

The heartbeatr workflow must be applied to a single experiment each time. By *experiment* we mean a collection of PULSE data where all the relevant parameters are invariant, including (but not limited):

- the version of the firmware installed in the PULSE device (multi-channel or one-channel)
- the names of all channels (including unused channels)
- the frequency at which data was captured

Note also that even if two PULSE systems have been used in the same *scientific experiment*, data from each device must be processed independently, and only merged at the end. There's no drawback in doing so, it just is important to understand that that's how data must be processed by the heartbeatr-package.

Normalizing and summarising data

Both pulse_normalize() and pulse_summarise() aren't included in PULSE() because they aren't essential for the PULSE data processing pipeline and the choosing of values for their parameters require an initial look at the data. However, it is very often crucial to normalize the heart rate estimates produced so that comparisons across individuals can more reliably be made, and it also often important to manage the amount of data points produced before running statistical analyses on the data to avoid oversampling, meaning that users should consider running the output from PULSE() though both these functions before considering the data as fully processed and ready for subsequent analysis. Check both functions for additional details on their role on the entire processing pipeline (?pulse_normalize and ?pulse_summarise).

Additional details

Check the help files of the underlying functions to obtain additional details about each of the steps implemented under PULSE(), namely:

- pulse_read() describes constraints to the type of files that can be read with the heartbeatr-package and explains how time zones are handled.
- pulse_split() provides important advice on how to set window_width_secs and window_shift_secs, what to expect when lower/higher values are used, and explains how easily to run the heartbeatr-package with parallel computing.
- pulse_optimize() explains in detail how the optimization process (interpolation + smoothing) behaves and how it impacts the performance of the analysis.
- pulse_heart() outlines the algorithm used to identify peaks in the heart beat wave data and some of its limitations.
- pulse_doublecheck() explains the method used to detect situations when the algorithm's processing resulted in an heart beat frequency double the real value.
- pulse_choose_keep() selects the best estimates when raw_v_smoothed = TRUE and classifies data points as keep or reject.

Also check

- pulse_normalize() for important info about individual variations on baseline heart rate.
- pulse_summarise() for important info about oversampling and strategies to handle that.
- PULSE_by_chunks() for processing large datasets.

BPM

To convert to Beats Per Minute (bpm), simply multiply hz and ci by 60.

See Also

- approx() is used by pulse_interpolate() for the linear interpolation of PULSE data
- ksmooth() is used by pulse_smooth() for the kernel smoothing of PULSE data
- pulse_read(), pulse_split(), pulse_optimize(), pulse_heart(), pulse_doublecheck() and pulse_choose_keep() are the functions used in the complete heartbeatr processing workflow
- pulse_normalize() and pulse_summarise() are important post-processing functions
- pulse_plot() and pulse_plot_raw() can be used to inspect the processed data

Examples

```
## Begin prepare data ----
paths <- pulse_example()
chn <- paste0("c", formatC(1:10, width = 2, flag = "0"))
## End prepare data ----

# Execute the entire PULSE data processing pipeline with only one call
PULSE(
paths,
 discard_channels = chn[-8],
 raw_v_smoothed   = FALSE,
 show_progress    = FALSE
 )

# Equivalent to...
x <- pulse_read(paths)
multi <- x$multi
x$data <- x$data[,c("time", "c08")]
x <- pulse_split(x)
x <- pulse_optimize(x, raw_v_smoothed = FALSE, multi = multi)
x <- pulse_heart(x)
x <- pulse_doublecheck(x)
x <- pulse_choose_keep(x)
x
```

PULSE_by_chunks *Process PULSE data file by file (STEPS 1-6)*

Description

This function runs PULSE() file by file, instead of attempting to read all files at once. This is required when datasets are too large (more than 20-30 files), as otherwise the system may become stuck due to the amount of data that needs to be kept in the memory. Because the results of processing data for each hourly file in the dataset are saved to a job_folder, PULSE_by_chunks() has the added benefit of allowing the entire job to be stopped and resumed, facilitating the advance in the processing even if a crash occurs.

Usage

```
PULSE_by_chunks(
  folder,
  allow_dir_create = FALSE,
  chunks = 2,
  bind_data = TRUE,
  window_width_secs = 30,
  window_shift_secs = 60,
  min_data_points = 0.8,
  interpolation_freq = 40,
  bandwidth = 0.2,
  doublecheck = TRUE,
  lim_n = 3,
  lim_sd = 0.75,
  raw_v_smoothed = TRUE,
  correct = TRUE,
  discard_channels = NULL,
  keep_raw_data = TRUE,
  show_progress = TRUE
)
```

Arguments

folder	the path to a folder where several PULSE files are stored
allow_dir_create	logical, defaults to FALSE. Only when set to TRUE does PULSE_by_chunks() actually do anything. This is to force the user to accept that a job_folder will be created inside of the folder supplied - without this folder PULSE_by_chunks() cannot operate. It is STRONGLY advised to maintain a copy of the dataset being processed to avoid any inadvertent data loss. By setting allow_dir_create to TRUEthe user is taking responsibility for the management of their files.
chunks	numeric, defaults to 2. Corresponds to the number of files processed at once during each for cycle; higher numbers result in a quicker and more efficient

operation, but shouldn't be set too high, as otherwise the system may become overwhelmed once more (which is what `PULSE_by_chunks()` is designed to avoid).

`bind_data` logical, defaults to `TRUE`. If set to `TRUE`, after processing all chunks, `PULSE_by_chunks()` will try to read all files in the job_folder and return a single unified tibble with all data. Please be aware that there's a possibility that if the dataset is very large, the machine may become overwhelmed and crash due to lack of memory (still, all files stored in the job_folder will remain intact, and code may be written to analyze data also in chunks). If set to `FALSE`, `PULSE_by_chunks()` will return nothing after completing the processing of all files in the dataset, and the user must instead manually handle the reading and collating of all processed data in the job_folder.

`window_width_secs`
numeric, in seconds, defaults to `30`; the width of the time windows over which heart rate frequency will be computed.

`window_shift_secs`
numeric, in seconds, defaults to `60`; by how much each subsequent window is shifted from the preceding one.

`min_data_points`
numeric, defaults to `0.8`; decimal from 0 to 1, used as a threshold to discard incomplete windows where data is missing (e.g., if the sampling frequency is `20` and `window_width_secs = 30`, each window should include `600` data points, and so if `min_data_points = 0.8`, windows with less than `600 * 0.8 = 480` data points will be rejected).

`interpolation_freq`
numeric, defautls to `40`; value expressing the frequency (in Hz) to which PULSE data should be interpolated. Can be set to `0` (zero) or any value equal or greater than `40` (the default). If set to zero, no interpolation is performed.

`bandwidth` numeric, defaults to `0.2`; the bandwidth for the Kernel Regression Smoother. If equal to `0` (zero) no smoothing is applied. Normally kept low (`0.1 - 0.3`) so that only very high frequency noise is removed, but can be pushed up all the way to 1 or above (especially when the heartbeat rate is expected to be slow, as is typical of oysters, but double check the resulting data). Type `?ksmooth` for additional info.

`doublecheck` logical, defaults to `TRUE`; should `pulse_doublecheck()` be used? (it is rare, but there are instances when it should be disabled).

`lim_n` numeric, defaults to `3`; minimum number of peaks detected in each time window for it to be considered a "keep".

`lim_sd` numeric, defaults to `0.75`; maximum value for the sd of the time intervals between each peak detected for it to be considered a "keep"

`raw_v_smoothed` logical, defaults to `TRUE`; indicates whether or not to also compute heart rates before applying smoothing; this will increase the quality of the output but also double the processing time.

`correct` logical, defaults to `TRUE`; if `FALSE`, data points with `hz` values likely double the real value are flagged **BUT NOT CORRECTED**. If `TRUE`, `hz` (as well as `data`, `n`, `sd` and `ci`) are corrected accordingly. Note that the correction is not reversible!

`discard_channels`
: character vectors, containing the names of channels to be discarded from the analysis. `discard_channels` is forced to lowercase, but other than that, the **exact** names must be provided. Discarding unused channels can greatly speed the workflow!

`keep_raw_data`
: logical, defaults to `TRUE`; If set to `FALSE`, `$data` is set to `FALSE` (i.e., raw data is discarded), dramatically reducing the amount of disk space required to store the final output (usually, by two orders of magnitude). HOWEVER, note that it won't be possible to use `pulse_plot_raw()` anymore!

`show_progress`
: logical, defaults to `FALSE`. If set to `TRUE`, progress messages will be provided.

Value

A tibble with nrows = (number of channels) * (number of windows in `pulse_data_split`) and 13 columns:

- `i`, the order of each time window
- `smoothed`, logical flagging smoothed data
- `id`, PULSE channel IDs
- `time`, time at the center of each time window
- `data`, a list of tibbles with raw PULSE data for each combination of channel and window, with columns `time`, `val` and `peak` (`TRUE` in rows corresponding to wave peaks)
- `hz`, heartbeat rate estimate (in Hz)
- `n`, number of wave peaks identified
- `sd`, standard deviation of the intervals between wave peaks
- `ci`, confidence interval (hz ± ci)
- `keep`, logical indicating whether data points meet N and SD criteria
- `d_r`, ratio of consecutive asymmetric peaks
- `d_f`, logical flagging data points where heart beat frequency is likely double the real value

See Also

- `PULSE()` for all the relevant information about the the processing of `PULSE` data

Examples

```
##
```

pulse_choose_keep *(STEP 6) Choose the best heart beat frequency estimate from among two estimates derived from raw and smoothed data*

Description

- step 1 – pulse_read()
- step 2 – pulse_split()
- step 3 – pulse_optimize()
- step 4 – pulse_heart()
- step 5 – pulse_doublecheck()
- –>> **step 6 –** pulse_choose_keep() <<–

When running pulse_optimize() or PULSE() with raw_v_smoothed = TRUE, two estimates are generated for each data point, and pulse_choose_keep is used to automatically select the best one (based on N and SD levels set by the user). NOTE: if supplied with input data generated using raw_v_smoothed = FALSE, pulse_choose_keep outputs the same data, unchanged.

Usage

```
pulse_choose_keep(heart_rates, lim_n = 3, lim_sd = 0.75)
```

Arguments

heart_rates	the output from pulse_heart()
lim_n	numeric, defaults to 3; minimum number of peaks detected in each time window for it to be considered a "keep".
lim_sd	numeric, defaults to 0.75; maximum value for the sd of the time intervals between each peak detected for it to be considered a "keep"

Value

A tibble with the same structure as the input, but now with only one estimate for each combination of id and time (the one that was deemed better).

See Also

- pulse_read(), pulse_split(), pulse_optimize(), pulse_heart() and pulse_doublecheck() are the other functions needed for the complete PULSE processing workflow
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:2]
pulse_data_split <- pulse_split(pulse_data_sub)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
heart_rates <- pulse_heart(pulse_data_split)
## End prepare data ----

nrow(heart_rates)
heart_rates <- pulse_choose_keep(heart_rates)
nrow(heart_rates) # halved
```

---

pulse_data *PULSE multi-channel example data*

Description

A subset of data from an experiment monitoring the heartbeat of four *Mytilus* mussels

Usage

```
pulse_data
```

Format

A list with four elements:

- `$data`, a tibble with 20,094 rows and 11 columns - one column with timestamps (named `time`) and several columns of numeric data (the voltage readings from each channel of the PULSE system; all channels with unique names)
- `$multi`, a single logical value indicating if the data was generated using a multi-channel or single-channel PULSE system
- `$vrsn`, a single numeric value representing the firmware version of the PULSE system that generated the data
- `$freq`, a single integer value representing the sampling frequency used (in Hz)

pulse_doublecheck *(STEP 5) Fix heart rate frequencies double the real value*

Description

- step 1 – pulse_read()
- step 2 – pulse_split()
- step 3 – pulse_optimize()
- step 4 – pulse_heart()
- –>> **step 5 –** pulse_doublecheck() <<–
- step 6 – pulse_choose_keep()

Flag (and correct) data points where it is likely that the heart rate frequency computed corresponds to double the actual heart rate frequency due to the algorithm having identified two peaks per heart beat

Usage

```
pulse_doublecheck(heart_rates, flag = 0.9, correct = TRUE)
```

Arguments

heart_rates	the output from pulse_heart()
flag	numerical, decimal from 0 to 1, defaults to 0.9; values of d_r above this number will be flagged as instances where the algorithm resulted in double the real heart rate. Values above 1are meaningless (zero data points will be flagged), and values below ~0.66 are too lax (many data points will be flagged when they shouldn't).
correct	logical, defaults to TRUE; if FALSE, data points with hz values likely double the real value are flagged **BUT NOT CORRECTED**. If TRUE, hz (as well as data, n, sd and ci) are corrected accordingly. Note that the correction is not reversible!

Value

A tibble similar to the one used as input, now augmented with two new columns: d_r and d_f. Values of d_r (ratio) close to 1 are indicative that the value for hz determined by the algorithm should be halved. If correct was set to TRUE, d_f flags data points where hz **HAS BEEN HALVED**. If correct was set to FALSE, then d_f flags data points where hz **SHOULD BE HALVED**.

Heart beat frequency estimation

For many invertebrates, the circulatory system includes more than one contractile chamber, meaning that there are two consecutive movements that may or may not be detected by the PULSE system's IR sensors. Furthermore, when the sensor is attached to the shell of the animal, it remains at a fixed position even as the soft body tissues move below that. As a result, even if one takes explicit care to position the sensor in such a way that only one wave peak is detected for each heart beat cycle,

at some point the animal may move and the sensor's field of view may come to encompass both contractile chambers. When that occurs, the shape of the signal detected will include two peaks per heart beat cycle, the relative sizes of which may vary considerably. To be clear, there's nothing wrong with such a signal. However, it creates a problem: the algorithm detects peaks, and therefore, if two peaks are detected for each heart beat, the resulting estimate for the heart beat frequency will show a value twice as much as the real value.

Detection method

While it is often easy to discern if a PULSE data point has two peaks per heart beat upon visual inspection, to do so automatically is much harder. The strategy employed here relies on analyzing the intervals between consecutive peaks and looking for a regular alternation between longer and shorter intervals, as well as higher and lower peak signal values. If intervals are consistently shorter, then longer, then shorter again, we can assume that the distribution of interval times is bimodal, and that there are always two peaks more closer together separated by a longer interval - a classical two-peaks-per-heart-beat situation. For example, let's say 24 peaks are detected. We can compute the time span between each peak, which will correspond to 23 intervals (measured in seconds). Then, intervals can be classified as being longer or shorter than the preceding interval. Lastly, we divide the number of longer-than-previous intervals by the total number of intervals, deriving the ratio of switching intervals. Similarly, if peak signal values are consistently higher, then lower, then higher again, we can also assume that two different heart movements belonging to the same heartbeat are represented in the data, and a similar algorithm can be followed. The closer the ratio is to 1, the more certain we are that we are facing a situation where the algorithm will result in a heart beat frequency twice the real value. Because the choice of a threshold to flag data points as needing to be halved or not is arbitrary, both the flagging and the ratio are provided in the output, thus enabling a reassessment of the resulting classification.

See Also

- `pulse_heart()` generates the tibble that is used as input.
- `pulse_read()`, `pulse_split()`, `pulse_optimize()`, `pulse_heart()` and `pulse_choose_keep()` are the other functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once, including the identification of possible heart rate doublings

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:3]
pulse_data_split <- pulse_split(pulse_data_sub)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
heart_rates <- pulse_heart(pulse_data_split)
## End prepare data ----

# Correct heartbeat frequency estimates
pulse_doublecheck(heart_rates)
```

pulse_example *Get paths to pulse example files*

Description

heartbeatr-package comes bundled with several sample files in its inst/extdata directory. This function make them easy to access

Usage

```
pulse_example(pattern = NULL)
```

Arguments

pattern	Pattern to select one or more example files. Pattern is vectorized, so more than one value can be supplied. If NULL, all example files are listed.

Value

The full path to one or more example files, or the filenames of all example files available.

See Also

- pulse_read() can be used to read data from the example files
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once, and it can be called to read and process the example files

Examples

```
# Get the paths to all example files
pulse_example()
```

pulse_find_peaks_all_channels *Determine the heartbeat rate in all channels of a PULSE split window*

Description

Take data from PULSE data window and run pulse_find_peaks_one_channel in all channels.

Usage

```
pulse_find_peaks_all_channels(split_window)
```

Arguments

split_window	one element of the pulse_data_split list() (which is the output from pulse_split()).

Value

A tibble with up to 10 rows (one for each channel) and 7 columns:

- `id`, PULSE channel IDs
- `time`, time at the center of split_window_one_channel$time
- `data`, a list of tibbles with raw PULSE data for each combination of channel and window, with columns `time`, `val` and `peak` (`TRUE` when data points correspond to wave peaks)
- `hz`, heartbeat rate estimate (in Hz)
- `n`, number of wave peaks identified
- `sd`, standard deviation of the intervals between wave peaks (normalized)
- `ci`, confidence interval (hz ± ci)

BPM

To convert to Beats Per Minute, simply multiply `hz` and `ci` by 60.

See Also

- `pulse_find_peaks_all_channels()` runs `pulse_find_peaks_one_channel()` on all PULSE channels
- `pulse_read()`, `pulse_split()`, `pulse_optimize()`, `pulse_heart()` and `pulse_choose_keep()` are the functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(pulse_data_sub)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
split_window <- pulse_data_split$data[[1]]
## End prepare data ----

# Determine heartbeat rates in all channels in one time window
pulse_find_peaks_all_channels(split_window)
```

`pulse_find_peaks_one_channel` *Determine the heart beat frequency in one PULSE channel*

Description

Take data from one PULSE channel and identify the heartbeat wave peaks using an algorithm that searches for maxima across multiple scales.

Usage

```
pulse_find_peaks_one_channel(split_window_one_channel)
```

Arguments

`split_window_one_channel` a tibble with PULSE data for only one channel with columns `$time` and `$val`

Details

function builds upon code from https://github.com/ig248/pyampd

Value

A one-row tibble with 8 columns:

- `time`, time at the center of split_window_one_channel$time
- `t_pks`, time stamps of each wave peak identified
- `hz`, heartbeat rate estimate (in Hz)
- `n`, number of wave peaks identified
- `sd`, standard deviation of the intervals between wave peaks
- `ci`, confidence interval (hz ± ci)

Standard Deviation

The `sd` computed refers to the spread of the intervals between each peak identified. It is a measure of the quality of the raw data and the ability of the algorithm to identify a real heart beat. The lower the `sd`, the more regular are the intervals between peaks, and the more likely that the algorithm did find a real signal. Conversely, higher `sds` indicate that the peaks are found at irregular intervals, and is an indication of poor quality data. In detail, `sd` is computed by: 1) taking the timestamps for each peak identified [`t_pks`], 2) computing the intervals between each pair of consecutive peaks [`as.numeric(diff(t_pks))`], and 3) computing `sd` [`sd(intervals)`].

BPM

To convert to Beats Per Minute, simply multiply `hz` and `ci` by 60.

See Also

- pulse_find_peaks_all_channels() runs pulse_find_peaks_one_channel() on all PULSE channels
- pulse_read(), pulse_split(), pulse_optimize(), pulse_heart(), pulse_doublecheck() and pulse_choose_keep() are the functions needed for the complete PULSE processing workflow
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(pulse_data_sub)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
split_window <- pulse_data_split$data[[1]]
split_window_one_channel <- split_window[,1:2]
colnames(split_window_one_channel) <- c("time", "val")
# End prepare data ----

## Determine heartbeat rates in one channel in one time window
pulse_find_peaks_one_channel(split_window_one_channel)
```

---

pulse_halve *Halves heart beat frequencies computed by* pulse_heart

---

Description

Halves the heart beat frequency computed by pulse_heart when double peaks have been detected by pulse_correct. Note that the correction cannot be reverted (if just testing, store as a different variable). The associated stats are recalculated. This function is used by pulse_correct, it is not immediately usable as standalone.

Usage

```
pulse_halve(hr)
```

Arguments

hr	a tibble as the one used as input to pulse_doublecheck(), but with the additional column d_f, which flags rows where heart beat frequencies need to be halved. All rows supplied are halved, so input should be a filtered version of the full dataset.

Value

A tibble with as many rows as the one provided as input, but with data, hz, n, sd, and ci adjusted accordingly.

See Also

- pulse_doublecheck() is the function within the heartbeatr-package that uses pulse_halve
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once, including the identification and correction of possible heart rate doublings

---

pulse_heart *(STEP 4) Determine the heartbeat rate in all channels of a split PULSE object*

Description

- step 1 – pulse_read()
- step 2 – pulse_split()
- step 3 – pulse_optimize()
- ->> **step 4 –** pulse_heart() <<-
- step 5 – pulse_doublecheck()
- step 6 – pulse_choose_keep()

For each combination of channel and time window, determine the heartbeat rate automatically.

pulse_heart() takes the output from a call to pulse_optimize() (or pulse_split() if optimization is skipped, but that is highly discouraged) and employs an algorithm optimized for the identification of wave peaks in noisy data to determine the heart beat frequency in all channels of the PULSE dataset.

Usage

```
pulse_heart(pulse_data_split, msg = TRUE, show_progress = FALSE)
```

Arguments

pulse_data_split	the output from a call to pulse_split()
msg	logical, defaults to TRUE; should non-crucial messages (but not errors) be shown (mostly for use from within the wrapper function PULSE(), where it is set to FALSE to avoid the repetition of identical messages)
show_progress	logical, defaults to FALSE. If set to TRUE, progress messages will be provided.

Value

A tibble with nrows = (number of channels) * (number of windows in pulse_data_split) and 10 columns:

- i, index of each time window's order
- smoothed, whether the data has been smoothed with pulse_smooth()
- id, PULSE channel IDs

- `time`, time at the center of each time window
- `data`, a list of tibbles with raw PULSE data for each combination of channel and window, with columns `time`, `val` and `peak` (`TRUE` in rows corresponding to wave peaks)
- `hz`, heartbeat rate estimate (in Hz)
- `n`, number of wave peaks identified
- `sd`, standard deviation of the intervals between wave peaks
- `ci`, confidence interval (hz ± ci)
- `keep`, whether `n` and `sd` are within the target thresholds

BPM

To convert to Beats Per Minute, simply multiply `hz` and `ci` by 60.

See Also

- `pulse_find_peaks_all_channels()` runs `pulse_find_peaks_one_channel()` on all PULSE channels
- `pulse_read()`, `pulse_split()`, `pulse_optimize()`, `pulse_doublecheck()` and `pulse_choose_keep()` are the other functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once
- `pulse_summarise()` can be used to reduce the number of data points returned

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:3]
pulse_data_split <- pulse_split(pulse_data_sub)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
## End prepare data ----

# Determine heartbeat rates in all channels in all time window
pulse_heart(pulse_data_split)
```

`pulse_interpolate` *Increase the number of data points in PULSE data through interpolation*

Description

The performance of the algorithm employed in the downstream function `pulse_heart()` for the detection of heart beat wave crests depends significantly on there being a sufficient number of data points around each crest. `pulse_interpolate()` reshapes the data non-destructively and improves the likelihood of `pulse_heart()` successfully estimating the inherent heartbeat rates.

- `INTERPOLATION` is highly recommended because tests on real data have shown that a frequency of at least 40 Hz is crucial to ensure wave crests can be discerned even when the underlying heartbeat rate is high (i.e., at rates above 2-3 Hz). Since the PULSE multi-channel system is not designed to capture data at such high rates (partially because it would generate files unnecessarily large), `pulse_interpolate()` is used instead to artificially increase the temporal resolution of the data by linearly interpolating to the target frequency. It is important to note that this process DOES NOT ALTER the shape of the heart beat wave, it just introduces intermediary data points. Also, the only downside to using very high values for `interpolation_freq` is the proportional increase in computing time and size of the outputs together with minimal improvements in the performance of `pulse_heart()` - but no artefacts are expected.

Usage

```
pulse_interpolate(split_window, interpolation_freq = 40, multi)
```

Arguments

`split_window`	one element of the `pulse_data_split` list() (which is the output from `pulse_split()`).
`interpolation_freq`	numeric, defautls to `40`; value expressing the frequency (in Hz) to which PULSE data should be interpolated. Can be set to `0` (zero) or any value equal or greater than `40` (the default). If set to zero, no interpolation is performed.
`multi`	logical; was the data generated by a multi-channel system (`TRUE`) or a one-channel system (`FALSE`)?

Value

The same PULSE tibble supplied in `split_window`, but now with data interpolated to `interpolation_freq` (i.e., with more data points)

See Also

- `approx()` is used for the linear interpolation of PULSE data
- `pulse_optimize()` is a wrapper function that executes `pulse_interpolate` and `pulse_smooth()` sequentially
- `pulse_read()`, `pulse_split()`, `pulse_heart()`, `pulse_doublecheck()` and `pulse_choose_keep()` are the other functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(pulse_data_sub)
## End prepare data ----

# Interpolate data to 40 Hz
pulse_interpolate(pulse_data_split$data[[1]], 40, multi = pulse_data$multi)
```

`pulse_normalize` *Normalize PULSE heartbeat rate estimates*

Description

Take the output from `PULSE()` and compute the normalized heartbeat rates. The normalization of heartbeat rates is achieved by calculating, for each individual (i.e., PULSE channel), the average heartbeat rate during a reference baseline period (ideally measured during acclimation, before the stress-inducing treatment is initiated).

Usage

```
pulse_normalize(
  heart_rates,
  FUN = mean,
  t0 = NULL,
  span_mins = 10,
  overwrite = FALSE
)
```

Arguments

`heart_rates`	the output from `PULSE()`, `pulse_heart()`, `pulse_doublecheck()` or pulse_choose_keep.
`FUN`	the function to be applied to normalize the data within the baseline period (defaults to mean; median may be more suited in some situations; any other function that returns a single numeric value is technically acceptable).
`t0`	either `NULL` (default), a lubridate::POSIXct object or a character string that can be directly converted to a lubridate::POSIXct object. Represents the beginning of the period to be used to establish the baseline heart beat frequency (same value is used for all channels). If set to `NULL`, the baseline period is set to the earliest timestamp available.
`span_mins`	numeric, defaults to 10; number of minutes since `t0`, indicating the width of the baseline period (baseline from `t0` to `t0` + `span_mins` mins)
`overwrite`	logical, defaults to `FALSE`; should the normalized values be stored in a different column (`hz_norm` if `overwrite` = `FALSE`; RECOMMENDED) or replace the data in the column `hz` (`overwrite` = `TRUE`; WARNING: the original `hz` values cannot be recovered).

Value

The same tibble provided as input, with an additional column `hz_norm` containing the normalized heart beat frequencies (`overwrite = FALSE`) or with the same number of columns and normalized data saved to the column `hz` (`overwrite = TRUE`).

Details

Normalizing heartbeat rates is important because even individuals from the same species, the same age cohort and subjected to the same treatment will have different basal heart beat frequencies. After normalizing, these differences are minimized, and the analysis can focus on the change of hear beat frequency relative to a reference period (the baseline period chosen) rather than on the absolute values of heart beat frequency - which can be misleading.

The period chosen for the baseline doesn't need to be long - it's much more important that conditions (and hopefully heart beat frequencies) are as stable and least stressful as possible during that period.

After normalization, heart beat frequencies during the baseline period will, by definition, average to `1`. Elsewhere, normalized heart beat frequencies represent ratios relative to the baseline: `2` represents a heart beat frequency double the basal frequency, while `0.5` indicates half of the basal frequency. This means that two individuals may experience a doubling of heart beat frequency throughout an experiment even if their absolute heart beat frequencies are markedly different from each other (e.g., individual 1 with hz = 0.6 at t0 and hz = 1.2 at t1, and individual 2 with hz = 0.8 at t0 and hz = 1.6 at t1, will both show hz_norm = 1 at t0 and hz_norm = 2 at t1).

Different baseline periods for each channel

`pulse_normalize` only allows setting a single baseline period. If different periods are needed for different channels or groups of channels, generate two or more subsets of `heart_rates` containing `heart_rates$id` that share the same baseline periods, normalize each independently and bind all data together at the end (see the examples section below).

See Also

- `pulse_heart()`, `pulse_doublecheck()` and `pulse_choose_keep()` are the functions that generate the input for `pulse_normalize`
- `pulse_plot()` can be called to visualize the output from `pulse_normalize`
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once, and its output can also be passed on to `pulse_normalize`

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(
   pulse_data_sub,
   window_width_secs = 30,
   window_shift_secs = 60,
   min_data_points = 0.8)
pulse_data_split <- pulse_optimize(pulse_data_split, multi = pulse_data$multi)
```

```
heart_rates <- pulse_heart(pulse_data_split)
heart_rates <- pulse_doublecheck(heart_rates)
## End prepare data ----

# Normalize data using the same period as baseline for all channels
pulse_normalize(heart_rates)

# Using a different (complex) function
pulse_normalize(heart_rates, FUN = function(x) quantile(x, 0.4))

# Apply different baseline periods to two groups of IDs
group_1 <- c("limpet_1", "limpet_2")
rbind(
  # group_1
  pulse_normalize(heart_rates[ (heart_rates$id %in% group_1), ], span_mins = 10),
  # all other IDs
  pulse_normalize(heart_rates[!(heart_rates$id %in% group_1), ], span_mins = 30)
)
```

`pulse_optimize` *(STEP 3) Optimize PULSE data through interpolation and smoothing*

Description

- step 1 – pulse_read()
- step 2 – pulse_split()
- –>> **step 3 –** pulse_optimize() <<–
- step 4 – pulse_heart()
- step 5 – pulse_doublecheck()
- step 6 – pulse_choose_keep()

IMPORTANT NOTE: `pulse_optimize()` can be skipped, but that is highly discouraged.

The performance of the algorithm employed in the downstream function `pulse_heart()` for the detection of heartbeat wave crests depends significantly on `(i)` there being a sufficient number of data points around each crest and `(ii)` the data not being too noisy. `pulse_optimize()` uses first `pulse_interpolate()` and then `pulse_smooth()` to reshape the data and improve the likelihood of `pulse_heart()` successfully estimating the inherent heartbeat rates.

- `INTERPOLATION` is highly recommended because tests on real data have shown that a frequency of at least 40 Hz is crucial to ensure wave crests can be discerned even when the underlying heartbeat rate is high (i.e., at rates above 2-3 Hz). Since the PULSE multi-channel system is not designed to capture data at such high rates (partially because it would generate files unnecessarily large), `pulse_interpolate()` is used instead to artificially increase the temporal resolution of the data by linearly interpolating to the target frequency. It is important to note that this process DOES NOT ALTER the shape of the heart beat wave, it just introduces intermediary data points. Also, the only downside to using very high values for `interpolation_freq` is the proportional increase in computing time and size of the outputs together with minimal improvements in the performance of `pulse_heart()` - but no artefacts are expected.

- SMOOTHING should be experimented with when pulse_heart() produces too many heartbeat rate estimates that are clearly incorrect. In such situations, pulse_smooth() applies a smoothing filter (normal Kernel Regression Smoother) to the data to smooth out high-frequency noise and render a more sinusoidal wave, which is easier to handle. Unlike interpolation_freq, users should exercise caution when setting bandwidth and generally opt for lower values, as there's a threshold to bandwidth values above which the resulting smoothed pulse data becomes completely unrelated to the original data, and the subsequent heartbeat rates computed with pulse_heart() may be wrong. Always double-check the data after applying a stronger smoothing. Nonetheless, note that if applied with the default bandwidth, smoothing incurs no penalty and hardly changes the data - so it isn't worth going out of the way to not apply smoothing.

Usage

```
pulse_optimize(
  pulse_data_split,
  interpolation_freq = 40,
  bandwidth = 0.2,
  raw_v_smoothed = FALSE,
  multi
)
```

Arguments

pulse_data_split	the output from a call to pulse_split()
interpolation_freq	numeric, defautls to 40; value expressing the frequency (in Hz) to which PULSE data should be interpolated. Can be set to 0 (zero) or any value equal or greater than 40 (the default). If set to zero, no interpolation is performed.
bandwidth	numeric, defaults to 0.2; the bandwidth for the Kernel Regression Smoother. If equal to 0 (zero) no smoothing is applied. Normally kept low (0.1 - 0.3) so that only very high frequency noise is removed, but can be pushed up all the way to 1 or above (especially when the heartbeat rate is expected to be slow, as is typical of oysters, but double check the resulting data). Type ?ksmooth for additional info.
raw_v_smoothed	logical, defaults to FALSE; if set to FALSE, the output includes only one list obtained after applying interpolation and smoothing according to the values set. If set to TRUE, a list with two lists is returned, one after applying only interpolation (i.e., "raw"), and the other after applying both interpolation and smoothing (i.e, "smoothed").
multi	logical; was the data generated by a multi-channel system (TRUE) or a one-channel system (FALSE)?

Value

The same structure as the input data, which is a tibble with three columns, but now with the values on column $smoothed switched to TRUE if smoothing was applied and the contents of column

$data modified in accordance with the parameters called. If raw_v_smoothed is FALSE, the tibble returned will have the same number of rows as the input tibble. If raw_v_smoothed is TRUE, the tibble returned will have twice the number of rows as the input tibble, with half not smoothed (i.e., only interpolation applied), the other half smoothed (i.e., interpolation and smoothing applied) and the order indexes in $i duplicated. Downstream functions will process both types of output automatically.

Raw v smoothed

When raw_v_smoothed is set to TRUE, two heart rate estimates are produced for each data point - one based on the raw data and another after applying smoothing. The cost is an increase in processing time. The benefit is an improvement in the ability to estimate heart rates when the heart is beating faster, as in those cases smoothing the data may become counterproductive. When raw_v_smoothed = TRUE, pulse_choose_keep() will decide which of the estimates to retain for each data point (based on user-defined parameters).

See Also

- approx() is used by pulse_interpolate() for the linear interpolation of PULSE data
- ksmooth() is used by pulse_smooth() for the kernel smoothing of PULSE data
- pulse_optimize() is a wrapper function that executes pulse_interpolate() and pulse_smooth() sequentially
- pulse_read(), pulse_split(), pulse_heart(), pulse_doublecheck() and pulse_choose_keep() are the other functions needed for the complete PULSE processing workflow
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(pulse_data_sub)
## End prepare data ----

# Optimize data by interpolating to 40 Hz and applying a slight smoothing
pulse_optimize(pulse_data_split, 40, 0.2, multi = pulse_data$multi)
```

---

pulse_plot *Plot processed PULSE data*

Description

A shortcut function based on ggplot2 to facilitate the quick inspection of the results of the analysis (with the option to separate channels or not).

Usage

```
pulse_plot(
  heart_rates,
  ID = NULL,
  normalized = FALSE,
  smooth = TRUE,
  points = TRUE,
  facets = TRUE,
  bpm = FALSE
)
```

Arguments

heart_rates	the output from PULSE() (or pulse_heart() and any of the downstream functions).
ID	character string naming a single target channel id (must match exactly); defaults to NULL, which results in all IDs being plotted
normalized	logical, defaults to FALSE; whether to plot hz_norm (TRUE) or hz (FALSE).
smooth	logical, defaults to TRUE; whether to plot a loess smoothing curve (TRUE) or a line (FALSE).
points	logical, defaults to TRUE; whether to overlay data points.
facets	logical, defaults to TRUE; whether to separate channels in facets (TRUE) or to plot all data together (FALSE)
bpm	logical, defaults to FALSE; whether to plot heartbeat frequency using Hz (FALSE) or Beats Per Minute (TRUE); forced to FALSE if normalized is set to TRUE

Details

This function is **NOT meant** for high-level displaying of PULSE data - it's simply a quick shortcut to facilitate data inspection.

When inspecting the plot with smooth = TRUE, assess if the loess confidence intervals are too wide for any given channel, which is indicative of data with high variability (not ideal).

If using smooth = FALSE, then look for the width of the confidence interval for each data point.

Value

A ggplot object that can be augmented using ggplot2 syntax or plotted right away

See Also

- use pulse_plot_raw() to quickly plot the raw PULSE data for a given channel/split window
- PULSE() (or pulse_heart() and any of the downstream functions) generates the input for pulse_plot
- call pulse_normalize() to anchor heart beat data from each channel to a reference period during the experiment

Examples

```
## Begin prepare data ----
paths <- pulse_example()
chn <- paste0("c", formatC(1:10, width = 2, flag = "0"))
heart_rates <- PULSE(
  paths,
  discard_channels  = chn[-9],
  raw_v_smoothed = FALSE,
  show_progress = FALSE)
## End prepare data ----

# Default
pulse_plot(heart_rates)

# A single ID
pulse_plot(heart_rates, ID = "c09")

# Without facets, the different basal heartbeat rates become evident in #' non-normalized data
pulse_plot(heart_rates, facets = FALSE)

# Without facets, normalized data always lines up (around 1) during the #' baseline period
pulse_plot(
  pulse_normalize(heart_rates),
  normalized = TRUE,
  facets = FALSE)

# The plot can be modified using ggplot2 syntax
pulse_plot(heart_rates) + ggplot2::theme_dark()

# Data can also be visualized using BPM (Beats Per Minute)
pulse_plot(heart_rates, bpm = TRUE)

# If inspecting the heart rate estimates for a single ID and suppressing the
# LOESS smoothing, the confidence interval of each estimate is also shown
pulse_plot(heart_rates, ID = "c09", smooth = FALSE)
```

`pulse_plot_one` *heartbeatr utility function*

Description

Basic function to quickly plot data from one data point only

Usage

```
pulse_plot_one(data)
```

Arguments

`data`	PULSE tibble (tow columns: $time and $val)

Value

A ggplot object that can be augmented using ggplot2 syntax or plotted right away

Examples

```
## Begin prepare data ----
path <- pulse_example()
chn <- paste0("c", formatC(1:10, width = 2, flag = "0"))
heart_rates <- PULSE(
  path,
  discard_channels  = chn[-8],
  raw_v_smoothed = FALSE,
  show_progress = FALSE)
## End prepare data ----

pulse_plot_one(heart_rates$data[[1]])
```

---

pulse_plot_raw *Plot raw PULSE data*

Description

A shortcut function based on ggplot2 to facilitate the quick inspection of the raw data underlying the analysis (with the peaks detected signaled with red dots). Useful to visually inspect the performance of the algorithm.

Usage

```
pulse_plot_raw(heart_rates, ID, target_time = NULL, range = 0, target_i = NULL)
```

Arguments

heart_rates	the output from PULSE() (or pulse_heart() and any of the downstream functions).
ID	character string naming a single target channel id (must match exactly); defaults to NULL, which results in all IDs being plotted
target_time	a target time expressed as POSIX time or a string that can be converted directly by as.POSIXct(); target_i will be computed as the window closest to the target_time
range	numeric, defaults to 0 (only target_i will be plotted); value indicating how many more windows to plot (centered around the target window, i.e., if target_i = 5 and range = 2, windows 3 to 7 will be plotted, with window 5 at the center)
target_i	numeric; value pointing to the index of the target window (which can be found in the column i of heart_rates)

Details

This function is **NOT meant** for high-level displaying of the data - it's simply a quick shortcut to facilitate data inspection.

When inspecting the plot, assess if red dots top all heartbeats and no more than that. Difficult datasets may result in true heartbeats being missed (false negatives) or non-heartbeats (noise) being erroneously detected (false positives). Note that the wider the time window (controlled by the `window_width_secs` parameter in `pulse_split()`) and the higher the heartbeat rate, the less critical are a few false positives or negatives (over a 10 secs window, missing 1 peak in 10 results in hz to drop by 10% (from 1 to 0.9), while over a 30 secs window, missing 1 peak in 30 results in a drop of 3.33% (from 1 to 0.967), and missing 1 peak in 60 results in a drop of just 1.7%.

Value

A ggplot object that can be augmented using ggplot2 syntax or plotted right away

See Also

- use `pulse_plot()` to plot processed PULSE data for a several channels
- `PULSE()` (or `pulse_heart()` and any of the downstream functions) generates the input for `pulse_plot_raw`
- call `pulse_normalize()` to anchor heart beat data from each channel to a reference period during the experiment

Examples

```
## Begin prepare data ----
paths <- pulse_example()
chn <- paste0("c", formatC(1:10, width = 2, flag = "0"))
heart_rates <- PULSE(
  paths,
  discard_channels  = chn[-8],
  raw_v_smoothed = FALSE,
  show_progress = FALSE)
## End prepare data ----

# Single window (in both cases, the 5th window)
# using a target date and time
pulse_plot_raw(heart_rates, "c08", "2024-10-01 10:56")
# using the index
pulse_plot_raw(heart_rates, "c08", target_i = 5)

# Multiple windows (less detail, but more context)
pulse_plot_raw(heart_rates, "c08", "2024-10-01 10:56", 2)

# The plot can be modified using ggplot2 syntax
pulse_plot_raw(heart_rates, "c08", target_i = 5) + ggplot2::theme_classic()
```

pulse_read *(STEP 1) Read data from all PULSE files in the target folder*

Description

- –>> **step 1 –** pulse_read() <<–
- step 2 – pulse_split()
- step 3 – pulse_optimize()
- step 4 – pulse_heart()
- step 5 – pulse_doublecheck()
- step 6 – pulse_choose_keep()

Importing data from PULSE '.csv' files is the first step of the analysis of PULSE data.

pulse_read() checks that the paths provided by the user conform to certain expectations and then reads the data from all files and merges into a single tibble. Only data from the same experiment should be read at the same time (i.e., with the same channel names, collected with the same sampling frequency, and produced using a PULSE multi-channel or a PULSE one-channel system running the same firmware version throughout the experiment). To put it differently, one call to pulse_read() can only read files where the header is absolutely invariant, and only the data portion of the files differs. The output of pulse_read() can be directly passed on to pulse_split().

Usage

```
pulse_read(paths, msg = TRUE)
```

Arguments

paths	character vectors, containing file paths to CSV files produced by a PULSE system during a single experiment.
msg	logical, defaults to TRUE; should non-crucial messages (but not errors) be shown (mostly for use from within the wrapper function PULSE(), where it is set to FALSE to avoid the repetition of identical messages)

Value

A list with four elements:

- $data , tibble containing all data from all PULSE files
- $multi, logical indicating if the data is from a multi-channel system (TRUE) or from a one-channel system (FALSE)
- $vrsn , numeric value representing the version number of the PULSE system where the data was generated
- $freq , numeric value representing the sampling frequency used (in Hz)

Time zones

PULSE systems **ALWAYS** record data using **UTC +0**. This is intentional! If data were to be recorded using local time zones, issues with daylight saving time, leap seconds, etc. could spoil the dataset. Worse, should the information about which time zone had been used get lost or accidentally modified, the validity of the entire dataset could be compromised. By always using UTC +0 all these issues are minimized and the processing pipeline becomes vastly easier and more efficient. Still, this means that after the data has been processed using the `heartbeatr-package`, the user may need to adjust the time zone of all timestamps so that the timing matches other information relative to the experiment.

See Also

- `pulse_split()`, `pulse_optimize()`, `pulse_heart()`, `pulse_doublecheck()` and `pulse_choose_keep()` are the other functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
paths <- pulse_example()
## End prepare data ----

pulse_read(paths)
```

---

pulse_smooth *Smooth PULSE data*

Description

The performance of the algorithm employed in the downstream function `pulse_heart()` for the detection of heart beat wave crests depends significantly on the data not being too noisy. `pulse_smooth()` reshapes the data and improves the likelihood of `pulse_heart()` successfully estimating the inherent heartbeat rates.

- `SMOOTHING` should be experimented with when `pulse_heart()` produces too many heartbeat rate estimates that are clearly incorrect. In such situations, `pulse_smooth()` applies a smoothing filter (normal Kernel Regression Smoother) to the data to smooth out high-frequency noise and render a more sinusoidal wave, which is easier to handle. Unlike `interpolation_freq`, users should exercise caution when setting `bandwidth` and generally opt for lower values, as there's a threshold to bandwidth values above which the resulting smoothed pulse data becomes completely unrelated to the original data, and the subsequent heartbeat rates computed with `pulse_heart()` may be wrong. Always double-check the data after applying a stronger smoothing. Nonetheless, note that if applied with the default `bandwidth`, smoothing incurs no penalty and hardly changes the data - so it isn't worth going out of the way to not apply smoothing.

Usage

```
pulse_smooth(split_window, bandwidth = 0.2)
```

Arguments

split_window	one element of the pulse_data_split list() (which is the output from pulse_split()).
bandwidth	numeric, defaults to 0.2; the bandwidth for the Kernel Regression Smoother. If equal to 0 (zero) no smoothing is applied. Normally kept low (0.1 - 0.3) so that only very high frequency noise is removed, but can be pushed up all the way to 1 or above (especially when the heartbeat rate is expected to be slow, as is typical of oysters, but double check the resulting data). Type ?ksmooth for additional info.

Value

The same PULSE tibble supplied in split_window, but now with data for all channels transformed by smoothing.

See Also

- ksmooth() is used for the kernel smoothing of PULSE data
- pulse_optimize() is a wrapper function that executes pulse_interpolate() and pulse_smooth sequentially
- pulse_read(), pulse_split(), pulse_heart(), pulse_doublecheck() and pulse_choose_keep() are the other functions needed for the complete PULSE processing workflow
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
pulse_data_split <- pulse_split(pulse_data_sub)
## End prepare data ----

# Smooth data slightly ('bandwidth' = 0.2)
pulse_smooth(pulse_data_split$data[[1]], 0.2)
```

---

pulse_split *(STEP 2) Split* pulse_data *across sequential time windows*

Description

- step 1 – pulse_read()
- ->> **step 2 –** pulse_split() <<-
- step 3 – pulse_optimize()
- step 4 – pulse_heart()
- step 5 – pulse_doublecheck()
- step 6 – pulse_choose_keep()

After all raw PULSE data has been imported, the dataset must be split across sequential time windows.

pulse_split() takes the output from a call to pulse_read() and splits data across user-defined time windows. The output of pulse_split() can be immediately passed to pulse_heart(), or first optimized with pulse_optimize() and only then passed to pulse_heart() (highly recommended).

Usage

```
pulse_split(
  pulse_data,
  window_width_secs = 30,
  window_shift_secs = 60,
  min_data_points = 0.8,
  subset = 0,
  subset_seed = NULL,
  subset_reindex = FALSE,
  msg = TRUE
)
```

Arguments

pulse_data	the output from a call to pulse_read().
window_width_secs	numeric, in seconds, defaults to 30; the width of the time windows over which heart rate frequency will be computed.
window_shift_secs	numeric, in seconds, defaults to 60; by how much each subsequent window is shifted from the preceding one.
min_data_points	numeric, defaults to 0.8; decimal from 0 to 1, used as a threshold to discard incomplete windows where data is missing (e.g., if the sampling frequency is 20 and window_width_secs = 30, each window should include 600 data points, and so if min_data_points = 0.8, windows with less than 600 * 0.8 = 480 data points will be rejected).
subset	numerical, defaults to 0; the number of time windows to keep from the entire dataset (or the number of entries to reject if set to a negative value); smaller subsets make the entire processing quicker and facilitate the execution of trial runs to optimize parameter selection before processing the entire dataset.

subset_seed	numerical, defaults to NULL; only used if subset is different from 0; subset_seed controls the seed used when extracting a subset of the available data; if set to NULL, a random seed is selected, resulting in rows being selected randomly; alternatively, the user can set a specific seed in order to always select the same rows (important when the goal is to compare the impact of different parameter combinations using the exact same data points).
subset_reindex	logical, defaults to FALSE; only used if subset is different from 0; after extracting a subset of the available data, should rows be re-indexed (i.e., .$i made fully sequential); re-indexed rows make using pulse_plot_raw() easier, but row identity doesn't match anymore with row identity before subsetting.
msg	logical, defaults to TRUE; should non-crucial messages (but not errors) be shown (mostly for use from within the wrapper function PULSE(), where it is set to FALSE to avoid the repetition of identical messages)

Value

A tibble with three columns. Column $i stores the order of each time window. Column $smoothed is a logical vector flagging smoothed data (at this point defaulting to FALSE, but later if pulse_optimize is used, values can change to TRUE. Column $data is a list with all the valid time windows (i.e., complying with min_data_points), each window being a subset of pulse_data (a tibble with at least 2 columns (time + one or more channels) containing PULSE data with timestamps within that time window)

Window width and shift

A good starting point for window_width_secs is to set it to between 30 and 60 seconds.

As a rule of thumb, use lower values for data collected from animals with naturally faster heart rates and/or that have been subjected to treatments conducive to fast heart rates still (e.g., thermal performance ramps). In such cases, lower values will result in higher temporal resolution, which may be crucial if experimental conditions are changing rapidly. Conversely, experiments using animals with naturally slower heart rates and/or subjected to treatments that may cause heart rates to stabilize or even slow (e.g., control or cold treatments) may require the use of higher values for window_width_secs, as the resulting windows should encompass no less than 5-7 heartbeat cycles.

As for window_shift_secs, set it to a value:

- smaller than window_width_secs if overlap between windows is desired (not usually recommended) (if window_width_secs = 30 and window_shift_secs = 15, the first 3 windows will go from [0, 30), [15, 45) and [30, 60))
- equal to window_width_secs to process all data available (if window_width_secs = 30 and window_shift_secs = 30, the first 3 windows will go from [0, 30), [30, 60) and [60, 90))
- larger than window_width_secs to skip data (ideal for speeding up the processing of large datasets) (if window_width_secs = 30 and window_shift_secs = 60, the first 3 windows will go from [0, 30), [60, 90) and [120, 150))

In addition, also consider that lower values for the window_... parameters may lead to oversampling and a cascade of statistical issues, the resolution of which may end up negating any advantage gained.

Handling gaps in the dataset

`min_data_points` shouldn't be set too low, otherwise only nearly empty windows will be rejected.

See Also

- `pulse_read()`, `pulse_optimize()`, `pulse_heart()`, `pulse_doublecheck()` and `pulse_choose_keep()` are the other functions needed for the complete PULSE processing workflow
- `PULSE()` is a wrapper function that executes all the steps needed to process PULSE data at once

Examples

```
## Begin prepare data ----
pulse_data_sub <- pulse_data
pulse_data_sub$data <- pulse_data_sub$data[,1:5]
## End prepare data ----

pulse_split(pulse_data_sub)
```

---

pulse_summarise *Summarise PULSE heartbeat rate estimates over new time windows*

Description

Take the output from `PULSE()` and summarise `hz` estimates over new user-defined time windows using `FUN` (a summary function). In effect, this procedure reduces the number of data points available over time.

Note that the output of `pulse_summarise()` can be inspected with `pulse_plot()` but not `pulse_plot_raw()`.

Usage

```
pulse_summarise(
  heart_rates,
  FUN = stats::median,
  span_mins = 10,
  min_data_points = 2
)
```

Arguments

`heart_rates`	the output from `PULSE()`, `pulse_heart()`, `pulse_doublecheck()` or pulse_choose_keep.
`FUN`	a custom function, defaults to `median`; Note that `FUN` must take a vector of `numeric` values and output a single `numeric` value.
`span_mins`	integer, in `mins`, defaults to `10`; expresses the width of the new summary windows

`min_data_points`
: numeric, defaults to `2`; value indicating the minimum number of data points in each new summarizing window. Windows covering less data points are discarded. If set to `0` (zero), no window is ever discarded.

Value

A similar tibble as the one provided for input, but fewer columns and rows. Among the columns now absent is the `data` column (raw data is no longer available). IMPORTANT NOTE: Despite retaining the same names, several columns present in the output now provide slightly different information (because they are recalculated for each summarizing window): `time` corresponds to the first time stamp of the summarizing window; `n` shows the number of valid original windows used by the summary function; `sd` represents the standard deviation of all heartbeat rate estimates within each summarizing window (and not the standard deviation of the intervals between each identified wave peak, as was the case in `heart_rates`); `ci` is the confidence interval of the new value for `hz`.

Details

The PULSE multi-channel system captures data continuously. When processing those data, users should aim to obtain estimates of heart beat frequency at a rate that conforms to their system's natural temporal variability, or risk running into oversampling (which has important statistical implications and must be avoided or explicitly handled).

With this in mind, users can follow two strategies:

If, for example, users are targeting 1 data point every 5 mins...

- If the raw data is of good quality (i.e., minimal noise, signal wave with large amplitude), users can opt for a relatively narrow split_window (e.g, by setting `window_width_secs` in `PULSE()` (or `pulse_split()`) to `30` secs) and to only sample split_windows every 5 mins with `window_shift_secs = 300`. This means that data is processed in 5-mins split-windows where 30 secs of data are used and four and a half mins of data are skipped, yielding our target of 1 data point every 5 mins. Doing so will greatly speed up the processing of the data (less and smaller windows to work on), and the final output will immediately have the desired sample frequency. However, if any of the split_windows effectively analysed features a gap in the data or happens to coincide with the occasional drop in signal quality, those estimates of heartbeat rate will reflect that lack of quality (even if *better* data may be present in the four and a half mins of data that is being skipped). This strategy is usually used at the beginning to assess the dataset, and depending on the results, the more time-consuming strategy described next may have to be used instead.
- If sufficient computing power is available and/or the raw data can't be guaranteed to be high quality from beginning to end, users can opt for a strategy that scans the entire dataset without skipping any data. This can be achieved by setting `window_width_secs` and `window_shift_secs` in `PULSE()` (or `pulse_split()`) to the same low value. In this case, if both parameters are set to `30` secs, processing will take significantly longer and each 5 mins of data will result in `10` data points. Then, `pulse_summarise` can be used with `span_mins = 5` to summarise the data points back to the target sample frequency. More importantly, if the right summary function is used, this strategy can greatly reduce the negative impact of spurious *bad* readings. For example, setting `FUN = median`, will reduce the contribution of values of `hz` that deviate from the center ("wrong" values) to the final heartbeat estimate for a given time window). Thus, if the computational penalty is bearable, this more robust strategy can prove useful.

See Also

- pulse_heart(), pulse_doublecheck(), pulse_choose_keep(), and pulse_normalize() are the functions that generate the input for pulse_summarise
- pulse_plot() can be called to visualize the output from pulse_summarise (but not pulse_plot_raw())
- PULSE() is a wrapper function that executes all the steps needed to process PULSE data at once, and its output can also be passed on to pulse_summarise

Examples

```
## Begin prepare data ----
paths <- pulse_example()
heart_rates <- PULSE(
  paths,
  discard_channels = c(paste0("c0", c(1:7, 9)), "c10"),
  show_progress = FALSE
  )
## End prepare data ----

# Summarise heartbeat estimates (1 data point every 5 mins)
nrow(heart_rates) # == 13
summarised_5mins <- pulse_summarise(heart_rates, span_mins = 5)
nrow(summarised_5mins) # == 3
summarised_5mins

# using a custom function
pulse_summarise(heart_rates, span_mins = 5, FUN = function(x) quantile(x, 0.2))

# normalized data is supported automatically
pulse_summarise(pulse_normalize(heart_rates))

# Note that visualizing the output from 'plot_summarise()' with
#  'pulse_plot()' may result in many warnings
pulse_plot(summarised_5mins)
"> There were 44 warnings (use warnings() to see them)"

# That happens when the value chosen for 'span_mins' is such
#  that the output from 'plot_summarise()' doesn't contain
#  enough data points for the smoothing curve to be computed
# Alternatively, do one of the following:

# reduce 'span_mins' to still get enough data points
pulse_plot(pulse_summarise(heart_rates, span_mins = 2, min_data_points = 0))

# or disable the smoothing curve
pulse_plot(summarised_5mins, smooth = FALSE)
```

Index

∗ **datasets**
    pulse_data, 13

approx(), *8*, *22*, *27*

find_peaks, 2

is.pulse, 3

ksmooth(), *8*, *27*, *34*

lubridate::POSIXct, *23*

mean, *23*
median, *23*

PULSE, 3
PULSE(), *7*, *11, 12*, *15–17*, *19–24*, *27, 28*, *30, 31*, *33, 34*, *37–39*
PULSE_by_chunks, 9
PULSE_by_chunks(), *4*, *6*, *8*
pulse_choose_keep, 12, *23*, *37*
pulse_choose_keep(), *3*, *7, 8*, *12*, *14, 15*, *17*, *19–22*, *24, 25*, *27*, *32–35*, *37*, *39*
pulse_data, 13
pulse_doublecheck, 14
pulse_doublecheck(), *3*, *5*, *7, 8*, *10*, *12*, *14*, *19–25*, *27*, *32–35*, *37*, *39*
pulse_example, 16
pulse_find_peaks_all_channels, 16
pulse_find_peaks_all_channels(), *17*, *19*, *21*
pulse_find_peaks_one_channel, 18
pulse_find_peaks_one_channel(), *17*, *19*, *21*
pulse_halve, 19
pulse_heart, 20
pulse_heart(), *3*, *7, 8*, *12*, *14, 15*, *17*, *19, 20*, *22–25*, *27, 28*, *30–35*, *37*, *39*
pulse_interpolate, 21
pulse_interpolate(), *8*, *27*, *34*
pulse_normalize, 23
pulse_normalize(), *3*, *7, 8*, *28*, *31*, *39*
pulse_optimize, 25, *36*
pulse_optimize(), *3*, *7, 8*, *12*, *14, 15*, *17*, *19–22*, *25*, *27*, *32–35*, *37*
pulse_plot, 27
pulse_plot(), *3*, *8*, *24*, *31*, *37*, *39*
pulse_plot_one, 29
pulse_plot_raw, 30
pulse_plot_raw(), *3*, *8*, *28*, *37*, *39*
pulse_read, 32
pulse_read(), *3*, *7, 8*, *12*, *14–17*, *19–22*, *25*, *27*, *32*, *34, 35*, *37*
pulse_smooth, 33
pulse_smooth(), *8*, *20*, *22*, *27*
pulse_split, 34
pulse_split(), *3*, *7, 8*, *12*, *14–17*, *19–22*, *25–27*, *31–35*, *38*
pulse_summarise, 37
pulse_summarise(), *3*, *7, 8*, *21*