Package 'LightLogR'

June 10, 2025

Title Process Data from Wearable Light Loggers and Optical Radiation Dosimeters

Version 0.9.2

Description Import, processing, validation, and visualization of personal light exposure measurement data from wearable devices. The package implements features such as the import of data and metadata files, conversion of common file formats, validation of light logging data, verification of crucial metadata, calculation of common parameters, and semi-automated analysis and visualization.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

URL https://github.com/tscnlab/LightLogR,
https://tscnlab.github.io/LightLogR/,
https://zenodo.org/doi/10.5281/zenodo.11562600

BugReports https://github.com/tscnlab/LightLogR/issues

Imports cowplot, dplyr, ggplot2, ggsci, ggtext, hms, janitor, lubridate, magrittr, plotly, purrr, readr, rlang, scales, slider, stats, stringr, suntools, tibble, tidyr, utils

Depends R (>= 4.3)

LazyData true

Suggests covr, flextable, gghighlight, gt, gtsummary, knitr, patchwork, pkgload, rmarkdown, rsconnect, testthat (>= 3.0.0), tidyverse

Config/testthat/edition 3

Config/Needs/website rmarkdown

NeedsCompilation no

Author Johannes Zauner [aut, cre] (ORCID: <https://orcid.org/0000-0003-2171-4566>),
Manuel Spitschan [aut] (ORCID: <https://orcid.org/0000-0002-8572-9268>),
Steffen Hartmeyer [aut] (ORCID:

<https://orcid.org/0000-0002-2813-2668>),
MeLiDos [fnd],
EURAMET [fnd] (European Association of National Metrology Institutes. Website: www.euramet.org. Grant Number: 22NRM05 MeLiDos. Grant Statement: The project (22NRM05 MeLiDos) has received funding from the European Partnership on Metrology, co-financed from the European Union's Horizon Europe Research and Innovation Programme and by the Participating States.),
European Union [fnd] (Co-funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EURAMET. Neither the European Union nor the granting authority can be held responsible for them.),
TSCN-Lab [cph] (URL: www.tscnlab.org)

Maintainer Johannes Zauner <johannes.zauner@tum.de>

Repository CRAN

Date/Publication 2025-06-10 11:10:02 UTC

Contents

`add_Date_col` *Create a Date column in the dataset*

Description

Create a Date column in the dataset

Usage

```
add_Date_col(
  dataset,
  Date.colname = Date,
  group.by = FALSE,
  as.wday = FALSE,
  Datetime.colname = Datetime
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Date.colname` Name of the newly created column. Expects a `symbol`. The default(`Date`) works well with other functions in LightLogR. Will overwrite existing columns of identical name.

`group.by` Logical whether the output should be (additionally) grouped by the new column

`as.wday` Logical of whether the added column should calculate day of the week instead of date. If `TRUE` will create a factor with weekday abbreviations, where the week starts with `Mon`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

Value

a `data.frame` object identical to `dataset` but with the added column of Date data

Examples

```
sample.data.environment %>% add_Date_col()
#days of the week
sample.data.environment %>%
  add_Date_col(as.wday = TRUE, group.by = TRUE) |>
  summarize_numeric(remove = c("Datetime"))
```

`add_states` *Add states to a dataset based on groups and start/end times*

Description

`add_states()` brings states to a time series dataset. It uses the `States.dataset` to add states to the `dataset`. The `States.dataset` must at least contain the same variables as the `dataset` grouping, as well as a start and end time. Beware if both datasets operate on different time zones and consider to set `force.tz` = `TRUE`.

Usage

```
add_states(
  dataset,
  States.dataset,
  Datetime.colname = Datetime,
  start.colname = start,
  end.colname = end,
  force.tz = FALSE,
  leave.out = c("duration", "epoch")
)
```

Arguments

`dataset` A light logger dataset. Needs to be a dataframe.

`States.dataset` A light logger dataset. Needs to be a dataframe. This dataset must contain the same variables as the `dataset` grouping, as well as a start and end time. Any other column, that is not in `leave.out` will be added to the dataset.

`Datetime.colname` The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

`start.colname, end.colname` The columns that contain the start and end time. Need to be `POSIXct` and part of the `States.dataset`.

`force.tz`	If `TRUE`, the start and end times of the `States.dataset` will be forced to the same time zone as the `dataset` using `lubridate::force_tz()`. If `FALSE` (default), the start and end times of the `States.dataset` will be used as is.
`leave.out`	A character vector of columns that should not be carried over to the `dataset`

Details

Beware if columns in the `dataset` and `States.dataset` have the same name (other then grouping variables). The underlying function, `dplyr::left_join()` will mark the columns in the `dataset` with a suffix `.x`, and in the `States.dataset` with a suffix `.y`.

Value

a modified `dataset` with the states added. The states are added as new columns to the `dataset`. The columns are named after the columns in the `States.dataset`, except for the start and end times, which are removed.

Examples

```
states <-
sample.data.environment |>
  filter_Date(length = "1 day") |>
  extract_states(Daylight, MEDI > 1000)

states |> head(2)

#add states to a dataset and plot them - as we only looked for states on the
# first day (see above), only the first day will show up in the plot
sample.data.environment |>
 filter_Date(length = "2 day") |>
 add_states(states) |>
 gg_days() |>
 gg_state(Daylight)
```

`add_Time_col` *Create a Time-of-Day column in the dataset*

Description

Create a Time-of-Day column in the dataset

Usage

```
add_Time_col(
  dataset,
  Datetime.colname = Datetime,
  Time.colname = Time,
  output.dataset = TRUE
)
```

Arguments

dataset	A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.
Datetime.colname	column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.
Time.colname	Name of the newly created column. Expects a `symbol`. The default(`Time`) works well with other functions in LightLogR. Will overwrite existing columns of identical name.
output.dataset	should the output be a `data.frame` (Default `TRUE`) or a vector with `hms` (`FALSE`) times? Expects a `logical` scalar.

Value

a `data.frame` object identical to `dataset` but with the added column of Time-of-Day data, or a `vector` with the Time-of-Day-data

Examples

```
sample.data.environment %>% add_Time_col()
```

aggregate_Date *Aggregate dates to a single day*

Description

Condenses a `dataset` by aggregating the data to a single day per group, with a resolution of choice `unit`. `aggregate_Date()` is opinionated in the sense that it sets default handlers for each data type of `numeric`, `character`, `logical`, and `factor`. These can be overwritten by the user. Columns that do not fall into one of these categories need to be handled individually by the user (`...` argument) or will be removed during aggregation. If no unit is specified the data will simply be aggregated to the most common interval (`dominant.epoch`) in every group. `aggregate_Date()` is especially useful for summary plots that show an average day.

Usage

```
aggregate_Date(
  dataset,
  Datetime.colname = Datetime,
  unit = "none",
  type = c("round", "floor", "ceiling"),
  date.handler = stats::median,
  numeric.handler = mean,
  character.handler = function(x) names(which.max(table(x, useNA = "ifany"))),
```

```
  logical.handler = function(x) mean(x) >= 0.5,
  factor.handler = function(x) factor(names(which.max(table(x, useNA = "ifany")))),
  datetime.handler = stats::median,
  duration.handler = function(x) lubridate::duration(mean(x)),
  time.handler = function(x) hms::as_hms(mean(x)),
  ...
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`unit` Unit of binning. See `lubridate::round_date()` for examples. The default is `"none"`, which will not aggregate the data at all, but is only recommended for regular data, as the condensation across different days will be performed by time. Another option is `"dominant.epoch"`, which means everything will be aggregated to the most common interval. This is especially useful for slightly irregular data, but can be computationally expensive.

`type` One of `"round"`(the default), `"ceiling"` or `"floor"`. Setting chooses the relevant function from **lubridate**.

`date.handler` A function that calculates the aggregated day for each group. By default, this is set to `median`.

`numeric.handler`, `character.handler`, `logical.handler`, `factor.handler`, `datetime.handler`, `duration.handler`, `time.handler` functions that handle the respective data types. The default handlers calculate the `mean` or `median` for `numeric`, `POSIXct`, `duration`, and `hms`, and the `mode` for `character`, `factor` and `logical` types.

`...` arguments given over to `dplyr::summarize()` to handle columns that do not fall into one of the categories above.

Details

Summary values for type `POSIXct` are calculated as the median, because the mean can be nonsensical at times (e.g., the mean of Day1 18:00 and Day2 18:00, is Day2 6:00, which can be the desired result, but if the focus is on time, rather then on datetime, it is recommended that values are converted to times via `hms::as_hms()` before applying the function (the mean of 18:00 and 18:00 is still 18:00, not 6:00). Using the median as a default handler ensures a more sensible datetime.

`aggregate_Date()` splits the `Datetime` column into a `Date.data` and a `Time` column. It will create subgroups for each `Time` present in a group and aggregate each group into a single day, then remove the sub grouping.

Use the `...` to create summary statistics for each group, e.g. maximum or minimum values for each time point group.

Performing aggregate_Datetime() with any unit and then aggregate_Date() with a unit of "none" is equivalent to just using aggregate_Date() with that unit directly (provided the other arguments are set the same between the functions). Disentangling the two functions can be useful to split the computational cost for very small instances of unit in large datasets. It can also be useful to apply different handlers when aggregating data to the desired unit of time, before further aggregation to a single day, as these handlers as well as ... are used twice if the unit is not set to "none".

Value

A tibble with aggregated Datetime data, at maximum one day per group. If the handler arguments capture all column types, the number of columns will be the same as in the input dataset.

Examples

```
library(ggplot2)
#gg_days without aggregation
sample.data.environment %>%
 gg_days()

#with daily aggregation
sample.data.environment %>%
 aggregate_Date() %>%
 gg_days()

#with daily aggregation and a different time aggregation
sample.data.environment %>%
 aggregate_Date(unit = "15 mins", type = "floor") %>%
 gg_days()

#adding further summary statistics about the range of MEDI
 sample.data.environment %>%
 aggregate_Date(unit = "15 mins", type = "floor",
               MEDI_max = max(MEDI),
               MEDI_min = min(MEDI)) %>%
 gg_days() +
 geom_ribbon(aes(ymin = MEDI_min, ymax = MEDI_max), alpha = 0.5)
```

---

aggregate_Datetime *Aggregate Datetime data*

Description

Condenses a dataset by aggregating the data to a given (shorter) interval unit. aggregate_Datetime() is opinionated in the sense that it sets default handlers for each data type of numeric, character, logical, factor, duration, time, and datetime. These can be overwritten by the user. Columns that do not fall into one of these categories need to be handled individually by the user (... argument) or will be removed during aggregation. If no unit is specified the data will simply be aggregated to the most common interval (dominant.epoch), which is most often not an aggregation but a rounding.)

Usage

```
aggregate_Datetime(
  dataset,
  unit = "dominant.epoch",
  Datetime.colname = Datetime,
  type = c("round", "floor", "ceiling"),
  numeric.handler = mean,
  character.handler = function(x) names(which.max(table(x, useNA = "ifany"))),
  logical.handler = function(x) mean(x) >= 0.5,
  factor.handler = function(x) factor(names(which.max(table(x, useNA = "ifany")))),
  datetime.handler = mean,
  duration.handler = function(x) lubridate::duration(mean(x)),
  time.handler = function(x) hms::as_hms(mean(x)),
  ...
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`unit` Unit of binning. See `lubridate::round_date()` for examples. The default is `"dominant.epoch"`, which means everything will be aggregated to the most common interval. This is especially useful for slightly irregular data, but can be computationally expensive. `"none"` will not aggregate the data at all.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`type` One of `"round"`(the default), `"ceiling"` or `"floor"`. Setting chooses the relevant function from **lubridate**.

`numeric.handler`, `character.handler`, `logical.handler`, `factor.handler`, `datetime.handler`, `duration.handler`, `time.handler` functions that handle the respective data types. The default handlers calculate the `mean` or `median` for `numeric`, `POSIXct`, `duration`, and `hms`, and the `mode` for `character`, `factor` and `logical` types.

`...` arguments given over to `dplyr::summarize()` to handle columns that do not fall into one of the categories above.

Details

Summary values for type `POSIXct` are calculated as the mean, which can be nonsensical at times (e.g., the mean of Day1 18:00 and Day2 18:00, is Day2 6:00, which can be the desired result, but if the focus is on time, rather then on datetime, it is recommended that values are converted to times via `hms::as_hms()` before applying the function (the mean of 18:00 and 18:00 is still 18:00, not 6:00).

Value

A `tibble` with aggregated `Datetime` data. Usually the number of rows will be smaller than the input `dataset`. If the handler arguments capture all column types, the number of columns will be the same as in the input `dataset`.

Examples

```
#dominant epoch without aggregation
sample.data.environment %>%
 dominant_epoch()

#dominant epoch with 5 minute aggregation
sample.data.environment %>%
 aggregate_Datetime(unit = "5 mins") %>%
 dominant_epoch()

#dominant epoch with 1 day aggregation
sample.data.environment %>%
 aggregate_Datetime(unit = "1 day") %>%
 dominant_epoch()
```

---

alphaopic.action.spectra

Alphaopic (+ photopic) action spectra

Description

A dataframe of alphaopic action spectra plus the photopic action spectrum. The alphaopic action spectra are according to the CIE S 026/E:2018 standard. The alphaopic action spectra are for a 32-year-old standard observer. The photopic action spectrum is for a 2° standard observer.

Usage

```
alphaopic.action.spectra
```

Format

`alphaopic.action.spectra` A datafram with 471 rows and 7 columns:

wavelength integer of wavelength, from 360 to 830 nm. Unit is nm

melanopic numeric melanopic action spectrum

l_cone_opic numeric L-cone opic action spectrum

m_cone_opic numeric M-cone opic action spectrum

s_cone_opic numeric S-cone opic action spectrum

rhodopic numeric rhodopic action spectrum

photopic numeric photopic action spectrum

Source

https://www.cie.co.at/publications/cie-system-metrology-optical-radiation-iprgc-influenced-response

https://cie.co.at/datatable/cie-spectral-luminous-efficiency-photopic-vision

<https://files.cie.co.at/CIE S 026 alpha-opic Toolbox.xlsx>

References

CIE (2019). ISO/CIE 11664-1:2019(E). Colorimetry — Part 1: CIE standard colorimetric observers. Vienna, CIE

CIE (2018). CIE S 026/E:2018. CIE system for metrology of optical radiation for ipRGC-influenced responses of light. Vienna, CIE

`barroso_lighting_metrics` *Circadian lighting metrics from Barroso et al. (2014)*

Description

This function calculates the metrics proposed by Barroso et al. (2014) for light-dosimetry in the context of research on the non-visual effects of light. The following metrics are calculated:

Usage

```
barroso_lighting_metrics(
  Light.vector,
  Time.vector,
  epoch = "dominant.epoch",
  loop = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.
`loop`	Logical. Should the data be looped? Defaults to `FALSE`.
`na.rm`	Logical. Should missing values (NA) be removed for the calculation? Defaults to `FALSE`. If `TRUE`, for the calculation of `bright_cluster` and `dark_cluster`, missing values will be replaced by 0 (see `period_above_threshold`).
`as.df`	Logical. Should a data frame be returned? If `TRUE`, a data frame with seven columns will be returned. Defaults to `FALSE`.

Details

`bright_threshold` The maximum light intensity for which at least six hours of measurements are at the same or higher level.

`dark_threshold` The minimum light intensity for which at least eight hours of measurements are at the same or lower level.

`bright_mean_level` The 20% trimmed mean of all light intensity measurements equal or above the `bright_threshold`.

`dark_mean_level` The 20% trimmed mean of all light intensity measurements equal or below the `dark_threshold`.

`bright_cluster` The longest continuous time interval above the `bright_threshold`.

`dark_cluster` The longest continuous time interval below the `dark_threshold`.

`circadian_variation` A measure of periodicity of the daily lighting schedule over a given set of days. Calculated as the coefficient of variation of input light data.

Value

List or dataframe with the seven values: `bright_threshold`, `dark_threshold`, `bright_mean_level`, `dark_mean_level`, `bright_cluster`, `dark_cluster`, `circadian_variation`. The output type of `bright_cluster`, `dark_cluster`, is a duration object.

References

Barroso, A., Simons, K., & Jager, P. de. (2014). Metrics of circadian lighting for clinical investigations. *Lighting Research & Technology*, 46(6), 637–649. doi:10.1177/1477153513502664

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("B", 60 * 24),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*24-1)),
    MEDI = c(rep(sample(seq(0,1,0.1), 60*8, replace = TRUE)),
             rep(sample(1:1000, 16, replace = TRUE), each = 60))
  )

dataset1 %>%
  dplyr::reframe(barroso_lighting_metrics(MEDI, Datetime, as.df = TRUE))
```

`bright_dark_period` *Brightest or darkest continuous period*

Description

This function finds the brightest or darkest continuous period of a given timespan and calculates its `mean` light level, as well as the timing of the period's `onset`, `midpoint`, and `offset`. It is defined as the period with the maximum or minimum mean light level. Note that the data need to be regularly spaced (i.e., no gaps) for correct results.

Usage

```
bright_dark_period(
  Light.vector,
  Time.vector,
  period = c("brightest", "darkest"),
  timespan = "10 hours",
  epoch = "dominant.epoch",
  loop = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`period`	String indicating the type of period to look for. Can be either `"brightest"`(the default) or `"darkest"`.
`timespan`	The timespan across which to calculate. Can be either a duration or a duration string, e.g., `"1 day"` or `"10 sec"`.
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.
`loop`	Logical. Should the data be looped? If `TRUE`, a full copy of the data will be concatenated at the end of the data. Makes only sense for 24 h data. Defaults to `FALSE`.
`na.rm`	Logical. Should missing values be removed for the calculation? Defaults to `FALSE`.
`as.df`	Logical. Should the output be returned as a data frame? Defaults to `TRUE`.

Details

Assumes regular 24h light data. Otherwise, results may not be meaningful. Looping the data is recommended for finding the darkest period.

Value

A named list with the mean, onset, midpoint, and offset of the calculated brightest or darkest period, or if as.df == TRUE a data frame with columns named {period}_{timespan}_{metric}. The output type corresponds to the type of Time.vector, e.g., if Time.vector is HMS, the timing metrics will be also HMS, and vice versa for POSIXct.

References

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: centroidLE(), disparity_index(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
# Dataset with light > 250lx between 06:00 and 18:00
dataset1 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::as_datetime(0) + lubridate::hours(0:23),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )

dataset1 %>%
  dplyr::reframe(bright_dark_period(MEDI, Datetime, "brightest", "10 hours",
    as.df = TRUE))
dataset1 %>%
  dplyr::reframe(bright_dark_period(MEDI, Datetime, "darkest", "7 hours",
    loop = TRUE, as.df = TRUE))

# Dataset with duration as Time.vector
dataset2 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::dhours(0:23),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )

dataset2 %>%
  dplyr::reframe(bright_dark_period(MEDI, Datetime, "brightest", "10 hours",
                                    as.df = TRUE))
dataset2 %>%
  dplyr::reframe(bright_dark_period(MEDI, Datetime, "darkest", "5 hours",
                                    loop = TRUE, as.df = TRUE))
```

Brown2reference — *Add Brown et al. (2022) reference illuminance to a dataset*

Description

Adds several columns to a light logger dataset. It requires a column that contains the Brown states, e.g. "daytime", "evening", and "night". From that the function will add a column with the recommended illuminance, a column that checks if the illuminance of the dataset is within the recommended illuminance levels, and a column that gives a label to the reference.

Usage

```
Brown2reference(
  dataset,
  MEDI.colname = MEDI,
  Brown.state.colname = State.Brown,
  Brown.rec.colname = Reference,
  Reference.label = "Brown et al. (2022)",
  overwrite = FALSE,
  ...
)
```

Arguments

`dataset` A dataframe that contains a column with the Brown states

`MEDI.colname` The name of the column that contains the MEDI values which are used for checks against the Brown reference illuminance. Must be part of the dataset.

`Brown.state.colname` The name of the column that contains the Brown states. Must be part of the dataset.

`Brown.rec.colname` The name of the column that will contain the recommended illuminance. Must not be part of the dataset, otherwise it will throw an error.

`Reference.label` The label that will be used for the reference. Expects a `character` scalar.

`overwrite` If `TRUE` (defaults to `FALSE`), the function will overwrite the `Brown.rec.colname` column if it already exists.

`...` Additional arguments that will be passed to `Brown_rec()` and `Brown_check()`. This is only relevant to correct the names of the daytime states or the thresholds used within these states. See the documentation of these functions for more information.

Details

On a lower level, the function uses `Brown_rec()` and `Brown_check()` to create the required information.

Value

A dataframe on the basis of the `dataset` that contains the added columns.

References

https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001571

See Also

Other Brown: `Brown_check()`, `Brown_cut()`, `Brown_rec()`, `sleep_int2Brown()`

Examples

```
#add Brown reference illuminance to some sample data
testdata <- tibble::tibble(MEDI = c(100, 10, 1, 300),
                  State.Brown = c("day", "evening", "night", "day"))
Brown2reference(testdata)
```

`Brown_check` *Check whether a value is within the recommended illuminance/MEDI levels by Brown et al. (2022)*

Description

This is a lower level function. It checks a given value against a threshold for the states given by Brown et al. (2022). The function is vectorized. For `day` the threshold is a lower limit, for `evening` and `night` the threshold is an upper limit.

Usage

```
Brown_check(
  value,
  state,
  Brown.day = "day",
  Brown.evening = "evening",
  Brown.night = "night",
  Brown.day.th = 250,
  Brown.evening.th = 10,
  Brown.night.th = 1
)
```

Arguments

value	Illuminance value to check against the recommendation. needs to be numeric, can be a vector.
state	The state from Brown et al. (2022). Needs to be a character vector with the same length as value.
Brown.day, Brown.evening, Brown.night	The names of the states from Brown et al. (2022). These are the default values ("day", "evening", "night"), but can be changed if the names in state are different. Needs to be a character scalar.
Brown.day.th, Brown.evening.th, Brown.night.th	The thresholds for the states from Brown et al. (2022). These are the default values (250, 10, 1), but can be changed if the thresholds should be different. Needs to be a numeric scalar.

Value

A logical vector with the same length as value that indicates whether the value is within the recommended illuminance levels.

References

https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001571

See Also

Other Brown: Brown2reference(), Brown_cut(), Brown_rec(), sleep_int2Brown()

Examples

```
states <- c("day", "evening", "night", "day")
values <- c(100, 10, 1, 300)
Brown_check(values, states)
Brown_check(values, states, Brown.day.th = 100)
```

Brown_cut — *Create a state column that cuts light levels into sections by Brown et al. (2022)*

Description

This is a convenience wrapper arount cut() and dplyr::mutate(). It creates a state column dividing a light column into recommended levels by Brown et al. (2022). Cuts can be adjusted or extended with vector_cuts and vector_labels

Usage

```
Brown_cut(
  dataset,
  MEDI.colname = MEDI,
  New.state.colname = state,
  vector_cuts = c(-Inf, 1, 10, 250, Inf),
  vector_labels = "default",
  overwrite = TRUE
)
```

Arguments

`dataset`	A light exposure dataframe
`MEDI.colname`	The colname containing melanopic EDI values (or, alternatively, Illuminance). Defaults to `MEDI`. Expects a symbol.
`New.state.colname`	Name of the new column that will contain the cut data. Expects a symbol.
`vector_cuts`	Numeric vector of breaks for the cuts.
`vector_labels`	Vector of labels for the cuts. Must be one entry shorter than `vector_cuts`. `"default"` will produce nice labels for the default setting of `vector_cuts` (and throw an error otherwise).
`overwrite`	Logical. Should the `New.state.colname` overwrite a preexisting column in the dataset

Value

The input dataset with an additional (or overwritten) column containing a cut light vector

References

https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001571

See Also

Other Brown: `Brown2reference()`, `Brown_check()`, `Brown_rec()`, `sleep_int2Brown()`

Examples

```
sample.data.environment |>
Brown_cut(vector_labels = c("0-1lx", "1-10lx", "10-250lx", "250lx-Inf")) |>
dplyr::count(state)
```

Brown_rec *Set the recommended illuminance/MEDI levels by Brown et al. (2022)*

Description

This is a lower level function. It sets the recommended illuminance/MEDI levels by Brown et al. (2022) for a given state. The function is vectorized.

Usage

```
Brown_rec(
  state,
  Brown.day = "day",
  Brown.evening = "evening",
  Brown.night = "night",
  Brown.day.th = 250,
  Brown.evening.th = 10,
  Brown.night.th = 1
)
```

Arguments

`state` The state from Brown et al. (2022). Needs to be a character vector.

`Brown.day`, `Brown.evening`, `Brown.night`
The names of the states from Brown et al. (2022). These are the default values (`"day"`, `"evening"`, `"night"`), but can be changed if the names in `state` are different. Needs to be a character scalar.

`Brown.day.th`, `Brown.evening.th`, `Brown.night.th`
The thresholds for the states from Brown et al. (2022). These are the default values (`250`, `10`, `1`), but can be changed if the thresholds should be different. Needs to be a numeric scalar.

Value

A dataframe with the same length as `state` that contains the recommended illuminance/MEDI levels.

References

https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001571

See Also

Other Brown: `Brown2reference()`, `Brown_check()`, `Brown_cut()`, `sleep_int2Brown()`

Examples

```
states <- c("day", "evening", "night")
Brown_rec(states)
Brown_rec(states, Brown.day.th = 100)
```

---

centroidLE *Centroid of light exposure*

Description

This function calculates the centroid of light exposure as the mean of the time vector weighted in proportion to the corresponding binned light intensity.

Usage

```
centroidLE(
  Light.vector,
  Time.vector,
  bin.size = NULL,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`bin.size`	Value specifying size of bins to average the light data over. Must be either a duration or a duration string, e.g., `"1 day"` or `"10 sec"`. If nothing is provided, no binning will be performed.
`na.rm`	Logical. Should missing values be removed for the calculation? Defaults to `FALSE`.
`as.df`	Logical. Should the output be returned as a data frame? If `TRUE`, a data frame with a single column named `centroidLE` will be returned. Defaults to `FALSE`.

Value

Single column data frame or vector.

References

Phillips, A. J. K., Clerx, W. M., O'Brien, C. S., Sano, A., Barger, L. K., Picard, R. W., Lockley, S. W., Klerman, E. B., & Czeisler, C. A. (2017). Irregular sleep/wake patterns are associated with poorer academic performance and delayed circadian and sleep/wake timing. *Scientific Reports*, 7(1), 3216. doi:10.1038/s41598017031714

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: bright_dark_period(), disparity_index(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
# Dataset with POSIXct time vector
dataset1 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::as_datetime(0) + lubridate::hours(0:23),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset1 %>%
  dplyr::reframe(
    "Centroid of light exposure" = centroidLE(MEDI, Datetime, "2 hours")
  )

# Dataset with hms time vector
dataset2 <-
  tibble::tibble(
    Id = rep("A", 24),
    Time = hms::as_hms(lubridate::as_datetime(0) + lubridate::hours(0:23)),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset2 %>%
  dplyr::reframe(
    "Centroid of light exposure" = centroidLE(MEDI, Time, "2 hours")
  )

# Dataset with duration time vector
dataset3 <-
  tibble::tibble(
    Id = rep("A", 24),
    Hour = lubridate::duration(0:23, "hours"),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset3 %>%
  dplyr::reframe(
    "Centroid of light exposure" = centroidLE(MEDI, Hour, "2 hours")
  )
```

---

count_difftime — *Counts the Time differences (epochs) per group (in a grouped dataset)*

Description

Counts the Time differences (epochs) per group (in a grouped dataset)

Usage

```
count_difftime(dataset, Datetime.colname = Datetime)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

Value

a `tibble` with the number of occurences of each time difference per group

Examples

```
#count_difftime returns the number of occurences of each time difference
#and is more comprehensive in terms of a summary than `gap_finder` or
#`dominant_epoch`
count_difftime(sample.data.irregular)
dominant_epoch(sample.data.irregular)
gap_finder(sample.data.irregular)

#irregular data can be regularized with `aggregate_Datetime`
sample.data.irregular |>
 aggregate_Datetime(unit = "15 secs") |>
 count_difftime()
```

`create_Timedata` *create_Timedata*

Description

create_Timedata

Usage

```
create_Timedata(...)
```

Arguments

`...` Input arguments to `add_Time_col()`

Value

a data.frame object identical to dataset but with the added column of Time-of-Day data, or a vector with the Time-of-Day-data

Examples

```
sample.data.environment %>% create_Timedata()
```

---

cut_Datetime *Create Datetime bins for visualization and calculation*

Description

cut_Datetime is a wrapper around lubridate::round_date() (and friends) combined with dplyr::mutate(), to create a new column in a light logger dataset with a specified binsize. This can be "3 hours", "15 secs", or "0.5 days". It is a useful step between a dataset and a visualization or summary step.

Usage

```
cut_Datetime(
  dataset,
  unit = "3 hours",
  type = c("round", "floor", "ceiling"),
  Datetime.colname = Datetime,
  New.colname = Datetime.rounded,
  group_by = FALSE,
  ...
)
```

Arguments

dataset	A light logger dataset. Expects a dataframe. If not imported by LightLogR, take care to choose a sensible variable for the Datetime.colname.
unit	Unit of binning. See lubridate::round_date() for examples. The default is "3 hours".
type	One of "round"(the default), "ceiling" or "floor". Setting chooses the relevant function from **lubridate**.
Datetime.colname	column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol. Needs to be part of the dataset. Must be of type POSIXct.
New.colname	Column name for the added column in the dataset.
group_by	Should the data be grouped by the new column? Defaults to FALSE
...	Parameter handed over to lubridate::round_date() and siblings

Value

a `data.frame` object identical to `dataset` but with the added column of binned datetimes.

Examples

```
#compare Datetime and Datetime.rounded
sample.data.environment %>%
  cut_Datetime() %>%
  dplyr::slice_sample(n = 5)
```

`data2reference` *Create reference data from other data*

Description

Create reference data from almost any other data that has a datetime column and a data column. The reference data can even be created from subsets of the same data. Examples are that one participant can be used as a reference for all other participants, or that the first (second,...) day of every participant data is the reference for any other day. **This function needs to be carefully handled, when the reference data time intervals are shorter than the data time intervals. In that case, use `aggregate_Datetime()` on the reference data beforehand to lengthen the interval.**

Usage

```
data2reference(
  dataset,
  Reference.data = dataset,
  Datetime.column = Datetime,
  Data.column = MEDI,
  Id.column = Id,
  Reference.column = Reference,
  overwrite = FALSE,
  filter.expression.reference = NULL,
  across.id = FALSE,
  shift.start = FALSE,
  length.restriction.seconds = 60,
  shift.intervals = "auto",
  Reference.label = NULL
)
```

Arguments

`dataset`	A light logger dataset
`Reference.data`	The data that should be used as reference. By default the `dataset` will be used as reference.

Datetime.column
: Datetime column of the dataset and Reference.data. Need to be the same in both sets. Default is Datetime.

Data.column
: Data column in the Reference.data that is then converted to a reference. Default is MEDI.

Id.column
: Name of the Id.column in both the dataset and the Reference.data.

Reference.column
: Name of the reference column that will be added to the dataset. Default is Reference. Cannot be the same as any other column in the dataset and will throw an error if it is.

overwrite
: If TRUE (defaults to FALSE), the function will overwrite the Reference.colname column if it already exists.

filter.expression.reference
: Expression that is used to filter the Reference.data before it is used as reference. Default is NULL. See

across.id
: Grouping variables that should be ignored when creating the reference data. Default is FALSE. If TRUE, all grouping variables are ignored. If FALSE, no grouping variables are ignored. If a vector of grouping variables is given, these are ignored.

shift.start
: If TRUE, the reference data is shifted to the start of the respective group. Default is FALSE. The shift ignores the groups specified in across.id.

length.restriction.seconds
: Restricts the application of reference data to a maximum length in seconds. Default is 60 seconds. This is useful to avoid reference data being applied to long periods of time, e.g., when there are gaps in the reference data

shift.intervals
: Time shift in seconds, that is applied to every data point in the reference data. Default is "auto". If "auto", the shift is calculated by halving the most frequent time difference between two data points in the reference data. If a number is given, this number in seconds is used as the shift. Can also use lubridate::duration() to specify the shift.

Reference.label
: Label that is added to the reference data. If NULL, no label is added.

Details

To use subsets of data, use the filter.expression.reference argument to specify the subsets of data. The across.id argument specifies whether the reference data should be used across all or some grouping variables (e.g., across participants). The shift.start argument enables a shift of the reference data start time to the start of the respective group.

and @examples for more information. The expression is evaluated within dplyr::filter().

Value

A dataset with a new column Reference that contains the reference data.

Examples

```
library(dplyr)
library(lubridate)
library(ggplot2)

gg_reference <- function(dataset) {
dataset %>%
ggplot(aes(x = Datetime, y = MEDI, color = Id)) +
geom_line(linewidth = 1) +
geom_line(aes(y = Reference), color = "black", size = 0.25, linetype = "dashed") +
theme_minimal() + facet_wrap(~ Id, scales = "free_y")
}

#in this example, each data point is its own reference
sample.data.environment %>%
  data2reference() %>%
  gg_reference()

#in this example, the first day of each ID is the reference for the other days
#this requires grouping of the Data by Day, which is then specified in across.id
#also, shift.start needs to be set to TRUE, to shift the reference data to the
#start of the groupings
sample.data.environment %>% group_by(Id, Day = as_date(Datetime)) %>%
data2reference(
  filter.expression.reference =  as_date(Datetime) == min(as_date(Datetime)),
  shift.start = TRUE,
  across.id = "Day") %>%
  gg_reference()

#in this example, the Environment Data will be used as a reference
sample.data.environment %>%
data2reference(
  filter.expression.reference =  Id == "Environment",
  across.id = TRUE) %>%
  gg_reference()
```

`Datetime2Time` *Convert Datetime columns to Time columns*

Description

Convert Datetime columns to Time columns

Usage

```
Datetime2Time(
  dataset,
  cols = dplyr::where(lubridate::is.POSIXct),
  silent = FALSE
)
```

Arguments

dataset	A data.frame with POSIXct columns.
cols	The column names to convert. Expects a symbol. The default will convert all POSIXct columns. If uncertain whether columns exist in the dataset, use dplyr::any_of().
silent	Logical on whether no message shall be shown if input and output are identical. Defaults to FALSE (i.e., a message is shown).

Value

The input dataset with converted POSIXct columns as time (hms) columns. With the default settings, if no POSIXct column exists, input and output will be identical.

Examples

```
sample.data.environment |> Datetime2Time()
#more than one POSIX col
sample.data.environment |>
  dplyr::mutate(Datetime2 = lubridate::POSIXct(1)) |>
  Datetime2Time()
#only converting one of them
sample.data.environment |>
  dplyr::mutate(Datetime2 = lubridate::POSIXct(1)) |>
  Datetime2Time(Datetime)
#if uncertain whether column exists
sample.data.environment |>
  Datetime2Time(dplyr::any_of("Datetime3"))
```

Datetime_breaks *Create a (shifted) sequence of Datetimes for axis breaks*

Description

Take a vector of Datetimes and create a sequence of Datetimes with a given shift and interval. This is a helper function to create breaks for plotting, e.g. in gg_days(), and is best used in conjunction with Datetime_limits(). The function is a thin wrapper around seq().

Usage

```
Datetime_breaks(x, shift = lubridate::duration(12, "hours"), by = "1 day")
```

Arguments

x	a vector of Datetimes
shift	a numeric giving the number of duration object, e.g. lubridate::duration(12, "hours")
by	a character scalar giving the unit of the interval in base::seq()

Value

a vector of `Datetimes`

Examples

```
dataset <- c("2023-08-15", "2023-08-20")
Datetime_breaks(dataset)
Datetime_breaks(dataset, shift = 0)
Datetime_breaks(dataset, by = "12 hours")
```

`Datetime_limits` *Find or set sensible limits for Datetime axis*

Description

Take a vector of `Datetimes` and return the start of the first and end of the last day of data. The `start` and the `length` can be adjusted by `durations`, like `lubridate::ddays()`. It is used in the `gg_days()` function to return a sensible x-axis. This function is a thin wrapper around `lubridate::floor_date()` and `lubridate::ceiling_date()`.

Usage

```
Datetime_limits(
  x,
  start = NULL,
  length = NULL,
  unit = "1 day",
  midnight.rollover = FALSE,
  ...
)
```

Arguments

`x`	a vector of `Datetimes`
`start`	optional `duration` object, e.g. `lubridate::ddays(1)` that shifts the start of the `Datetime` vector by this amount.
`length`	optional `duration` object, e.g. `lubridate::ddays(7)` that shifts the end of the `Datetime` vector by this amount from the (adjusted) start. Depending on the data, you might have to subtract one day from the desired length to get the correct axis-scaling if you start at midnight.
`unit`	a `character` scalar giving the unit of rounding in `lubridate::floor_date()` and `lubridate::ceiling_date()`
`midnight.rollover`	a `logical` scalar indicating whether to rollover in cases of exact matches of rounded values and input values. Helpful if some cases fall exactly on the rounded values and others don't.
`...`	other arguments passed to `lubridate::floor_date()` and `lubridate::ceiling_date()`

Value

a 2 item vector of Datetimes with the (adjusted) start and end of the input vector.

Examples

```
dataset <- c("2023-08-15", "2023-08-20")
breaks <- Datetime_breaks(dataset)
Datetime_limits(breaks)
Datetime_limits(breaks, start = lubridate::ddays(1))
Datetime_limits(breaks, length = lubridate::ddays(2))
```

---

disparity_index   *Disparity index*

Description

This function calculates the continuous disparity index as described in Fernández-Martínez et al. (2018).

Usage

```
disparity_index(Light.vector, na.rm = FALSE, as.df = FALSE)
```

Arguments

Light.vector	Numeric vector containing the light data.
na.rm	Logical. Should missing values be removed? Defaults to FALSE
as.df	Logical. Should the output be returned as a data frame? If TRUE, a data frame with a single column named disparity_index will be returned. Defaults to FALSE.

Value

Single column data frame or vector.

References

Fernández-Martínez, M., Vicca, S., Janssens, I. A., Carnicer, J., Martín-Vide, J., & Peñuelas, J. (2018). The consecutive disparity index, D: A measure of temporal variability in ecological studies. *Ecosphere*, 9(12), e02527. doi:10.1002/ecs2.2527

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: bright_dark_period(), centroidLE(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::as_datetime(0) + lubridate::hours(0:23),
    MEDI = sample(0:1000, 24),
  )
dataset1 %>%
  dplyr::reframe(
    "Disparity index" = disparity_index(MEDI)
  )
```

---

dominant_epoch *Determine the dominant epoch/interval of a dataset*

Description

Calculate the dominant epoch/interval of a dataset. The dominant epoch/interval is the epoch/interval that is most frequent in the dataset. The calculation is done per group, so that you might get multiple variables. If two or more epochs/intervals are equally frequent, the first one (shortest one) is chosen.

Usage

```
dominant_epoch(dataset, Datetime.colname = Datetime)
```

Arguments

`dataset`	A light logger dataset. Needs to be a dataframe.
`Datetime.colname`	The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

Value

A `tibble` with one row per group and a column with the `dominant.epoch` as a `lubridate::duration()`. Also a column with the `group.indices`, which is helpful for referencing the `dominant.epoch` across dataframes of equal grouping.

See Also

Other regularize: `extract_gaps()`, `gap_finder()`, `gap_handler()`, `gapless_Datetimes()`, `has_gaps()`, `has_irregulars()`

Examples

```
dataset <-
tibble::tibble(Id = c("A", "A", "A", "B", "B", "B"),
               Datetime = lubridate::as_datetime(1) +
                          lubridate::days(c(0:2, 4, 6, 8)))
dataset
#get the dominant epoch by group
dataset %>%
dplyr::group_by(Id) %>%
dominant_epoch()

#get the dominant epoch of the whole dataset
dataset %>%
dominant_epoch()
```

---

dose — *Calculate the dose (value·hours)*

Description

This function calculates the dose from a time series. For light, this is equal to the actual definition of light exposure (CIE term luminous exposure). Output will always be provided in value·hours (e.g., for light, lx·hours).

Usage

```
dose(
  Light.vector,
  Time.vector,
  epoch = "dominant.epoch",
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.

`na.rm`	Logical. Should missing values (NA) be removed for the calculation? Defaults to `FALSE`.
`as.df`	Logical. Should a data frame with be returned? If `TRUE`, a data frame with a single column named `dose` will be returned. Defaults to `FALSE`.

Details

The time series does not have to be regular, however, it will be aggregated to a regular timeseries of the given epoch. Implicit gaps (i.e., no observations), will be converted to NA values (which can be ignored with `na.rm = TRUE`).

Value

A numeric object as single value, or single column data frame with the dose in value·hours

References

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
dose(c(1,1,1,1), lubridate::dhours(c(1:4)), na.rm = TRUE)
#with gaps
dose(c(1,1,1), lubridate::dhours(c(1,3:4)), na.rm = TRUE)
#gaps can be aggregated to a coarser interval, which can be sensibe
#if they are still representative
dose(c(1,1,1), lubridate::dhours(c(1,3:4)), na.rm = TRUE, epoch = "2 hours")
```

`dst_change_handler` *Handle jumps in Daylight Savings (DST) that are missing in the data*

Description

When data is imported through `LightLogR` and a timezone applied, it is assumed that the timestamps are correct - which is the case, e.g., if timestamps are stored in `UTC`, or they are in local time. Some if not most measurement devices are set to local time before a recording interval starts. If during the recording a daylight savings jump happens (in either direction), the device might not adjust timestamps for this change. This results in an unwanted shift in the data, starting at the time of the DST jump and likely continues until the end of a file. `dst_change_handler` is used to detect such jumps within a group and apply the correct shift in the data (i.e., the shift that should have been applied by the device).

important Note that this function is only useful if the time stamp in the raw data deviates from the actual date-time. Note also, that this function results in a gap during the DST jump, which should be handled by `gap_handler()` afterwards. It will also result in potentially double the timestamps during the jum back from DST to standard time. This will result in some inconsistencies with some functions, so we recommend to use `aggregate_Datetime()` afterwards with a `unit` equal to the dominant epoch. Finally, the function is not equipped to handle more than one jump per group. The jump is based on whether the group starts out with DST or not. **the function will remove datetime rows with `NA` values**.

Usage

```
dst_change_handler(
  dataset,
  Datetime.colname = Datetime,
  filename.colname = NULL
)
```

Arguments

`dataset` dataset to be summarized, must be a `dataframe`

`Datetime.colname`
name of the column that contains the Datetime data, expects a `symbol`

`filename.colname`
(optional) column name that contains the filename. If provided, it will use this column as a temporary grouping variable additionally to the `dataset` grouping.

Details

The detection of a DST jump is based on the function `lubridate::dst()` and jumps are only applied within a group. During import, this function is used if `dst_adjustment = TRUE` is set and includes by default the filename as the grouping variable, additionally to `Id`.

Value

A `tibble` with the same columns as the input dataset, but shifted

See Also

Other DST: `dst_change_summary()`

Examples

```
#create some data that crosses a DST jump
data <-
 tibble::tibble(
 Datetime = seq.POSIXt(from = as.POSIXct("2023-03-26 01:30:00", tz = "Europe/Berlin"),
                     to = as.POSIXct("2023-03-26 03:00:00", tz = "Europe/Berlin"),
                     by = "30 mins"),
```

```
                   Value = 1)
#as can be seen next, there is a gap in the data - this is necessary when
#using a timezone with DST.
data$Datetime
#Let us say now, that the device did not adjust for the DST - thus the 03:00
#timestamp is actually 04:00 in local time. This can be corrected for by:
data %>% dst_change_handler() %>% .$Datetime
```

`dst_change_summary` *Get a summary of groups where a daylight saving time change occurs.*

Description

Get a summary of groups where a daylight saving time change occurs.

Usage

```
dst_change_summary(dataset, Datetime.colname = Datetime)
```

Arguments

`dataset` dataset to be summarized, must be a `dataframe`

`Datetime.colname` name of the column that contains the Datetime data, expects a `symbol`

Value

a `tibble` with the groups where a dst change occurs. The column `dst_start` is a boolean that indicates whether the start of this group occurs during daylight savings.

See Also

Other DST: `dst_change_handler()`

Examples

```
sample.data.environment %>%
  dplyr::mutate(Datetime =
  Datetime + lubridate::dweeks(8)) %>%
  dst_change_summary()
```

durations *Calculate duration of data in each group*

Description

This function calculates the total duration of data in each group of a dataset, based on a datetime column and a variable column. It uses the dominant epoch (interval) of each group to calculate the duration.

Usage

```
durations(
  dataset,
  Variable.colname = Datetime,
  Datetime.colname = Datetime,
  count.NA = FALSE,
  show.missing = FALSE,
  show.interval = FALSE,
  FALSE.as.NA = FALSE
)
```

Arguments

`dataset` A light logger dataset. Expects a dataframe. If not imported by LightLogR, take care to choose sensible variables for the Datetime.colname and Variable.colname.

`Variable.colname` Column name that contains the variable for which to calculate the duration. Expects a symbol. Needs to be part of the dataset.

`Datetime.colname` Column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol. Needs to be part of the dataset. Must be of type POSIXct.

`count.NA` Logical. Should NA values in Variable.colname be counted as part of the duration? Defaults to FALSE.

`show.missing` Logical. Should the duration of NAs be provided in a separate column "Missing"? Defaults to FALSE.

`show.interval` Logical. Should the dominant epoch (interval) be shown in a column "interval"? Defaults to FALSE.

`FALSE.as.NA` Logical. Should FALSE values in the Variable.colname be treated as NA (i.e., missing)?

Value

A tibble with one row per group and a column "duration" containing the duration of each group as a `lubridate::duration()`. If `show.missing = TRUE`, a column "missing" is added with the duration of NAs, and a column "total" with the total duration. If `show.interval = TRUE`, a column "interval" is added with the dominant epoch of each group.

Examples

```
# Calculate the duration of a dataset
durations(sample.data.environment)

# create artificial gaps in the data
gapped_data <-
sample.data.environment |>
  dplyr::filter(MEDI >= 10) |>
  gap_handler(full.days = TRUE)

#by default, the Datetime column is selected for the `Variable.colname`,
#basically ignoring NA measurement values
gapped_data |>
 durations(count.NA = TRUE)

# Calculate the duration where MEDI are available
durations(gapped_data, MEDI)

# Calculate the duration, show the duration of NAs separately
durations(gapped_data, MEDI, show.missing = TRUE)

# Calculate the duration, show the dominant epoch
durations(gapped_data, Variable.colname = MEDI, show.interval = TRUE)

# Calculate durations for day and night separately
gapped_data |>
  add_photoperiod(coordinates = c(48.52, 9.06)) |>
  dplyr::group_by(photoperiod.state, .add = TRUE) |>
  durations(Variable.colname = MEDI, show.interval = TRUE, show.missing = TRUE)
```

`duration_above_threshold`
Duration above/below threshold or within threshold range

Description

This function calculates the duration spent above/below a specified threshold light level or within a specified range of light levels.

Usage

```
duration_above_threshold(
  Light.vector,
  Time.vector,
  comparison = c("above", "below"),
  threshold,
  epoch = "dominant.epoch",
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

Light.vector	Numeric vector containing the light data.
Time.vector	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
comparison	String specifying whether the time above or below threshold should be calculated. Can be either "above" (the default) or "below". If two values are provided for threshold, this argument will be ignored.
threshold	Single numeric value or two numeric values specifying the threshold light level(s) to compare with. If a vector with two values is provided, the time within the two thresholds will be calculated.
epoch	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either "dominant.epoch" (the default) for a guess based on the data, or a valid duration string, e.g., "1 day" or "10 sec".
na.rm	Logical. Should missing values (NA) be removed for the calculation? Defaults to FALSE.
as.df	Logical. Should a data frame with be returned? If TRUE, a data frame with a single column named duration_{comparison}_{threshold} will be returned. Defaults to FALSE.

Value

A duration object as single value, or single column data frame.

References

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: bright_dark_period(), centroidLE(), disparity_index(), dose(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
N <- 60
# Dataset with epoch = 1min
dataset1 <-
  tibble::tibble(
    Id = rep("A", N),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(1:N),
    MEDI = sample(c(sample(1:249, N / 2), sample(250:1000, N / 2))),
  )
# Dataset with epoch = 30s
dataset2 <-
  tibble::tibble(
    Id = rep("B", N),
```

```
    Datetime = lubridate::as_datetime(0) + lubridate::seconds(seq(30, N * 30, 30)),
    MEDI = sample(c(sample(1:249, N / 2), sample(250:1000, N / 2))),
  )
dataset.combined <- rbind(dataset1, dataset2)

dataset1 %>%
 dplyr::reframe("TAT >250lx" = duration_above_threshold(MEDI, Datetime, threshold = 250))

dataset1 %>%
 dplyr::reframe(duration_above_threshold(MEDI, Datetime, threshold = 250, as.df = TRUE))

# Group by Id to account for different epochs
dataset.combined %>%
  dplyr::group_by(Id) %>%
 dplyr::reframe("TAT >250lx" = duration_above_threshold(MEDI, Datetime, threshold = 250))
```

`exponential_moving_average` *Exponential moving average filter (EMA)*

Description

This function smoothes the data using an exponential moving average filter with a specified decay half-life.

Usage

```
exponential_moving_average(
  Light.vector,
  Time.vector,
  decay = "90 min",
  epoch = "dominant.epoch"
)
```

Arguments

`Light.vector`	Numeric vector containing the light data. Missing values are replaced by 0.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`decay`	The decay half-life controlling the exponential smoothing. Can be either a duration or a string. If it is a string, it needs to be a valid duration string, e.g., `"1 day"` or `"10 sec"`. The default is set to `"90 mins"` for a biologically relevant estimate (see the reference paper).
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.

Details

The timeseries is assumed to be regular. Missing values in the light data will be replaced by 0.

Value

A numeric vector containing the smoothed light data. The output has the same length as `Light.vector`.

References

Price, L. L. A. (2014). On the Role of Exponential Smoothing in Circadian Dosimetry. *Photochemistry and Photobiology*, 90(5), 1184-1192. doi:10.1111/php.12282

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
sample.data.environment.EMA = sample.data.environment %>%
  dplyr::filter(Id == "Participant") %>%
  filter_Datetime(length = lubridate::days(2)) %>%
  dplyr::mutate(MEDI.EMA = exponential_moving_average(MEDI, Datetime))

# Plot to compare results
sample.data.environment.EMA %>%
  ggplot2::ggplot(ggplot2::aes(x = Datetime)) +
  ggplot2::geom_line(ggplot2::aes(y = MEDI), colour = "black") +
  ggplot2::geom_line(ggplot2::aes(y = MEDI.EMA), colour = "red")
```

`extract_clusters` *Find and extract clusters from a dataset*

Description

`extract_clusters()` searches for and summarizes clusters where data meets a certain condition. Clusters have a specified duration and can be interrupted while still counting as one cluster. The variable can either be a column in the dataset or an expression that gets evaluated in a `dplyr::mutate()` call.

Cluster start and end times are shifted by half of the epoch each. E.g., a state lasting for 4 measurement points will have a duration of 4 measurement intervals, and a state only occuring once, of one interval. This deviates from simply using the time difference between the first and last occurance, which would be one epoch shorter (e.g., the start and end points for a state lasting a single point is identical, i.e., zero duration)

Groups will not be dropped, meaning that summaries based on the clusters will account for groups without clusters.

For correct cluster identification, there can be no gaps in the data! Gaps can inadvertently be introduced to a gapless dataset through grouping. E.g., when grouping by photoperiod (day/night) within a participant, this introduces gaps between the individual days and nights that together form the group. To avoid this, either group by individual days and nights (e.g., by using `number_states()` before grouping), which will make sure a cluster cannot extend beyond any grouping. Alternatively, you can set `handle.gaps = TRUE` (at computational cost).

`add_clusters()` identifies clusters and adds them back into the dataset through a rolling join. This is a convenience function built on `extract_clusters()`.

Usage

```
extract_clusters(
  data,
  Variable,
  Datetime.colname = Datetime,
  cluster.duration = "30 mins",
  duration.type = c("min", "max"),
  interruption.duration = 0,
  interruption.type = c("max", "min"),
  cluster.colname = state.count,
  return.only.clusters = TRUE,
  drop.empty.groups = TRUE,
  handle.gaps = FALSE,
  add.label = FALSE
)

add_clusters(
  data,
  Variable,
  Datetime.colname = Datetime,
  cluster.duration = "30 mins",
  duration.type = c("min", "max"),
  interruption.duration = 0,
  interruption.type = c("max", "min"),
  cluster.colname = state,
  handle.gaps = FALSE
)
```

Arguments

`data`	A light logger dataset. Expects a dataframe.
`Variable`	The variable or condition to be evaluated for clustering. Can be a column name or an expression.

`Datetime.colname`
: Column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol.

`cluster.duration`
: The minimum or maximum duration of a cluster. Defaults to 30 minutes. Expects a lubridate duration object (or a numeric in seconds).

`duration.type`
: Type of the duration requirement for clusters. Either "min" (minimum duration) or "max" (maximum duration). Defaults to "min".

`interruption.duration`
: The duration of allowed interruptions within a cluster. Defaults to 0 (no interruptions allowed).

`interruption.type`
: Type of the interruption duration. Either "max" (maximum interruption) or "min" (minimum interruption). Defaults to "max".

`cluster.colname`
: Name of the column to use for the cluster identification. Defaults to "state.count". Expects a symbol.

`return.only.clusters`
: Whether to return only the identified clusters (TRUE) or also include non-clusters (FALSE). Defaults to TRUE.

`drop.empty.groups`
: Logical. Should empty groups be dropped? Only works if `.drop = FALSE` has not been used with the current grouping prior to calling the function. Default to `TRUE`. If set to `FALSE` can lead to an error if factors are present in the grouping that have more levels than actual data. Can, however, be useful and necessary when summarizing the groups further, e.g. through `summarize_numeric()` - having an empty group present is important when averaging numbers.

`handle.gaps`
: Logical whether the data shall be treated with `gap_handler()`. Is set to `FALSE` by default, due to computational costs.

`add.label`
: Logical. Option to add a label to the output containing the condition. E.g., `MEDI>500|d>=30min|i<=5min` for clusters of melanopic EDI larger than 500, at least 30 minutes long (`d`), allowing interruptions of up to 5 minutes at a time (`i`).

Value

For `extract_clusters()` a dataframe containing the identified clusters or all time periods, depending on `return.only.clusters`.

For `add_clusters()` a dataframe containing the original data with an additional column for cluster identification.

Examples

```
dataset <-
sample.data.environment |>
dplyr::filter(Id == "Participant") |>
filter_Date(length = "1 day")
```

```
# Extract clusters with minimum duration of 1 hour and interruptions of up to 5 minutes
dataset |>
 extract_clusters(
  MEDI > 250,
  cluster.duration = "1 hour",
  interruption.duration = "5 mins"
)

# Add clusters to a dataset where lux values are above 20 for at least 30 minutes
dataset_with_clusters <-
dataset %>% add_clusters(MEDI > 20)

#peak into the dataset
dataset_with_clusters[4500:4505,]
```

---

extract_gaps *Extract gap episodes from the data*

Description

Finds and extracts gap episodes from a dataset. If no variable is provided, it will look for implicit gaps (gaps in the regular interval), if a variable is provided, it will look for implicit and explicit gaps (NA in the variable)

Usage

```
extract_gaps(
  dataset,
  Variable.colname = NULL,
  Datetime.colname = Datetime,
  epoch = "dominant.epoch",
  full.days = TRUE,
  include.implicit.gaps = TRUE
)
```

Arguments

`dataset` A light logger dataset. Needs to be a dataframe.

`Variable.colname`
Column name of the variable to check for NA values. Expects a symbol or NULL (only implicit gaps).

`Datetime.colname`
The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

epoch	The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.
full.days	If `TRUE`, the gapless sequence will include the whole first and last day where there is data.
include.implicit.gaps	Logical. Whether to expand the datetime sequence and search for implicit gaps, or not. Default is `TRUE`. If no `Variable.colname` is provided, this argument will be ignored. **If there are implicit gaps, gap calculation can be incorrect whenever there are missing explicit gaps flanking implicit gaps!**

Value

A dataframe containing gap times per grouping variable

See Also

Other regularize: `dominant_epoch()`, `gap_finder()`, `gap_handler()`, `gapless_Datetimes()`, `has_gaps()`, `has_irregulars()`

Examples

```
#removing some data to create gaps
sample.data.environment |>
 dplyr::filter(MEDI <= 50000) |>
 extract_gaps() |> head()

#not searching for implicit gaps
sample.data.environment |>
  dplyr::filter(MEDI <= 50000) |>
 extract_gaps(MEDI, include.implicit.gaps = FALSE)

#making implicit gaps explicit changes the summary
sample.data.environment |>
  dplyr::filter(MEDI <= 50000) |>
  gap_handler()|>
  extract_gaps(MEDI, include.implicit.gaps = FALSE) |> head()
```

---

extract_metric   *Add metrics to extracted sSummary*

Description

This helper function adds metric values to an extract, like from `extract_states()` or `extract_clusters()`. E.g., the average value of a variable during a cluster or state instance might be of interest. The metrics must be specified by the user using the `...` argument.

Usage

```
extract_metric(
  extracted_data,
  data,
  identifying.colname = state.count,
  Datetime.colname = Datetime,
  ...
)
```

Arguments

`extracted_data` A dataframe containing cluster or state summaries, typically from `extract_clusters()` or `extract_states()`.

`data` The original dataset that produced `extracted_data`

`identifying.colname` Name of the column in `extracted_data` that uniquely identifies each row (in addition to the groups. Expects a symbol. Defaults to `state.count`

`Datetime.colname` Column name that contains the datetime in `data`. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol. This argument is only necessary if `data` does not contain the `cluster.colname`.

`...` Arguments specifying the metrics to add summary. For example: `"mean_lux"` = `mean(lux)`.

Details

The original `data` does not have to have the cluster/state information, but it will be computationally faster if it does.

Value

A dataframe containing the extracted data with added metrics.

Examples

```
# Extract clusters and add mean MEDI value
sample.data.environment |>
filter_Date(length = "2 days") |>
extract_clusters(MEDI > 1000) |>
extract_metric(
  sample.data.environment,
  "mean_medi" = mean(MEDI, na.rm = TRUE)
) |>
dplyr::select(Id, state.count, duration, mean_medi)

# Extract states and add mean MEDI value
dataset <-
sample.data.environment |>
filter_Date(length = "2 days") |>
```

```
 add_photoperiod(c(48.5, 9))

dataset |>
  extract_states(photoperiod.state) |>
  extract_metric(dataset, mean_lux = mean(MEDI)) |>
  dplyr::select(state.count, duration, mean_lux)
```

`extract_states` *Extract summaries of states*

Description

Extracts a state from a dataset and provides their start and end times, as well as duration and epoch. The state does not have to exist in the dataset, but can be dynamically created. Extracted states can have group-dropping disabled, meaning that summaries based on the extracted states show empty groups as well.

Usage

```
extract_states(
  data,
  State.colname,
  State.expression = NULL,
  Datetime.colname = Datetime,
  handle.gaps = FALSE,
  epoch = "dominant.epoch",
  drop.empty.groups = TRUE,
  group.by.state = TRUE
)
```

Arguments

`data` A light logger dataset. Expects a dataframe.

`State.colname` The variable or condition to be evaluated for state exctration. Expects a symbol. If it is not part of the data, a `State.expression` is required.

`State.expression` If `State.colname` is not part of the `data`, this expression will be evaluated to generate the state. The result of this expression will be used for grouping, so it is recommended to be factor-like. If `State.colname` **is** part of the `data`, this argument will be ignored

`Datetime.colname` Column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol.

`handle.gaps` Logical whether the data shall be treated with `gap_handler()`. Is set to `FALSE` by default, due to computational costs.

epoch
: The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.

drop.empty.groups
: Logical. Should empty groups be dropped? Only works if `.drop = FALSE` has not been used with the current grouping prior to calling the function. Default to `TRUE`. If set to `FALSE` can lead to an error if factors are present in the grouping that have more levels than actual data. Can, however, be useful and necessary when summarizing the groups further, e.g. through `summarize_numeric()` - having an empty group present is important when averaging numbers.

group.by.state
: Logical. Should the output be automatically be grouped by the new state?

Value

a dataframe with one row per state instance. Each row will consist of the original dataset grouping, the state column. A state.count column, start and end Datetimes, as well as a duration of the state

Examples

```
#summarizing states "photoperiod"
states <-
sample.data.environment |>
  add_photoperiod(c(48.52, 9.06)) |>
  extract_states(photoperiod.state)
states |> head(2)
states |> tail(2)
states |> summarize_numeric(c("state.count", "epoch"))
```

`filter_Datetime` *Filter Datetimes in a dataset.*

Description

Filtering a dataset based on Dates or Datetimes may often be necessary prior to calcuation or visualization. The functions allow for a filtering based on simple `strings` or `Datetime` scalars, or by specifying a length. They also support prior **dplyr** grouping, which is useful, e.g., when you only want to filter the first two days of measurement data for every participant, regardless of the actual date. If you want to filter based on times of the day, look to `filter_Time()`.

Usage

```
filter_Datetime(
  dataset,
  Datetime.colname = Datetime,
  start = NULL,
  end = NULL,
  length = NULL,
```

```
  length_from_start = TRUE,
  full.day = FALSE,
  tz = NULL,
  only_Id = NULL,
  filter.expr = NULL
)

filter_Date(..., start = NULL, end = NULL)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`start, end` For `filter_Datetime()` a `POSIXct` or `character` scalar in the form of `"yyyy-mm-dd hh-mm-ss"` giving the respective start and end time positions for the filtered dataframe. If you only want to provide `dates` in the form of `"yyyy-mm-dd"`, use the wrapper function `filter_Date()`.

- If one or both of start/end are not provided, the times will be taken from the respective extreme values of the `dataset`.
- If `length` is provided and one of start/end is not, the other will be calculated based on the given value.
- If `length` is provided and both of start/end are NULL, the time from the respective start is taken.

`length` Either a Period or Duration from **lubridate**. E.g., `days(2) + hours(12)` will give a period of 2.5 days, whereas `ddays(2) + dhours(12)` will give a duration. For the difference between periods and durations look at the documentation from **lubridate**. Basically, periods model clocktimes, whereas durations model physical processes. This matters on several occasions, like leap years, or daylight savings. You can also provide a `character` scalar in the form of e.g. "1 day", which will be converted into a period.

`length_from_start` A `logical` indicating whether the `length` argument should be applied to the start (default, TRUE) or the end of the data (FALSE). Only relevant if neither the `start` nor the `end` arguments are provided.

`full.day` A `logical` indicating whether the `start` param should be rounded to a full day, when only the `length` argument is provided (Default is FALSE). This is useful, e.g., when the first observation in the dataset is slightly after midnight. If TRUE, it will count the length from midnight on to avoid empty days in plotting with `gg_day()`.

`tz` Timezone of the start/end times. If `NULL` (the default), it will take the timezone from the `Datetime.colname` column.

`only_Id` An expression of `ids` where the filtering should be applied to. If `NULL` (the default), the filtering will be applied to all `ids`. Based on the this expression,

	the dataset will be split in two and only where the given expression evaluates to `TRUE`, will the filtering take place. Afterwards both sets are recombined and sorted by `Datetime`.
`filter.expr`	Advanced filtering conditions. If not `NULL` (default) and given an `expression`, this is used to `dplyr::filter()` the results. This can be useful to filter, e.g. for group-specific conditions, like starting after the first two days of measurement (see examples).
`...`	Parameter handed over to `lubridate::round_date()` and siblings

Value

a `data.frame` object identical to `dataset` but with only the specified Dates/Times.

See Also

Other filter: `filter_Time()`

Other filter: `filter_Time()`

Examples

```
library(lubridate)
library(dplyr)
#baseline
range.unfiltered <- sample.data.environment$Datetime %>% range()
range.unfiltered

#setting the start of a dataset
sample.data.environment %>%
filter_Datetime(start = "2023-08-31 12:00:00") %>%
pull(Datetime) %>%
range()

#setting the end of a dataset
sample.data.environment %>%
filter_Datetime(end = "2023-08-31 12:00:00") %>% pull(Datetime) %>% range()

#setting a period of a dataset
sample.data.environment %>%
filter_Datetime(end = "2023-08-31 12:00:00", length = days(2)) %>%
pull(Datetime) %>% range()

#setting only the period of a dataset
sample.data.environment %>%
filter_Datetime(length = days(2)) %>%
pull(Datetime) %>% range()

#advanced filtering based on grouping (second day of each group)
sample.data.environment %>%
#shift the "Environment" group by one day
mutate(
Datetime = ifelse(Id == "Environment", Datetime + ddays(1), Datetime) %>%
```

```
as_datetime()) -> sample
sample %>% summarize(Daterange = paste(min(Datetime), max(Datetime), sep = " - "))
#now we can use the `filter.expr` argument to filter from the second day of each group
sample %>%
filter_Datetime(filter.expr = Datetime > Datetime[1] + days(1)) %>%
summarize(Daterange = paste(min(Datetime), max(Datetime), sep = " - "))

sample.data.environment %>% filter_Date(end = "2023-08-31")
```

---

filter_Datetime_multiple

Filter multiple times based on a list of arguments.

Description

filter_Datetime_multiple() is a wrapper around filter_Datetime() or filter_Date() that allows the cumulative filtering of Datetimes based on varying filter conditions. It is most useful in conjunction with the only_Id argument, e.g., to selectively cut off dates depending on participants (see examples)

Usage

```
filter_Datetime_multiple(
  dataset,
  arguments,
  filter_function = filter_Datetime,
  ...
)
```

Arguments

dataset	A light logger dataset
arguments	A list of arguments to be passed to filter_Datetime() or filter_Date(). each list entry must itself be a list of arguments, e.g, list(start = "2021-01-01", only_Id = quote(Id == 216)). Expressions have to be quoted with quote() or rlang::expr().
filter_function	The function to be used for filtering, either filter_Datetime (the default) or filter_Date
...	Additional arguments passed to the filter function. If the length argument is provided here instead of the argument, it has to be written as a string, e.g., length = "1 day", instead of length = lubridate::days(1).

Value

A dataframe with the filtered data

Examples

```
arguments <- list(
 list(start = "2023-08-31", only_Id = quote(Id == "Participant")),
 list(end = "2023-08-31", only_Id = quote(Id == "Environment")))
 #compare the unfiltered dataset
 sample.data.environment %>% gg_overview(Id.colname = Id)
 #compare the unfiltered dataset
 sample.data.environment %>%
 filter_Datetime_multiple(arguments = arguments, filter_Date) %>%
 gg_overview(Id.colname = Id)
```

`filter_Time` *Filter Times in a dataset.*

Description

Filter Times in a dataset.

Usage

```
filter_Time(
  dataset,
  Datetime.colname = Datetime,
  start = NULL,
  end = NULL,
  length = NULL
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`start, end, length` a `character` scalar in the form of `"hh-mm-ss"` giving the respective start, end, or length for the filtered dataframe. The input can also come from a `POSIXct` datetime, where only the time component will be used.

- If one or both of start/end are not provided, the times will be taken from the respective extreme values of the `dataset`.
- If `length` is provided and one of start/end is not, the other will be calculated based on the given value.
- If `length` is provided and both of start/end are not, the time from the respective start is taken.

Value

a `data.frame` object identical to `dataset` but with only the specified Times.

See Also

Other filter: `filter_Datetime()`

Examples

```
sample.data.environment %>%
filter_Time(start = "4:00:34", length = "12:00:00") %>%
dplyr::pull(Time) %>% range() %>% hms::as_hms()
```

---

frequency_crossing_threshold

Frequency of crossing light threshold

Description

This functions calculates the number of times a given threshold light level is crossed.

Usage

```
frequency_crossing_threshold(
  Light.vector,
  threshold,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`threshold`	Single numeric value specifying the threshold light level to compare with.
`na.rm`	Logical. Should missing light values be removed? Defaults to `FALSE`.
`as.df`	Logical. Should the output be returned as a data frame? If `TRUE`, a data frame with a single column named `frequency_crossing_{threshold}` will be returned. Defaults to `FALSE`.

Value

Data frame or matrix with pairs of threshold and calculated values.

References

Alvarez, A. A., & Wildsoet, C. F. (2013). Quantifying light exposure patterns in young adult students. *Journal of Modern Optics*, 60(14), 1200–1208. doi:10.1080/09500340.2013.845700

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
N = 60
dataset1 <-
  tibble::tibble(
    Id = rep("A", N),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(1:N),
    MEDI = sample(c(sample(1:249, N / 2), sample(250:1000, N / 2))),
  )

dataset1 %>%
 dplyr::reframe("Frequency crossing 250lx" = frequency_crossing_threshold(MEDI, threshold = 250))

dataset1 %>%
  dplyr::reframe(frequency_crossing_threshold(MEDI, threshold = 250, as.df = TRUE))
```

`gain.ratio.tables` *Gain / Gain-ratio tables to normalize counts*

Description

A list of tables containing gain and gain-ratios to normalize counts across different sensor gains.

Usage

```
gain.ratio.tables
```

Format

`gain.ratio.tables` A list containing two-column tibbles

TSL2585 gain table for the ambient light sensor TSL2585

Info A named `character` vector specifying the version and date a sensor was added

Details

Utility: Some sensors provide raw counts and gain levels as part of their output. In some cases it is desirable to compare counts between sensors, e.g., to gauge daylight outside by comparing UV counts to photopic counts (a high ratio of UV/Pho indicates outside daylight). Or to gauge daylight inside by comparing IR counts to photopic counts (a high ratio of IR/Pho with a low ratio of UV/Pho indicates daylight in the context of LED or fluorescent lighting)

---

gapless_Datetimes *Create a gapless sequence of Datetimes*

Description

Create a gapless sequence of Datetimes. The Datetimes are determined by the minimum and maximum Datetime in the dataset and an epoch. The epoch can either be guessed from the dataset or specified by the user.

Usage

```
gapless_Datetimes(
  dataset,
  Datetime.colname = Datetime,
  epoch = "dominant.epoch",
  full.days = FALSE
)
```

Arguments

`dataset` A light logger dataset. Needs to be a dataframe.

`Datetime.colname` The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

`epoch` The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.

`full.days` If `TRUE`, the gapless sequence will include the whole first and last day where there is data.

Value

A `tibble` with a gapless sequence of `Datetime` as specified by `epoch`.

See Also

Other regularize: `dominant_epoch()`, `extract_gaps()`, `gap_finder()`, `gap_handler()`, `has_gaps()`, `has_irregulars()`

Examples

```
dataset <-
tibble::tibble(Id = c("A", "A", "A", "B", "B", "B"),
              Datetime = lubridate::as_datetime(1) +
              lubridate::days(c(0:2, 4, 6, 8))) %>%
              dplyr::group_by(Id)

dataset %>% gapless_Datetimes()
dataset %>% dplyr::ungroup() %>%  gapless_Datetimes()
dataset %>% gapless_Datetimes(epoch = "1 day")
```

`gap_finder` *Check for and output gaps in a dataset*

Description

Quickly check for implicit missing `Datetime` data. Outputs a message with a short summary, and can optionally return the gaps as a `tibble`. Uses `gap_handler()` internally.

Usage

```
gap_finder(
  dataset,
  Datetime.colname = Datetime,
  epoch = "dominant.epoch",
  gap.data = FALSE,
  silent = FALSE,
  full.days = FALSE
)
```

Arguments

`dataset` A light logger dataset. Needs to be a dataframe.

`Datetime.colname` The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

`epoch` The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.

`gap.data` Logical. If `TRUE`, returns a `tibble` of the gaps in the dataset. Default is `FALSE`.

`silent` Logical. If `TRUE`, suppresses the message with the summary of the gaps in the dataset. Default is `FALSE`. Only used for unit tests.

`full.days` If `TRUE`, the gapless sequence will include the whole first and last day where there is data.

Details

The gap_finder() function is a wrapper around gap_handler() with the behavior argument set to "gaps". The main difference is that gap_finder() returns a message with a short summary of the gaps in the dataset, and that the tibble with the gaps contains a column gap.id that indicates the gap number, which is useful to determine, e.g., the consecutive number of gaps between measurement data.

Value

Prints message with a short summary of the gaps in the dataset. If gap.data = TRUE, returns a tibble of the gaps in the dataset.

See Also

Other regularize: dominant_epoch(), extract_gaps(), gap_handler(), gapless_Datetimes(), has_gaps(), has_irregulars()

Examples

```
dataset <-
tibble::tibble(Id = c("A", "A", "A", "B", "B", "B"),
              Datetime = lubridate::as_datetime(1) +
                         lubridate::days(c(0:2, 4, 6, 8)) +
                         lubridate::hours(c(0,12,rep(0,4)))) %>%
dplyr::group_by(Id)
dataset

#look for gaps assuming the epoch is the dominant epoch of each group
gap_finder(dataset)

#return the gaps as a tibble
gap_finder(dataset, gap.data = TRUE)

#assuming the epoch is 1 day, we have different gaps, and the datapoint at noon is now `irregular`
gap_finder(dataset, epoch = "1 day")
```

---

gap_handler *Fill implicit gaps in a light logger dataset*

Description

Datasets from light loggers often have implicit gaps. These gaps are implicit in the sense that consecutive timestamps (Datetimes) might not follow a regular epoch/interval. This function fills these implicit gaps by creating a gapless sequence of Datetimes and joining it to the dataset. The gapless sequence is determined by the minimum and maximum Datetime in the dataset (per group) and an epoch. The epoch can either be guessed from the dataset or specified by the user. A sequence of gapless Datetimes can be created with the gapless_Datetimes() function, whereas the dominant epoch in the data can be checked with the dominant_epoch() function. The behaviour argument specifies how the data is combined. By default, the data is joined with a full join, which means that all rows from the gapless sequence are kept, even if there is no matching row in the dataset.

Usage

```
gap_handler(
  dataset,
  Datetime.colname = Datetime,
  epoch = "dominant.epoch",
  behavior = c("full_sequence", "regulars", "irregulars", "gaps"),
  full.days = FALSE
)
```

Arguments

`dataset`	A light logger dataset. Needs to be a dataframe.
`Datetime.colname`	The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.
`epoch`	The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.
`behavior`	The behavior of the join of the `dataset` with the `gapless` sequence. Can be one of `"full_sequence"` (the default), `"regulars"`, `"irregulars"`, or `"gaps"`. See @return for details.
`full.days`	If `TRUE`, the gapless sequence will include the whole first and last day where there is data.

Value

A modified `tibble` similar to `dataset` but with handling of implicit gaps, depending on the `behavior` argument:

- `"full_sequence"` adds timestamps to the `dataset` that are missing based on a full sequence of `Datetimes` (i.e., the gapless sequence). The `dataset` is this equal (no gaps) or greater in the number of rows than the input. One column is added. `is.implicit` indicates whether the row was added (`TRUE`) or not (`FALSE`). This helps differentiating measurement values from values that might be imputed later on.
- `"regulars"` keeps only rows from the gapless sequence that have a matching row in the dataset. This can be interpreted as a row-reduced `dataset` with only regular timestamps according to the `epoch`. In case of no gaps this tibble has the same number of rows as the input.
- `"irregulars"` keeps only rows from the `dataset` that do not follow the regular sequence of `Datetimes` according to the `epoch`. In case of no gaps this tibble has 0 rows.
- `"gaps"` returns a `tibble` of all implicit gaps in the dataset. In case of no gaps this tibble has 0 rows.

See Also

Other regularize: `dominant_epoch()`, `extract_gaps()`, `gap_finder()`, `gapless_Datetimes()`, `has_gaps()`, `has_irregulars()`

Examples

```
dataset <-
tibble::tibble(Id = c("A", "A", "A", "B", "B", "B"),
              Datetime = lubridate::as_datetime(1) +
                         lubridate::days(c(0:2, 4, 6, 8)) +
                         lubridate::hours(c(0,12,rep(0,4)))) %>%
dplyr::group_by(Id)
dataset
#assuming the epoch is 1 day, we can add implicit data to our dataset
dataset %>% gap_handler(epoch = "1 day")

#we can also check whether there are irregular Datetimes in our dataset
dataset %>% gap_handler(epoch = "1 day", behavior = "irregulars")

#to get to the gaps, we can use the "gaps" behavior
dataset %>% gap_handler(epoch = "1 day", behavior = "gaps")

#finally, we can also get just the regular Datetimes
dataset %>% gap_handler(epoch = "1 day", behavior = "regulars")
```

---

gap_table *Tabular summary of data and gaps in all groups*

Description

gap_table() creates a gt::gt() with one row per group, summarizing key gap and gap-related information about the dataset. These include the available data, total duration, number of gaps, missing implicit and explicit data, and, optionally, irregular data.

Usage

```
gap_table(
  dataset,
  Variable.colname = MEDI,
  Variable.label = "melanopic EDI",
  title = "Summary of available and missing data",
  Datetime.colname = Datetime,
  epoch = "dominant.epoch",
  full.days = TRUE,
  include.implicit.gaps = TRUE,
  check.irregular = TRUE,
  get.df = FALSE
)
```

Arguments

dataset	A light logger dataset. Needs to be a dataframe.

`Variable.colname`
: Column name of the variable to check for NA values. Expects a symbol.

`Variable.label`
: Clear name of the variable. Expects a string

`title`
: Title string for the table

`Datetime.colname`
: The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.

`epoch`
: The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.

`full.days`
: If `TRUE`, the gapless sequence will include the whole first and last day where there is data.

`include.implicit.gaps`
: Logical. Whether to expand the datetime sequence and search for implicit gaps, or not. Default is `TRUE`. If no `Variable.colname` is provided, this argument will be ignored. **If there are implicit gaps, gap calculation can be incorrect whenever there are missing explicit gaps flanking implicit gaps!**

`check.irregular`
: Logical on whether to include irregular data in the summary, i.e. data points that do not fall on the regular sequence.

`get.df`
: Logical whether the dataframe should be returned instead of a `gt::gt()` table

Value

A gt table about data and gaps in the dataset

Examples

```
sample.data.environment |> dplyr::filter(MEDI <= 50000) |> gap_table()
```

`gg_day` *Create a simple Time-of-Day plot of light logger data, faceted by Date*

Description

`gg_day()` will create a simple ggplot for every data in a dataset. The result can further be manipulated like any ggplot. This will be sensible to refine styling or guides.

Usage

```
gg_day(
  dataset,
  start.date = NULL,
  end.date = NULL,
  x.axis = Datetime,
```

```
  y.axis = MEDI,
  aes_col = NULL,
  aes_fill = NULL,
  group = Id,
  geom = "point",
  scales = c("fixed", "free_x", "free_y", "free"),
  x.axis.breaks = hms::hms(hours = seq(0, 24, by = 3)),
  y.axis.breaks = c(-10^(5:0), 0, 10^(0:5)),
  y.scale = "symlog",
  y.scale.sc = FALSE,
  x.axis.label = "Time of Day",
  y.axis.label = "Illuminance (lx, MEDI)",
  format.day = "%d/%m",
  title = NULL,
  subtitle = NULL,
  interactive = FALSE,
  facetting = TRUE,
  jco_color = TRUE,
  ...
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `x.axis.`.

`start.date, end.date` Choose an optional start or end date within your `dataset`. Expects a `date`, which can also be a `character` that is interpretable as a date, e.g., `"2023-06-03"`. If you need a Datetime or want to cut specific times of each day, use the `filter_Datetime()` function. Defaults to `NULL`, which means that the plot starts/ends with the earliest/latest date within the `dataset`.

`x.axis, y.axis` column name that contains the datetime (x, defaults to `"Datetime"` which is automatically correct for data imported with LightLogR) and the dependent variable (y, defaults to `"MEDI"`, or melanopic EDI, which is a standard measure of stimulus strength for the nonvisual effects of light). Expects a `symbol`. Needs to be part of the `dataset`.

`aes_col, aes_fill` optional arguments that define separate sets and colors or fills them. Expects anything that works with the layer data `ggplot2::aes()`. The default color palette can be overwritten outside the function (see examples).

`group` Optional column name that defines separate sets. Useful for certain geoms like `boxplot`.Expects anything that works with the layer data `ggplot2::aes()`

`geom` What geom should be used for visualization? Expects a `character`

- `"point"` for `ggplot2::geom_point()`
- `"line"` for `ggplot2::geom_line()`
- `"ribbon"` for `ggplot2::geom_ribbon()`

- as the value is just input into the geom_ function from **ggplot2**, other variants work as well, but are not extensively tested.

scales
: For ggplot2::facet_wrap(), should scales be "fixed", "free" or free in one dimension ("free_y" is the default). Expects a character.

x.axis.breaks, y.axis.breaks
: Where should breaks occur on the x and y.axis? Expects a numeric vector with all the breaks. If you want to activate the default behaviour of **ggplot2**, you need to put in ggplot2::waiver().

y.scale
: How should the y-axis be scaled?
  - Defaults to "symlog", which is a logarithmic scale that can also handle negative values.
  - "log10" would be a straight logarithmic scale, but cannot handle negative values.
  - "identity" does nothing (continuous scaling).
  - a transforming function, such as symlog_trans() or scales::identity_trans(), which allow for more control.

y.scale.sc
: logical for whether scientific notation shall be used. Defaults to FALSE.

x.axis.label, y.axis.label
: labels for the x- and y-axis. Expects a character.

format.day
: Label for each day. Default is %d/%m, which shows the day and month. Expects a character. For an overview of sensible options look at base::strptime()

title
: Plot title. Expects a character.

subtitle
: Plot subtitle. Expects a character.

interactive
: Should the plot be interactive? Expects a logical. Defaults to FALSE.

facetting
: Should an automated facet by day be applied? Default is TRUE and uses the Day.data variable that the function also creates if not present.

jco_color
: Should the ggsci::scale_color_jco() color palette be used? Defaults to TRUE.

...
: Other options that get passed to the main geom function. Can be used to adjust to adjust size, linewidth, or linetype.

Details

Besides plotting, the function creates two new variables from the given Datetime:

- Day.data is a factor that is used for facetting with ggplot2::facet_wrap(). Make sure to use this variable, if you change the faceting manually. Also, the function checks, whether this variable already exists. If it does, it will only convert it to a factor and do the faceting on that variable.
- Time is an hms created with hms::as_hms() that is used for the x.axis

The default scaling of the y-axis is a symlog scale, which is a logarithmic scale that only starts scaling after a given threshold (default = 0). This enables values of 0 in the plot, which are common in light logger data, and even enables negative values, which might be sensible for non-light data.

See symlog_trans() for details on tweaking this scale. The scale can also be changed to a normal or logarithmic scale - see the y.scale argument for more.

The default scaling of the color and fill scales is discrete, with the ggsci::scale_color_jco() and ggsci::scale_fill_jco() scales. To use a continuous scale, use the jco_color = FALSE setting. Both fill and color aesthetics are set to NULL by default. For most geoms, this is not important, but geoms that automatically use those aesthetics (like geom_bin2d, where fill = stat(count)) are affected by this. Manually adding the required aesthetic (like aes_fill = ggplot2::stat(count) will fix this).

Value

A ggplot object

Examples

```
#use `col`for separation of different sets
plot <- gg_day(
sample.data.environment,
scales = "fixed",
end.date = "2023-08-31",
y.axis.label = "mEDI (lx)",
aes_col = Id)
plot

#you can easily overwrite the color scale afterwards
plot + ggplot2::scale_color_discrete()

#or change the facetting
plot + ggplot2::facet_wrap(~Day.data + Id)
```

---

gg_days — *Create a simple datetime plot of light logger data, faceted by group*

Description

gg_days() will create a simple ggplot along the timeline. The result can further be manipulated like any ggplot. This will be sensible to refine styling or guides. Through the x.axis.limits arguments, the plot can be much refined to align several groups of differing datetime ranges. It uses the Datetime_limits() function to calculate the limits of the x-axis. Another notable functions that are used are Datetime_breaks() to calculate the breaks of the x-axis.

Usage

```
gg_days(
  dataset,
  x.axis = Datetime,
  y.axis = MEDI,
  aes_col = NULL,
```

```
    aes_fill = NULL,
    group = NULL,
    geom = "line",
    scales = c("free_x", "free_y", "fixed", "free"),
    x.axis.breaks = Datetime_breaks,
    y.axis.breaks = c(-10^(5:0), 0, 10^(0:5)),
    y.scale = "symlog",
    y.scale.sc = FALSE,
    x.axis.label = "Local Date/Time",
    y.axis.label = "Illuminance (lx, MEDI)",
    x.axis.limits = Datetime_limits,
    x.axis.format = "%a %D",
    title = NULL,
    subtitle = NULL,
    interactive = FALSE,
    facetting = TRUE,
    jco_color = TRUE,
    ...
)
```

Arguments

dataset
: A light logger dataset. Expects a dataframe. If not imported by LightLogR, take care to choose a sensible variable for the x.axis..

x.axis, y.axis
: column name that contains the datetime (x, defaults to "Datetime" which is automatically correct for data imported with LightLogR) and the dependent variable (y, defaults to "MEDI", or melanopic EDI, which is a standard measure of stimulus strength for the nonvisual effects of light). Expects a symbol. Needs to be part of the dataset.

aes_col, aes_fill
: optional input that defines separate sets and colors or fills them. Expects anything that works with the layer data ggplot2::aes().

group
: Optional column name that defines separate sets. Useful for certain geoms like boxplot.Expects anything that works with the layer data ggplot2::aes()

geom
: What geom should be used for visualization? Expects a character

  - "point" for ggplot2::geom_point()
  - "line" for ggplot2::geom_line()
  - "ribbon" for ggplot2::geom_ribbon()
  - as the value is just input into the geom_ function from **ggplot2**, other variants work as well, but are not extensively tested.

scales
: For ggplot2::facet_wrap(), should scales be "fixed", "free" or "free" in one dimension ("free_x" is the default). Expects a character.

x.axis.breaks
: The (major) breaks of the x-axis. Defaults to Datetime_breaks(). The function has several options for adjustment. The default setting place a major break every 12 hours, starting at 12:00 of the first day.

y.axis.breaks
: Where should breaks occur on the y.axis? Expects a `numeric vector` with all the breaks or a function that calculates them based on the limits. If you want to activate the default behaviour of **ggplot2**, you need to put in `ggplot2::waiver()`.

y.scale
: How should the y-axis be scaled?
  - Defaults to `"symlog"`, which is a logarithmic scale that can also handle negative values.
  - `"log10"` would be a straight logarithmic scale, but cannot handle negative values.
  - `"identity"` does nothing (continuous scaling).
  - a transforming function, such as `symlog_trans()` or `scales::identity_trans()`, which allow for more control.

y.scale.sc
: `logical` for whether scientific notation shall be used. Defaults to `FALSE`.

x.axis.label, y.axis.label
: labels for the x- and y-axis. Expects a `character`.

x.axis.limits
: The limits of the x-axis. Defaults to `Datetime_limits()`. Can and should be adjusted to shift the x-axis to align different groups of data.

x.axis.format
: The format of the x-axis labels. Defaults to `"%a %D"`, which is the weekday and date. See `base::strptime()` for more options.

title
: Plot title. Expects a `character`.

subtitle
: Plot subtitle. Expects a `character`.

interactive
: Should the plot be interactive? Expects a `logical`. Defaults to `FALSE`.

facetting
: Should an automated facet by grouping be applied? Default is `TRUE`.

jco_color
: Should the `ggsci::scale_color_jco()` color palette be used? Defaults to `TRUE`.

...
: Other options that get passed to the main geom function. Can be used to adjust to adjust size, linewidth, or linetype.

Details

The default scaling of the y-axis is a `symlog` scale, which is a logarithmic scale that only starts scaling after a given threshold (default = 0). This enables values of 0 in the plot, which are common in light logger data, and even enables negative values, which might be sensible for non-light data. See `symlog_trans()` for details on tweaking this scale. The scale can also be changed to a normal or logarithmic scale - see the y.scale argument for more.

Value

A ggplot object

Examples

```
dataset <-
sample.data.environment %>%
aggregate_Datetime(unit = "5 mins")
```

```
dataset %>% gg_days()
#restrict the x-axis to 3 days
dataset %>%
gg_days(
x.axis.limits = \(x) Datetime_limits(x, length = lubridate::ddays(3))
)
```

`gg_doubleplot` *Double Plots*

Description

The function is by default opinionated, and will automatically select the best way to display the double date plot. However, the user can also manually select the type of double date plot to be displayed: repeating each day (default when there is only one day in all of the groups), or displaying consecutive days (default when there are multiple days in the groups).

Usage

```
gg_doubleplot(
  dataset,
  Datetime.colname = Datetime,
  type = c("auto", "repeat", "next"),
  geom = "ribbon",
  alpha = 0.5,
  col = "grey40",
  fill = "#EFC000FF",
  linewidth = 0.4,
  x.axis.breaks.next = Datetime_breaks,
  x.axis.format.next = "%a %D",
  x.axis.breaks.repeat = ~Datetime_breaks(.x, by = "6 hours", shift =
    lubridate::duration(0, "hours")),
  x.axis.format.repeat = "%H:%M",
  ...
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`Datetime.colname` column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`type` One of "auto", "repeat", or "next". The default is "auto", which will automatically select the best way to display the double date plot based on the amount of days in the dataset (`all = 1 >> "repeat", else "next"`). "repeat" will repeat each day in the plot, and "next" will display consecutive days.

geom
: The type of geom to be used in the plot. The default is "ribbon".

alpha, linewidth
: The alpha and linewidth setting of the geom. The default is 0.5 and 0.4, respectively.

col, fill
: The color and fill of the geom. The default is "#EFC000FF". If the parameters aes_col or aes_fill are used through ..., these will override the respective col and fill parameters.

x.axis.breaks.next, x.axis.breaks.repeat
: Datetime breaks when consecutive days are displayed (type = "next") or days are repeated (type = "repeat"). Must be a function. The default for next is a label at 12:00 am of each day, and for repeat is a label every 6 hours.

x.axis.format.next, x.axis.format.repeat
: Datetime label format when consecutive days are displayed (type = "next") or days are repeated (type = "repeat"). The default for next is "%a %D", showing the date, and for repeat ist "%H:%M", showing hours and minutes. See base::strptime() for more options.

...
: Arguments passed to gg_days(). When the arguments aes_col and aes_fill are used, they will invalidate the col and fill parameters.

Details

gg_doubleplot() is a wrapper function for gg_days(), combined with an internal function to duplicate and reorganize dates in a dataset for a *double plot* view. This means that the same day is displayed multiple times within the plot in order to reveal pattern across days.

Value

a ggplot object

Examples

```
#take only the Participant data from sample data, and three days
library(dplyr)
library(lubridate)
library(ggplot2)
sample.data <-
sample.data.environment %>%
dplyr::filter(Id == "Participant") %>%
filter_Date(length = ddays(3))

#create a double plot with the default settings
sample.data %>% gg_doubleplot()

#repeat the same day in the plot
sample.data %>% gg_doubleplot(type = "repeat")

#more examples that are not executed for computation time:

#use the function with more than one Id
```

```
sample.data.environment %>%
filter_Date(length = ddays(3)) %>%
gg_doubleplot(aes_fill = Id, aes_col = Id) +
facet_wrap(~ Date.data, ncol = 1, scales = "free_x", strip.position = "left")

#if data is already grouped by days, type = "repeat" will be automatic
sample.data.environment %>%
dplyr::group_by(Date = date(Datetime), .add = TRUE) %>%
filter_Date(length = ddays(3)) %>%
gg_doubleplot(aes_fill = Id, aes_col = Id) +
guides(fill = "none", col = "none") + #remove the legend
facet_wrap(~ Date.data, ncol = 1, scales = "free_x", strip.position = "left")

#combining `aggregate_Date()` with `gg_doubleplot()` easily creates a good
#overview of the data
sample.data.environment %>%
aggregate_Date() %>%
gg_doubleplot()
```

---

gg_gaps *Visualize gaps and irregular data*

Description

`gg_gaps()` is built upon `gg_days()`, `gap_finder()`, and `gg_state()` to visualize where gaps and irregular data in a dataset are. The function does not differentiate between `implicit gaps`, which are missing timestamps of the regular interval, `explicit gaps`, which are `NA` values. Optionally, the function shows `irregular data`, which are datapoints that fall outside the regular interval.

Usage

```
gg_gaps(
  dataset,
  Variable.colname = MEDI,
  Datetime.colname = Datetime,
  fill.gaps = "red",
  col.irregular = "red",
  alpha = 0.5,
  on.top = FALSE,
  epoch = "dominant.epoch",
  full.days = TRUE,
  show.irregulars = FALSE,
  group.by.days = FALSE,
  include.implicit.gaps = TRUE,
  ...
)
```

Arguments

- `dataset` — A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `x.axis.`.
- `Variable.colname` — Variable that becomes the basis for gap analysis. expects a symbol
- `Datetime.colname` — The column that contains the datetime. Needs to be a `POSIXct` and part of the dataset.
- `fill.gaps` — Fill color for the gaps
- `col.irregular` — Dot color for irregular data
- `alpha` — A numerical value between 0 and 1 representing the transparency of the gaps Default is 0.5.
- `on.top` — Logical scalar. If `TRUE`, the states will be plotted on top of the existing plot. If `FALSE`, the states will be plotted underneath the
- `epoch` — The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.
- `full.days` — Logical. Whether full days are expected, even for the first and last measurement
- `show.irregulars` — Logical. Show irregular data points. Default is `FALSE`.
- `group.by.days` — Logical. Whether data should be grouped by days. This can make sense if only very few days from large groups are affected
- `include.implicit.gaps` — Logical. Whether the time series should be expanded only the current observations taken.
- `...` — Additional arguments given to `gg_days()`. Can be used to change the color or other aesthetic properties.

Value

a `ggplot` object with all gaps and optionally irregular data. Groups that do not have any gaps nor irregular data will be removed for clarity. Null if no groups remain

Examples

```
#calling gg_gaps on a healthy dataset is pointless
sample.data.environment |> gg_gaps()

#creating a gapped and irregular dataset
bad_dataset <-
sample.data.environment |>
  aggregate_Datetime(unit = "5 mins") |>
  dplyr::filter(Id == "Participant") |>
  filter_Date(length = "2 days") |>
  dplyr::mutate(
```

```
    Datetime = dplyr::if_else(
      lubridate::date(Datetime) == max(lubridate::date(Datetime)),
            Datetime, Datetime + 1
      )
    ) |>
dplyr::filter(MEDI <250)
bad_dataset |> has_gaps()
bad_dataset |> has_irregulars()

#by default, gg_gaps() only shows gaps
bad_dataset |> gg_gaps()

#it can also show irregular data
bad_dataset |> gg_gaps(show.irregulars = TRUE)
```

---

gg_heatmap *Plot a heatmap across days and times of day*

Description

This function plots a heatmap of binned values across the day over all days in a group. It also allows doubleplot functionality. **gg_heatmap() does not work with the additive functions gg_photoperiod() and gg_state().

Usage

```
gg_heatmap(
  dataset,
  Variable.colname = MEDI,
  Datetime.colname = Datetime,
  unit = "1 hour",
  doubleplot = c("no", "same", "next"),
  date.title = "Date",
  date.breaks = 1,
  date.labels = "%d/%m",
  time.title = "Local time (HH:MM)",
  time.breaks = hms::hms(hours = seq(0, 48, by = 6)),
  time.labels = "%H:%M",
  fill.title = "Illuminance\n(lx, mel EDI)",
  fill.scale = "symlog",
  fill.labels = function(x) format(x, scientific = FALSE, big.mark = " "),
  fill.breaks = c(-10^(5:0), 0, 10^(0:5)),
  fill.limits = c(0, 10^5),
  fill.remove = FALSE,
  ...
)
```

Arguments

`dataset`	A light dataset
`Variable.colname`	The column name of the variable to display. Defaults to `MEDI`. Expects a symbol.
`Datetime.colname`	The column name of the datetime column. Defaults to `Datetime`. Expects a symbol.
`unit`	level of aggregation for `Variable.colname`. Defaults to `"1 hour"`. Expects a duration or duration-coercible value
`doubleplot`	Should the data be plotted as a doubleplot. Default is "no". "next" will plot the respective next day after the first, "same" will plot the same day twice.
`date.title`	Title text of the y-axis. Defaults to `Date`
`date.breaks`	Spacing of date breaks. Defaults to `1` (every day)
`date.labels`	Formatting code of the date labels
`time.title`	Title text of the x-axis. Defaults to `Local time (HH:MM)`
`time.breaks`	Spacing of time breaks. Defauls to every six hours.
`time.labels`	Formatting code of the time labels
`fill.title`	Title text of the value (fill) scale.
`fill.scale`	Scaling of the value (fill) scale. Defaults to `"symlog"` (see `symlog_trans()`)
`fill.labels`	Formula to format the label values.
`fill.breaks`	Breaks in the fill scale
`fill.limits`	Limits of the fill scale. A length-2 numeric with the lower and upper scale. If one is replaced with `NA`, this limit will be based on the data.
`fill.remove`	Logical. Should the fill scale be removed? Handy when the fill scale is to be replaced by another scale without the console messages warning about existing scale
`...`	Other arguments to provide to the underlying `ggplot2::geom_raster()`

Details

The function uses `ggplot2::scale_fill_viridis_c()` for the fill scale. The scale can be substituted by any other scale via the standard `+` command of ggplot2. It is recommended to set `fill.remove = TRUE` to reduce warnings.

Value

A ggplot object

Examples

```
sample.data.environment |> gg_heatmap()

#heatmap with doubleplot
sample.data.environment |> gg_heatmap(doubleplot = "next")
```

```
#change the unit of aggregation
sample.data.environment |> gg_heatmap(unit = "5 mins")

#change the limits of the fill scale
sample.data.environment |> gg_heatmap(fill.limits = c(0, 10^4))
```

`gg_overview` *Plot an overview of dataset intervals with implicit missing data*

Description

Plot an overview of dataset intervals with implicit missing data

Usage

```
gg_overview(
  dataset,
  Datetime.colname = Datetime,
  Id.colname = Id,
  gap.data = NULL,
  ...,
  interactive = FALSE
)
```

Arguments

`dataset`	A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `x.axis`..
`Datetime.colname`	column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`.
`Id.colname`	The column name of the Id column (default is `Id`), needs to be in the `dataset`. This is also used as the y-axis variable and is the minimum grouping variable.
`gap.data`	Optionally provide a `tibble` with `start` and `end` `Datetimes` of gaps per group. If not provided, the function uses `gap_finder()` to calculate implicit missing data. This might be computationally intensive for large datasets and many missing data. In these cases it can make sense to calculate those gaps beforehand and provide them to the function. If an empty `tibble` (`tibble::tibble()`) is provided, the function will just plot the start and end dates of the dataset, which is computationally very fast at the cost of additional info.
`...`	Additional arguments given to the main `ggplot2::aes()` used for styling depending on data within the `dataset`
`interactive`	Should the plot be interactive? Expects a `logical`. Defaults to `FALSE`.

Value

A ggplot object

Examples

```
sample.data.environment %>% gg_overview()
```

---

gg_photoperiod *Add photoperiods to gg_day() or gg_days() plots*

Description

gg_photoperiod() is a helper function to add photoperiod information to plots generated with gg_day() or gg_days(). The function can either draw on the dawn and dusk columns of the dataset or use the coordinates and solarDep arguments to calculate the photoperiods. The time series must be based on a column called Datetime.

Usage

```
gg_photoperiod(
  ggplot_obj,
  coordinates = NULL,
  alpha = 0.2,
  solarDep = 6,
  on.top = FALSE,
  ...
)
```

Arguments

ggplot_obj	A ggplot object generated with gg_day() or gg_days() (or gg_doubleplot(). The dataset used to create these **must** have a Datetime column.
coordinates	A two element numeric vector representing the latitude and longitude of the location. If NULL, the default, the function will look for the dawn and dusk columns in the dataset. If those are not present, (and in the POSIXct format), the function will stop with an error. Further, if NULL, the solarDep argument will be ignored.
alpha	A numerical value between 0 and 1 representing the transparency of the photoperiods. Default is 0.2.
solarDep	A numerical value representing the solar depression angle between 90 and -90. This means a value of 6 equals **-6** degrees above the horizon. Default is 6, equalling Civil dawn/dusk. Other common values are 12 degrees for Nautical dawn/dusk, 18 degrees for Astronomical dawn/dusk, and 0 degrees for Sunrise/Sunset. Note that the output columns will always be named dawn and dusk, regardless of the solarDep value.

`on.top`	Logical scalar. If `TRUE`, the photoperiods will be plotted on top of the existing plot. If `FALSE`, the photoperiods will be plotted underneath the existing plot. Default is `FALSE`.
`...`	Additional arguments given to the `ggplot2::geom_rect()` used to construct the photoperiod shading. Can be used to change the fill color or other aesthetic properties.

Details

If used in combination with `gg_doubleplot()`, with that function in the `type = "repeat"` setting (either manually set, or because there is only one day of data per group present), photoperiods need to be added separately through `add_photoperiod()`, or the second photoperiod in each panel will be off by one day. See the examples for more information.

In general, if the photoperiod setup is more complex, it makes sense to add it prior to plotting and make sure the photoperiods are correct.

Value

a modified `ggplot` object with the photoperiods added.

See Also

Other photoperiod: `photoperiod()`

Examples

```
coordinates <- c(48.521637, 9.057645)
#adding photoperiods to a ggplot
sample.data.environment |>
  gg_days() |>
  gg_photoperiod(coordinates)

#adding photoperiods prior to plotting
sample.data.environment |>
  add_photoperiod(coordinates, solarDep = 0) |>
  gg_days() |>
  gg_photoperiod()

#more examples that are not executed for computation time:

#plotting photoperiods automatically works for both gg_day() and gg_days()
sample.data.environment |>
  gg_day() |>
  gg_photoperiod(coordinates)

#plotting for gg_doubleplot mostly works fine
sample.data.environment |>
  filter_Date(length = "2 days") |>
  gg_doubleplot() |>
  gg_photoperiod(coordinates)
```

```
#however, in cases where only one day of data per group is available, or the
#type = "repeat" setting is used, the photoperiods need to be added
#separately. Otherwise the second day will be off by one day in each panel.
#The visual difference is subtle, and might not be visible at all, as
#photoperiod only every changes by few minutes per day.

#WRONG
sample.data.environment |>
  filter_Date(length = "1 days") |>
  gg_doubleplot() |>
  gg_photoperiod(coordinates)

#CORRECT
sample.data.environment |>
  filter_Date(length = "1 days") |>
  add_photoperiod(coordinates) |>
  gg_doubleplot() |>
  gg_photoperiod()
```

---

gg_state *Add states to gg_day() or gg_days() plots*

Description

gg_state() is a helper function to add state information to plots generated with gg_day(), gg_days(), or gg_doubleplot(). The function can draw on any column in the dataset, but factor-like or logical columns make the most sense. The time series must be based on a column called Datetime.

Usage

```
gg_state(
  ggplot_obj,
  State.colname,
  aes_fill = NULL,
  aes_col = NULL,
  alpha = 0.2,
  on.top = FALSE,
  ignore.FALSE = TRUE,
  ...
)
```

Arguments

ggplot_obj	A ggplot object generated with gg_day() or gg_days() (or gg_doubleplot(). The dataset used to create these **must** have a Datetime column.
State.colname	The colname of the state to add to the plot. Must be part of the dataset. Expects a symbol.

aes_fill, aes_col
: conditional aesthetics for ggplot2::geom_rect(). The default (NULL) will be ignored, so that col and fill arguments can be set through the ... arguments. As the states work from a summarized dataset, only a few columns are available for filling/coloring: The State.colname, Grouping variables, and variables created by using extract_states().

alpha
: A numerical value between 0 and 1 representing the transparency of the states. Default is 0.2.

on.top
: Logical scalar. If TRUE, the states will be plotted on top of the existing plot. If FALSE, the states will be plotted underneath the existing plot. Default is FALSE.

ignore.FALSE
: Logical that drops FALSE values of a logical state column, so that only TRUE values are recognized as a state. Is only relevant for logical state columns and will be ignored otherwise. Default is TRUE.

...
: Additional arguments given to the ggplot2::geom_rect() used to construct the state shading. Can be used to change the fill color or other aesthetic properties.

Value

a modified ggplot object with the states added.

Examples

```
#creating a simple TRUE/FALSE state in the sample data: Light above 250 lx mel EDI
#and a second state that cuts data into chunks relating to the Brown et al. 2022 thresholds
#(+aggregating Data to 5 minute intervals & reducing it to three days)
state_data <-
  sample.data.environment |>
   dplyr::mutate(state = MEDI > 250) |>
   Brown_cut(MEDI, state2) |>
   aggregate_Datetime(unit = "5 mins") |>
   filter_Datetime(length = "3 days")

state_data |>
 gg_days() |>
 gg_state(state)

#state 2 has more than one valid state, thus we need to assign a fill aesthetic
state_data |>
 gg_days() |>
 gg_state(state2, aes_fill = state2) +
 ggplot2::scale_fill_manual(values=c("#868686FF", "#EFC000FF", "#0073C2FF"))
 #this line is simply for sensible colors

#same, but with gg_day()
state_data |>
 dplyr::filter(Id == "Participant") |>
 gg_day(geom = "line") |>
 gg_state(state, fill = "red")
```

```
#more complex state
state_data |>
dplyr::filter(Id == "Participant") |>
gg_day(geom = "line") |>
gg_state(state2, aes_fill = state2)

#with gg_doubleplot
state_data |>
dplyr::filter(Id == "Participant") |>
gg_doubleplot() |>
gg_state(state2, aes_fill = state2)
```

---

has_gaps *Does a dataset have implicit gaps*

Description

Returns TRUE if there are implicit gaps in the dataset and FALSE if it is gapless. Gaps can make sense depending on the grouping structure, but the general sequence of Datetimes within a dataset should always be gapless.

Usage

```
has_gaps(dataset, Datetime.colname = Datetime, epoch = "dominant.epoch")
```

Arguments

`dataset` A light logger dataset. Needs to be a dataframe.

`Datetime.colname` The column that contains the datetime. Needs to be a POSIXct and part of the dataset.

`epoch` The epoch to use for the gapless sequence. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid `lubridate::duration()` string, e.g., "1 day" or "10 sec".

Value

logical

See Also

Other regularize: `dominant_epoch()`, `extract_gaps()`, `gap_finder()`, `gap_handler()`, `gapless_Datetimes()`, `has_irregulars()`

Examples

```
#The sample dataset does not have gaps
sample.data.environment |> has_gaps()
#removing some of the data creates gaps
sample.data.environment |> dplyr::filter(MEDI <= 50000) |> has_gaps()

#having a grouped dataframe where the groups span multiple unconnected parts
#is considered a gap, which can be relevant, e.g., when searching for clusters
sample.data.environment |>
  add_photoperiod(c(47.1, 10)) |>
  dplyr::group_by(photoperiod.state) |>
  has_gaps()

 #to avoid this, use `number_states()` for grouping
 sample.data.environment |>
  add_photoperiod(c(48.52, 9.06)) |>
  number_states(photoperiod.state) |>
  dplyr::group_by(photoperiod.state.count, .add = TRUE) |>
  has_gaps()
```

---

has_irregulars *Does a dataset have irregular data*

Description

Returns TRUE if there are irregular data in the dataset and FALSE if not. Irregular data can make sense if two datasets within a single group are shifted to one another, e.g., if it contains data from two separate recording sessions. The second session will be unlikely to have started at the exact interval timing of the first session. While this is not problematic in itself, it is still recommended to rectify the Datetimes to a common timestamp if time precision permits it, e.g., through aggregate_Datetime() or cut_Datetime().

Usage

```
has_irregulars(dataset, Datetime.colname = Datetime, epoch = "dominant.epoch")
```

Arguments

dataset	A light logger dataset. Needs to be a dataframe.
Datetime.colname	The column that contains the datetime. Needs to be a POSIXct and part of the dataset.
epoch	The epoch to use for the gapless sequence. Can be either a lubridate::duration() or a string. If it is a string, it needs to be either '"dominant.epoch"' (the default) for a guess based on the data or a valid lubridate::duration() string, e.g., "1 day" or "10 sec".

Value

logical

See Also

Other regularize: dominant_epoch(), extract_gaps(), gap_finder(), gap_handler(), gapless_Datetimes(), has_gaps()

Examples

```
#the sample dataset does not have any irregular data
sample.data.environment |> has_irregulars()

#even removing some data does not make it irregular, as all the Datetimes
#still fall in the regular interval
sample.data.environment |> dplyr::filter(MEDI <= 50000) |> has_irregulars()

#shifting some of the data will create irregular data
sample.data.environment |>
  dplyr::mutate(
   Datetime = dplyr::if_else(
     sample(c(TRUE, FALSE), dplyr::n(), replace = TRUE), Datetime, Datetime + 1
     )
   ) |>
   has_irregulars()
```

---

import_adjustment *Adjust device imports or make your own*

Description

Adjust device imports or make your own

Usage

```
import_adjustment(import_expr)
```

Arguments

`import_expr`	A named list of import expressions. The basis for `LightLogR`'s import functions is the included dataset `ll_import_expr()`. If this function were to be given that exact dataset, and bound to a variable called `import`, it would be identical to the `import` function. See `details`.

Details

This function should only be used with some knowledge of how expressions work in R. The minimal required output for an expression to work as expected, it must lead to a data frame containing a `Datetime` column with the correct time zone. It has access to all arguments defined in the description of `import_Dataset()`. The `...` argument should be passed to whatever csv reader function is used, so that it works as expected. Look at `ll_import_expr()$ActLumus` for a quite minimal example.

Value

A list of import functions

Examples

```
#create a new import function for the ActLumus device, same as the old
new_import <- import_adjustment(ll_import_expr())
#the new one is identical to the old one in terms of the function body
identical(body(import$ActLumus), body(new_import$ActLumus))

#change the import expression for the LYS device to add a message at the top
new_import_expr <- ll_import_expr()
new_import_expr$ActLumus[[4]] <-
rlang::expr({ cat("**This is a new import function**\n")
data
})
new_import <- import_adjustment(new_import_expr)
filepath <-
system.file("extdata/205_actlumus_Log_1020_20230904101707532.txt.zip",
            package = "LightLogR")
#Now, a message is printed when the import function is called
data <- new_import$ActLumus(filepath, auto.plot = FALSE)
```

`import_Dataset` *Import a light logger dataset or related data*

Description

Imports a dataset and does the necessary transformations to get the right column formats. Unless specified otherwise, the function will set the timezone of the data to `UTC`. It will also enforce an `Id` to separate different datasets and will order/arrange the dataset within each `Id` by Datetime. See the Details and Devices section for more information and the full list of arguments.

Usage

```
import_Dataset(device, ...)

import
```

Arguments

device	From what device do you want to import? For a few devices, there is a sample data file that you can use to test the function (see the examples). See supported_devices() for a list of supported devices and see below for more information on devices with specific requirements.
...	Parameters that get handed down to the specific import functions

Format

An object of class list of length 19.

Details

There are specific and a general import function. The general import function is described below, whereas the specific import functions take the form of import$device(). The general import function is a thin wrapper around the specific import functions. The specific import functions take the following arguments:

- filename: Filename(s) for the Dataset. Can also contain the filepath, but path must then be NULL. Expects a character. If the vector is longer than 1, multiple files will be read in into one Tibble.
- path: Optional path for the dataset(s). NULL is the default. Expects a character.
- n_max: maximum number of lines to read. Default is Inf.
- tz: Timezone of the data. "UTC" is the default. Expects a character. You can look up the supported timezones with OlsonNames().
- Id.colname: Lets you specify a column for the id of a dataset. Expects a symbol (Default is Id). This column will be used for grouping (dplyr::group_by()).
- auto.id: If the Id.colname column is not part of the dataset, the Id can be automatically extracted from the filename. The argument expects a regular expression regex and will by default just give the whole filename without file extension.
- manual.id: If this argument is not NULL, and no Id column is part of the dataset, this character scalar will be used. **We discourage the use of this arguments when importing more than one file**
- silent: If set to TRUE, the function will not print a summary message of the import or plot the overview. Default is FALSE.
- locale: The locale controls defaults that vary from place to place.
- not.before: Remove data prior to this date. This argument is provided to start of filter_Date(). Data will be filtered out before any of the summaries are shown.
- dst_adjustment: If a file crosses daylight savings time, but the device does not adjust time stamps accordingly, you can set this argument to TRUE, to apply this shift manually. It is selective, so it will only be done in files that cross between DST and standard time. Default is FALSE. Uses dst_change_handler() to do the adjustment. Look there for more infos. It is not equipped to handle two jumps in one file (so back and forth between DST and standard time), but will work fine if jums occur in separate files.

- auto.plot: a logical on whether to call gg_overview() after import. Default is TRUE. But is set to FALSE if the argument silent is set to TRUE.
- ...: supply additional arguments to the **readr** import functions, like na. Might also be used to supply arguments to the specific import functions, like column_names for Actiwatch_Spectrum devices. Those devices will always throw a helpful error message if you forget to supply the necessary arguments. If the Id column is already part of the dataset it will just use this column. If the column is not present it will add this column and fill it with the filename of the importfile (see param auto.id).
- print_n can be used if you want to see more rows from the observation intervals
- remove_duplicates can be used if identical observations are present within or across multiple files. The default is FALSE. The function keeps only unique observations (=rows) if set to' TRUE'. This is a convenience implementation of dplyr::distinct().

Value

Tibble/Dataframe with a POSIXct column for the datetime

Devices

The set of import functions provide a convenient way to import light logger data that is then perfectly formatted to add metadata, make visualizations and analyses. There are a number of devices supported, where import should just work out of the box. To get an overview, you can simply call the supported_devices() dataset. The list will grow continuously as the package is maintained.

```
supported_devices()
#>  [1] "ActLumus"              "ActTrust"              "Actiwatch_Spectrum"
#>  [4] "Actiwatch_Spectrum_de" "Circadian_Eye"         "Clouclip"
#>  [7] "DeLux"                 "GENEActiv_GGIR"        "Kronowise"
#> [10] "LIMO"                  "LYS"                   "LiDo"
#> [13] "LightWatcher"          "MotionWatch8"          "OcuWEAR"
#> [16] "Speccy"                "SpectraWear"           "VEET"
#> [19] "nanoLambda"
```

ActLumus:

Manufacturer: Condor Instruments

Model: ActLumus

Implemented: Sep 2023

A sample file is provided with the package, it can be accessed through system.file("extdata/205_actlumus_Log_1020_ package = "LightLogR"). It does not need to be unzipped to be imported. This sample file is a good example for a regular dataset without gaps

LYS:

Manufacturer: LYS Technologies

Model: LYS Button

Implemented: Sep 2023

A sample file is provided with the package, it can be accessed through system.file("extdata/sample_data_LYS.csv", package = "LightLogR"). This sample file is a good example for an irregular dataset.

Actiwatch_Spectrum & Actiwatch_Spectrum_de:
Manufacturer: Philips Respironics
Model: Actiwatch Spectrum
Implemented: Nov 2023 / July 2024
Important note: The `Actiwatch_Spectrum` function is for an international/english formatting. The `Actiwatch_Spectrum_de` function is for a german formatting, which slightly differs in the datetime format, the column names, and the decimal separator.

ActTrust:
Manufacturer: Condor Instruments
Model: ActTrust1, ActTrust2
Implemented: Mar 2024
This function works for both ActTrust 1 and 2 devices

Speccy:
Manufacturer: Monash University
Model: Speccy
Implemented: Feb 2024

DeLux:
Manufacturer: Intelligent Automation Inc
Model: DeLux
Implemented: Dec 2023

LiDo:
Manufacturer: University of Lucerne
Model: LiDo
Implemented: Nov 2023

SpectraWear:
Manufacturer: University of Manchester
Model: SpectraWear
Implemented: May 2024

NanoLambda:
Manufacturer: NanoLambda
Model: XL-500 BLE
Implemented: May 2024

LightWatcher:
Manufacturer: Object-Tracker
Model: LightWatcher
Implemented: June 2024

VEET:

Manufacturer: Meta Reality Labs

Model: VEET

Implemented: July 2024

Required Argument: `modality` A character scalar describing the modality to be imported from. Can be one of `"ALS"` (Ambient light sensor), `"IMU"` (Inertial Measurement Unit), `"INF"` (Information), `"PHO"` (Spectral Sensor), `"TOF"` (Time of Flight)

Circadian_Eye:

Manufacturer: Max-Planck-Institute for Biological Cybernetics, Tübingen

Model: melanopiQ Circadian Eye (Prototype)

Implemented: July 2024

Kronowise:

Manufacturer: Kronohealth

Model: Kronowise

Implemented: July 2024

GENEActiv with GGIR preprocessing:

Manufacturer: Activeinsights

Model: GENEActiv

Note: This import function takes GENEActiv data that was preprocessed through the GGIR package. By default, `GGIR` aggregates light data into intervals of 15 minutes. This can be set by the `windowsizes` argument in GGIR, which is a three-value vector, where the second values is set to 900 seconds by default. To import the preprocessed data with `LightLogR`, the `filename` argument requires a path to the parent directory of the GGIR output folders, specifically the `meta` folder, which contains the light exposure data. Multiple `filenames` can be specified, each of which needs to be a path to a different GGIR parent directory. GGIR exports can contain data from multiple participants, these will always be imported fully by providing the parent directory. Use the `pattern` argument to extract sensible `Ids` from the *.RData* filenames within the *meta/basic/* folder. As per the author, Dr. Vincent van Hees, GGIR preprocessed data are always in local time, provided the `desiredtz`/`configtz` are properly set in GGIR. `LightLogR` still requires a timezone to be set, but will not timeshift the import data.

MotionWatch 8:

Manufacturer: CamNtech

Implemented: September 2024

LIMO:

Manufacturer: ENTPE

Implemented: September 2024

LIMO exports `LIGHT` data and `IMU` (inertia measurements, also UV) in separate files. Both can be read in with this function, but not at the same time. Please decide what type of data you need and provide the respective filenames.

OcuWEAR:

Manufacturer: Ocutune

Implemented: September 2024

OcuWEAR data contains spectral data. Due to the format of the data file, the spectrum is not directly part of the tibble, but rather a list column of tibbles within the imported data, containing a `Wavelength` (nm) and `Intensity` (mW/m^2) column.

Clouclip:

Manufacturer: Clouclip

Implemented: April 2025

Clouclip export files have the ending `.xls`, but are no real Microsoft Excel files, rather they are tab-separated text files. LightLogR thus does not read them in with an excel import routine. The measurement columns `Lux` and `Dis` contain sentinel values. `-1` (`Dis` and `Lux`) indicates sleep mode, whereas `204` (only `Dis`) indicates an out of range measurement. These values will be set to `NA`, and an additional column is added that translates these status codes. The columns carry the name `{.col}_status`.

Examples

Imports made easy:

To import a file, simple specify the filename (and path) and feed it to the `import_Dataset` function. There are sample datasets for all devices.

The import functions provide a basic overview of the data after import, such as the intervals between measurements or the start and end dates.

```
filepath <- system.file("extdata/205_actlumus_Log_1020_20230904101707532.txt.zip", package = "LightL
dataset <- import_Dataset("ActLumus", filepath, auto.plot = FALSE)
#>
#> Successfully read in 61'016 observations across 1 Ids from 1 ActLumus-file(s).
#> Timezone set is UTC.
#> The system timezone is Europe/Berlin. Please correct if necessary!
#>
#> First Observation: 2023-08-28 08:47:54
#> Last Observation: 2023-09-04 10:17:04
#> Timespan: 7.1 days
#>
#> Observation intervals:
#>   Id                                           interval.time     n pct
#> 1 205_actlumus_Log_1020_20230904101707532.txt 10s           61015 100%
```

Import functions can also be called directly:

```
dataset <- import$ActLumus(filepath, auto.plot = FALSE)
#>
#> Successfully read in 61'016 observations across 1 Ids from 1 ActLumus-file(s).
#> Timezone set is UTC.
#> The system timezone is Europe/Berlin. Please correct if necessary!
#>
#> First Observation: 2023-08-28 08:47:54
```

```
#> Last Observation: 2023-09-04 10:17:04
#> Timespan: 7.1 days
#>
#> Observation intervals:
#>   Id                                          interval.time     n pct
#> 1 205_actlumus_Log_1020_20230904101707532.txt 10s           61015 100%
```

```
dataset %>%
dplyr::select(Datetime, TEMPERATURE, LIGHT, MEDI, Id) %>%
dplyr::slice(1500:1505)
#> # A tibble: 6 x 5
#> # Groups:   Id [1]
#>   Datetime            TEMPERATURE LIGHT  MEDI Id
#>   <dttm>                    <dbl> <dbl> <dbl> <fct>
#> 1 2023-08-28 12:57:44        26.9  212.  202. 205_actlumus_Log_1020_20230904101~
#> 2 2023-08-28 12:57:54        26.9  208.  199. 205_actlumus_Log_1020_20230904101~
#> 3 2023-08-28 12:58:04        26.9  205.  196. 205_actlumus_Log_1020_20230904101~
#> 4 2023-08-28 12:58:14        26.8  204.  194. 205_actlumus_Log_1020_20230904101~
#> 5 2023-08-28 12:58:24        26.9  203.  194. 205_actlumus_Log_1020_20230904101~
#> 6 2023-08-28 12:58:34        26.8  204.  195. 205_actlumus_Log_1020_20230904101~
```

See Also

supported_devices

---

import_Statechanges *Import data that contain* Datetimes *of* Statechanges

---

Description

Auxiliary data greatly enhances data analysis. This function allows the import of files that contain Statechanges, i.e., specific time points of when a State (like sleep or wake) starts.

Usage

```
import_Statechanges(
  filename,
  path = NULL,
  sep = ",",
  dec = ".",
  structure = c("wide", "long"),
  Datetime.format = "ymdHMS",
  tz = "UTC",
  State.colnames,
  State.encoding = State.colnames,
  Datetime.column = Datetime,
  Id.colname,
```

```
  State.newname = State,
  Id.newname = Id,
  keep.all = FALSE,
  silent = FALSE
)
```

Arguments

filename
: Filename(s) for the Dataset. Can also contain the filepath, but `path` must then be `NULL`. Expects a `character`. If the vector is longer than `1`, multiple files will be read in into one Tibble.

path
: Optional path for the dataset(s). `NULL` is the default. Expects a `character`.

sep
: String that separates columns in the import file. Defaults to `","`.

dec
: String that indicates a decimal separator in the import file. Defaults to `"."`.

structure
: String that specifies whether the import file is in the `long` or `wide` format. Defaults to `"wide"`.

Datetime.format
: String that specifies the format of the `Datetimes` in the file. The default `"ymdHMS"` specifies a format like "2023-07-10 10:00:00". In the function, `lubridate::parse_date_time()` does the actual conversion - the documentation can be searched for valid inputs.

tz
: Timezone of the data. `"UTC"` is the default. Expects a `character`. You can look up the supported timezones with `OlsonNames()`.

State.colnames
: Column name or vector of column names (the latter only in the `wide` format). Expects a `character`.
  - In the `wide` format, the column names indicate the `State` and must contain `Datetimes`. The columns will be pivoted to the columns specified in `Datetime.column` and `State.newname`.
  - In the `long` format, the column contains the `State`

State.encoding
: In the `wide` format, this enables recoding the column names to state names, if there are any differences. The default uses the `State.colnames` argument. Expects a `character` (vector) with the same length as `State.colnames`.

Datetime.column
: Symbol of the `Datetime` column (which is also the default).
  - In the `wide` format, this is the newly created column from the `Datetimes` in the `State.colnames`.
  - In the `long` format, this is the existing column that contains the `Datetimes`.

Id.colname
: Symbol of the column that contains the `ID` of the subject.

State.newname
: Symbol of the column that will contain the `State` of the subject. In the `wide` format, this is the newly created column from the `State.colnames`. In the `long` format, this argument is used to rename the `State` column.

Id.newname
: Column name used for renaming the `Id.colname` column.

keep.all
: Logical that specifies whether all columns should be kept in the output. Defaults to `FALSE`.

silent
: Logical that specifies whether a summary of the imported data should be shown. Defaults to `FALSE`.

Details

Data can be present in the long or wide format.

- In the `wide` format, multiple `Datetime` columns indicate the state through the column name. These get pivoted to the `long` format and can be recoded through the `State.encoding` argument.
- In the `long` format, one column indicates the `State`, while the other gives the `Datetime`.

Value

a dataset with the `ID`, `State`, and `Datetime` columns. May contain additional columns if `keep.all` is `TRUE`.

Examples

```
#get the example file from within the package
path <- system.file("extdata/",
package = "LightLogR")
file.sleep <- "205_sleepdiary_all_20230904.csv"

#import Data in the wide format (sleep/wake times)
import_Statechanges(file.sleep, path,
Datetime.format = "dmyHM",
State.colnames = c("sleep", "offset"),
State.encoding = c("sleep", "wake"),
Id.colname = record_id,
sep = ";",
dec = ",")

#import in the long format (Comments on sleep)
import_Statechanges(file.sleep, path,
                   Datetime.format = "dmyHM",
                   State.colnames = "comments",
                   Datetime.column = sleep,
                   Id.colname = record_id,
                   sep = ";",
                   dec = ",", structure = "long")
```

`interdaily_stability` *Interdaily stability (IS)*

Description

This function calculates the variability of 24h light exposure patterns across multiple days. Calculated as the ratio of the variance of the average daily pattern to the total variance across all days. Calculated with mean hourly light levels. Ranges between 0 (Gaussian noise) and 1 (Perfect Stability).

Usage

```
interdaily_stability(
  Light.vector,
  Datetime.vector,
  use.samplevar = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Datetime.vector`	Vector containing the time data. Must be POSIXct.
`use.samplevar`	Logical. Should the sample variance be used (divide by N-1)? By default (`FALSE`), the population variance (divide by N) is used, as described in Van Someren et al. (1999).
`na.rm`	Logical. Should missing values be removed? Defaults to `FALSE`.
`as.df`	Logical. Should the output be returned as a data frame? If `TRUE`, a data frame with a single column named `interdaily_stability` will be returned. Defaults to `FALSE`.

Details

Note that this metric will always be 1 if the data contains only one 24 h day.

Value

Numeric value or dataframe with column 'IS'.

References

Van Someren, E. J. W., Swaab, D. F., Colenda, C. C., Cohen, W., McCall, W. V., & Rosenquist, P. B. (1999). Bright Light Therapy: Improved Sensitivity to Its Effects on Rest-Activity Rhythms in Alzheimer Patients by Application of Nonparametric Methods. *Chronobiology International*, 16(4), 505–518. doi:10.3109/07420529908998724

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
set.seed(1)
N <- 24 * 7
# Calculate metric for seven 24 h days with two measurements per hour
dataset1 <-
  tibble::tibble(
    Id = rep("A", N * 2),
   Datetime = lubridate::as_datetime(0) + c(lubridate::minutes(seq(0, N * 60 - 30, 30))),
    MEDI = sample(1:1000, N * 2)
  )
dataset1 %>%
  dplyr::summarise(
    "Interdaily stability" = interdaily_stability(MEDI, Datetime)
  )
```

`interval2state` *Adds a state column to a dataset from interval data*

Description

This function can make use of `Interval` data that contain `States` (like `"sleep"`, `"wake"`, `"wear"`) and add a column to a light logger `dataset`, where the `State` of every `Datetime` is specified, based on the participant's `Id`.

Usage

```
interval2state(
  dataset,
  State.interval.dataset,
  Datetime.colname = Datetime,
  State.colname = State,
  Interval.colname = Interval,
  Id.colname.dataset = Id,
  Id.colname.interval = Id,
  overwrite = FALSE,
  output.dataset = TRUE
)
```

Arguments

`dataset` A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.

`State.interval.dataset`
Name of the dataset that contains `State` and `Interval` columns. Interval data can be created, e.g., through `sc2interval()`.

`Datetime.colname`
: column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.

`State.colname`, `Interval.colname`
: Column names of the `State` and `Interval` in the `State.interval.dataset`. Expects a `symbol`. `State` can't be in the `dataset` yet or the function will give an error. You can also set `overwrite = TRUE`.

`Id.colname.dataset`, `Id.colname.interval`
: Column names of the participant's `Id` in both the `dataset` and the `State.interval.dataset`. On the off-chance that there are inconsistencies, the names can be different. If the datasets were imported and preprocessed with LightLogR, this just works. Both datasets need an `Id`, because the states will be added based not only on the `Datetime`, but also depending on the dataset.

`overwrite`
: If `TRUE` (defaults to `FALSE`), the function will overwrite the `State.colname` column if it already exists.

`output.dataset`
: should the output be a `data.frame` (Default `TRUE`) or a vector with `hms` (`FALSE`) times? Expects a `logical` scalar.

Value

One of

- a `data.frame` object identical to `dataset` but with the state column added
- a `vector` with the states

Examples

```
#create a interval dataset
library(tibble)
library(dplyr)
library(lubridate)
library(rlang)
library(purrr)
states <- tibble::tibble(Datetime = c("2023-08-15 6:00:00",
                                      "2023-08-15 23:00:00",
                                      "2023-08-16 6:00:00",
                                      "2023-08-16 22:00:00",
                                      "2023-08-17 6:30:00",
                                      "2023-08-18 1:00:00",
                                      "2023-08-18 6:00:00",
                                      "2023-08-18 22:00:00",
                                      "2023-08-19 6:00:00",
                                      "2023-08-19 23:00:00",
                                      "2023-08-20 6:00:00",
                                      "2023-08-20 22:00:00"),
                         State = rep(c("wake", "sleep"), 6),
                         Wear = rep(c("wear", "no wear"), 6),
                         Performance = rep(c(100, 0), 6),
                         Id = "Participant")
```

```
intervals <- sc2interval(states)

#create a dataset with states
dataset_with_states <-
sample.data.environment %>%
interval2state(State.interval.dataset = intervals)

#visualize the states - note that the states are only added to the respective ID in the dataset
library(ggplot2)
ggplot(dataset_with_states, aes(x = Datetime, y = MEDI, color = State)) +
 geom_point() +
 facet_wrap(~Id, ncol = 1)

#import multiple State columns from the interval dataset
#interval2state will only add a single State column to the dataset,
#which represents sleep/wake in our case
dataset_with_states[8278:8283,]

#if we want to add multiple columns we can either perfom the function
#multiple times with different states:
dataset_with_states2 <-
dataset_with_states %>%
interval2state(State.interval.dataset = intervals, State.colname = Wear)
dataset_with_states2[8278:8283,]

#or we can use `purrr::reduce` to add multiple columns at once
dataset_with_states3 <-
syms(c("State", "Wear", "Performance")) %>%
reduce(\(x,y) interval2state(x, State.interval.dataset = intervals, State.colname = !!y),
.init = sample.data.environment)

#Note:
# - the State.colnames have to be provided as symbols (`rlang::syms`)
# - the reduce function requires a two argument function `\(x,y)`, where `x`
#   is the dataset to be continiously modified and `y` is the symbol of the
#   State column name to be added
# - the `!!` operator from `rlang` is used to exchange `y` with each symbol
# - the `.init` argument is the initial dataset to be modified

#this results in all states being applied
dataset_with_states3[8278:8283,]
```

---

intradaily_variability — *Intradaily variability (IV)*

Description

This function calculates the variability of consecutive Light levels within a 24h day. Calculated as the ratio of the variance of the differences between consecutive Light levels to the total variance across the day. Calculated with mean hourly Light levels. Higher values indicate more fragmentation.

Usage

```
intradaily_variability(
  Light.vector,
  Datetime.vector,
  use.samplevar = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

- `Light.vector` Numeric vector containing the light data.
- `Datetime.vector` Vector containing the time data. Must be POSIXct.
- `use.samplevar` Logical. Should the sample variance be used (divide by N-1)? By default (`FALSE`), the population variance (divide by N) is used, as described in Van Someren et al. (1999).
- `na.rm` Logical. Should missing values be removed? Defaults to `FALSE`.
- `as.df` Logical. Should the output be returned as a data frame? If `TRUE`, a data frame with a single column named `intradaily_variability` will be returned. Defaults to `FALSE`.

Value

Numeric value or dataframe with column 'IV'.

References

Van Someren, E. J. W., Swaab, D. F., Colenda, C. C., Cohen, W., McCall, W. V., & Rosenquist, P. B. (1999). Bright Light Therapy: Improved Sensitivity to Its Effects on Rest-Activity Rhythms in Alzheimer Patients by Application of Nonparametric Methods. *Chronobiology International*, 16(4), 505–518. doi:10.3109/07420529908998724

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
set.seed(1)
N <- 24 * 2
# Calculate metric for two 24 h days with two measurements per hour
dataset1 <-
  tibble::tibble(
    Id = rep("A", N * 2),
   Datetime = lubridate::as_datetime(0) + c(lubridate::minutes(seq(0, N * 60 - 30, 30))),
    MEDI = sample(1:1000, N * 2)
  )
dataset1 %>%
  dplyr::summarise(
    "Intradaily variability" = intradaily_variability(MEDI, Datetime)
  )
```

---

join_datasets *Join similar Datasets*

Description

Join Light logging datasets that have a common structure. The least commonality are identical columns for `Datetime` and `Id` across all sets.

Usage

```
join_datasets(
  ...,
  Datetime.column = Datetime,
  Id.column = Id,
  add.origin = FALSE,
  debug = FALSE
)
```

Arguments

`...` `Object` names of datasets that need to be joined.

`Datetime.column, Id.column`
Column names for the `Datetime` and `id` columns. The defaults (`Datetime`, `Id`) are already set up for data imported with LightLogR.

`add.origin` Should a column named `dataset` in the joined data indicate from which dataset each observation originated? Defaults to `FALSE` as the `Id` column should suffice. Expects a `logical`.

`debug` Output changes to a tibble indicating which dataset is missing the respective `Datetime` or `Id` column. Expects a `logical` and defaults to `FALSE`.

Value

One of

- a `data.frame` of joined datasets
- a `tibble` of datasets with missing columns. Only if `debug = TRUE`

Examples

```
#load in two datasets
path <- system.file("extdata",
package = "LightLogR")
file.LL <- "205_actlumus_Log_1020_20230904101707532.txt.zip"
file.env <- "cyepiamb_CW35_Log_1431_20230904081953614.txt.zip"
dataset.LL <- import$ActLumus(file.LL, path = path, auto.id = "^(\\d{3})")
dataset.env <- import$ActLumus(file.env, path = path, manual.id = "CW35")

#join the datasets
joined <- join_datasets(dataset.LL, dataset.env)

#compare the number of rows
nrow(dataset.LL) + nrow(dataset.env) == nrow(joined)

#debug, when set to TRUE, will output a tibble of datasets with missing necessary columns
dataset.LL <- dataset.LL %>% dplyr::select(-Datetime)
join_datasets(dataset.LL, dataset.env, debug = TRUE)
```

`ll_import_expr` *Get the import expression for a device*

Description

Returns the import expression for all device in LightLogR.

Usage

```
ll_import_expr()
```

Details

These expressions are used to import and prepare data from specific devices. The list is made explicit, so that a user, requiring slight changes to the import functions, (e.g., because a timestamp is formatted differently) can modify or add to the list. The list can be turned into a fully functional import function through `import_adjustment()`.

Value

A list of import expressions for all supported devices

See Also

import_Dataset, import_Dataset

Examples

```
ll_import_expr()[1]
```

---

log_zero_inflated — *Add a defined number to a numeric and log transform it*

Description

Frequently, light exposure data need to be log-transformed. Because light exposure data frequently also contain many zero-values, adding a small value avoids losing those observations. Must be applied with care and reported.

`exp_zero_inflated()` is the reverse function to `log_zero_inflated()`.

Usage

```
log_zero_inflated(x, offset = 0.1, base = 10)

exp_zero_inflated(x, offset = 0.1, base = 10)
```

Arguments

`x`	A numeric vector
`offset`	the amount to add to `x`, by default `0.1`
`base`	The logarithmic base, by default `10`

Value

a transformed numeric vector

References

Johannes Zauner, Carolina Guidolin, Manuel Spitschan (2025) How to deal with darkness: Modelling and visualization of zero-inflated personal light exposure data on a logarithmic scale. bioRxiv. doi: https://doi.org/10.1101/2024.12.30.630669

Examples

```
c(0, 1, 10, 100, 1000, 10000) |> log_zero_inflated()

#For use in a function
sample.data.environment |>
  dplyr::filter(Id == "Participant") |>
  dplyr::group_by(Date = lubridate::wday(Datetime, label = TRUE, week_start = 1)) |>
  dplyr::summarize(
  TAT250 = duration_above_threshold(log_zero_inflated(MEDI),
                                    Datetime,
                                    threshold = log_zero_inflated(250)
                                    )
                   )


#Calling exp_zero_inflated on data transformed with log_zero_inflated yields to the original result
c(0, 1, 10, 100, 1000, 10000) |> log_zero_inflated() |> exp_zero_inflated()
```

---

mean_daily *Calculate mean daily metrics from daily summary*

Description

mean_daily calculates a three-row summary of metrics showing average weekday, weekend, and mean daily values of all non-grouping numeric columns. The basis is a dataframe that contains metrics per weekday, or per date (with calculate.from.Date = Datetime). The function requires a column specifying the day of the week as a factor (with Monday as the weekstart), or it can calculate this from a date column if provided.

Usage

```
mean_daily(
  data,
  Weekend.type = Date,
  na.rm = TRUE,
  calculate.from.Date = NULL,
  prefix = "average_",
  filter.empty = FALSE,
  sub.zero = FALSE,
  Datetime2Time = TRUE
)
```

Arguments

data	A dataframe containing the metrics to summarize
Weekend.type	A column in the dataframe that specifies the day of the week as a factor, where weekstart is Monday (so weekends are 6 and 7 in numeric representation). If it is a date, it will be converted to this factor

na.rm	Logical, whether to remove NA values when calculating means. Default is TRUE.
calculate.from.Date	Optional. A column in the dataframe containing dates from which to calculate the Weekend.type. If provided, Weekend.type will be generated from this column.
prefix	String that is the prefix on summarized values
filter.empty	Filter out empty rows. Default is FALSE
sub.zero	Logical. Should missing values be replaced by zero? Defaults to FALSE. Will throw an error, if it happens on a type other than double.
Datetime2Time	Logical of whether POSIXct columns should be transformed into hms(time) columns, which is usually sensible for averaging (default is TRUE). Calls Datetime2Time() with default settings (all POSIXct are affected).

Details

Summary values for type POSIXct are calculated as the mean, which can be nonsensical at times (e.g., the mean of Day1 18:00 and Day2 18:00, is Day2 6:00, which can be the desired result, but if the focus is on time, rather then on datetime, it is recommended that values are converted to times via hms::as_hms() before applying the function (the mean of 18:00 and 18:00 is still 18:00, not 6:00).

Value

A dataframe with three rows representing average weekday, weekend, and mean daily values of all numeric columns

Examples

```
# Create sample data
sample_data <- data.frame(
  Date = factor(c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"),
               levels = c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")),
  lux = c(250, 300, 275, 280, 290, 350, 320),
  duration = lubridate::as.duration(c(120, 130, 125, 135, 140, 180, 160))
)

# Calculate mean daily metrics
mean_daily(sample_data)

# With a Date column
sample_data_with_date <- data.frame(
  Date = seq(as.Date("2023-05-01"), as.Date("2023-05-07"), by = "day"),
  lux = c(250, 300, 275, 280, 290, 350, 320),
  duration = lubridate::as.duration(c(120, 130, 125, 135, 140, 180, 160))
)

mean_daily(sample_data_with_date)
```

mean_daily_metric *Calculate mean daily metrics from Time Series*

Description

mean_daily_metric is a convenience wrapper around mean_daily that summarizes data imported with LightLogR per weekday and calculates mean daily values for a specific metric. Examples include duration_above_threshold() (the default), or durations().

Usage

```
mean_daily_metric(
  data,
  Variable,
  Weekend.type = Date,
  Datetime.colname = Datetime,
  metric_type = duration_above_threshold,
  prefix = "average_",
  filter.empty = FALSE,
  Datetime2Time = TRUE,
  ...
)
```

Arguments

data	A dataframe containing light logger data imported with LightLogR
Variable	The variable column to analyze. Expects a symbol. Needs to be part of the dataset.
Weekend.type	A (new) column in the dataframe that specifies the day of the week as a factor
Datetime.colname	Column name containing datetime values. Defaults to Datetime
metric_type	The metric function to apply, default is duration_above_threshold()
prefix	String that is the prefix on summarized values
filter.empty	Filter out empty rows. Default is FALSE
Datetime2Time	Logical of whether POSIXct columns should be transformed into hms(time) columns, which is usually sensible for averaging (default is TRUE). Calls Datetime2Time() with default settings (all POSIXct are affected).
...	Additional arguments passed to the metric function

Value

A dataframe with three rows representing average weekday, weekend, and mean daily values for the specified metric

Examples

```
# Calculate mean daily duration above threshold. As the data only contains
# data for two days, Weekend and Mean daily will throw NA
sample.data.irregular |>
aggregate_Datetime(unit = "1 min") |>
mean_daily_metric(
  Variable = lux,
  threshold = 100
)

# again with another dataset
sample.data.environment |>
  mean_daily_metric(
  Variable = MEDI,
  threshold = 250)

# by default, datetime columns are converted to time
sample.data.environment |>
  mean_daily_metric(
  Variable = MEDI,
  metric_type = timing_above_threshold,
  threshold = 250)
```

---

midpointCE — *Midpoint of cumulative light exposure.*

Description

This function calculates the timing corresponding to half of the cumulative light exposure within the given time series.

Usage

```
midpointCE(Light.vector, Time.vector, na.rm = FALSE, as.df = FALSE)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`na.rm`	Logical. Should missing values be removed for the calculation? If `TRUE`, missing values will be replaced by zero. Defaults to `FALSE`.
`as.df`	Logical. Should the output be returned as a data frame? If `TRUE`, a data frame with a single column named `midpointCE` will be returned. Defaults to `FALSE`.

Value

Single column data frame or vector.

References

Shochat, T., Santhi, N., Herer, P., Flavell, S. A., Skeldon, A. C., & Dijk, D.-J. (2019). Sleep Timing in Late Autumn and Late Spring Associates With Light Exposure Rather Than Sun Time in College Students. *Frontiers in Neuroscience*, 13. doi:10.3389/fnins.2019.00882

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::as_datetime(0) + lubridate::hours(0:23),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset1 %>%
  dplyr::reframe(
    "Midpoint of cmulative exposure" = midpointCE(MEDI, Datetime)
  )

# Dataset with HMS time vector
dataset2 <-
  tibble::tibble(
    Id = rep("A", 24),
    Time = hms::as_hms(lubridate::as_datetime(0) + lubridate::hours(0:23)),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset2 %>%
  dplyr::reframe(
    "Midpoint of cmulative exposure" = midpointCE(MEDI, Time)
  )

# Dataset with duration time vector
dataset3 <-
  tibble::tibble(
    Id = rep("A", 24),
    Hour = lubridate::duration(0:23, "hours"),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )
dataset3 %>%
  dplyr::reframe(
    "Midpoint of cmulative exposure" = midpointCE(MEDI, Hour)
  )
```

normalize_counts — *Normalize counts between sensor outputs*

Description

This is a niche helper function to normalize counts. Some sensors provide raw counts and gain levels as part of their output. In some cases it is desirable to compare counts between sensors, e.g., to gauge daylight outside by comparing UV counts to photopic counts (a high ratio of UV/Pho indicates outside daylight). Or to gauge daylight inside by comparing IR counts to photopic counts (a high ratio of IR/Pho with a low ratio of UV/Pho indicates daylight in the context of LED or fluorescent lighting). The user can provide their own gain ratiotable, or use a table provided for a sensor in the `gain.ratio.table` dataset from `LightLogR`.

Usage

```
normalize_counts(dataset, gain.columns, count.columns, gain.ratio.table)
```

Arguments

`dataset` a `data.table` containing gain and count columns.

`gain.columns` a `character` vector of columns in the `dataset` containing a gain setting. Columns must not repeat.

`count.columns` a `character` vector of columns in the `dataset` containing raw count data. Must be of the same length as `gain.columns`, and the order must conform to the order in `gain.columns`.

`gain.ratio.table` a two-column tibble containing `gain` and `gain.ratio` information. Can be provided by the user or use the `gain.ratio.table` dataset.

Value

an extended dataset with new columns containing normalized counts

See Also

Other Spectrum: `spectral_integration()`, `spectral_reconstruction()`

Examples

```
example.table <-
tibble::tibble(
uvGain = c(4096, 1024, 2),
visGain = c(4096, 4096, 4096),
irGain = c(2,2,2),
uvValue = c(692, 709, 658),
visValue = c(128369, 129657, 128609),
irValue = c(122193, 127113, 124837))
```

```
gain.columns = c("uvGain", "visGain", "irGain")
count.columns = c("uvValue", "visValue", "irValue")

example.table |>
normalize_counts(gain.columns, count.columns, gain.ratio.tables$TSL2585)
```

number_states   *Number non-consecutive state occurrences*

Description

number_states() creates a new column in a dataset that takes a state column and assigns a count value to each state, rising every time a state is replaced by another state. E.g., a column with the states "day" and "night" will produce a column indicating whether this is "day 1", "day 2", and so forth, as will the "night" state with "night 1", "night 2", etc. Grouping within the input dataset is respected, i.e., the count will reset for each group.

Usage

```
number_states(
  dataset,
  state.colname,
  colname.extension = ".count",
  use.original.state = TRUE
)
```

Arguments

dataset	A data.frame with a state column.
state.colname	Column name that contains the state. Expects a symbol. Needs to be part of the dataset. Can be of any type, but character and factor make the most sense.
colname.extension	The extension that is added to the state name to create the new column. Defaults to ".count".
use.original.state	Logical, whether the original state should be part of the output column.

Details

The state column is not limited to two states, but can have as many states as needed. Also, it does not matter in which time frames these states change, so they do not necessarily conform to a 24-hour day. NA values will be treated as their own state.

Gaps in the data can lead to non-sensible outcomes, e.g. if there is no in-between state/observation between a day state at "18:00:00" and a day state at "6:00:00" - this would be counted as day 1 still. In these cases, the gap_handler() function can be useful to a priori add observations.

Value

The input `dataset` with an additional column that counts the occurrences of each state. The new column will of type `character` if `use.original.state = TRUE` and `integer` otherwise.

Examples

```
dataset <- tibble::tibble(
 state =
 c("day", "day", "day", "night", "night", "day", "day", "night",
 "night", "night", "day", "night")
 )
number_states(dataset, state)
number_states(dataset, state, use.original.state = FALSE)

#example with photoperiods, calculating the mean values for each day and night
coordinates <- c(48.52, 9.06)
sample.data.environment |>
  add_photoperiod(coordinates) |>
  number_states(photoperiod.state) |>
  dplyr::group_by(photoperiod.state.count, .add = TRUE) |>
  dplyr::summarize(mean_MEDI = mean(MEDI)) |>
  tail(13)
```

`nvRC` *Non-visual circadian response*

Description

This function calculates the non-visual circadian response (nvRC). It takes into account the assumed response dynamics of the non-visual system and the circadian rhythm and processes the light exposure signal to quantify the effective circadian-weighted input to the non-visual system (see Details).

Usage

```
nvRC(
  MEDI.vector,
  Illuminance.vector,
  Time.vector,
  epoch = "dominant.epoch",
  sleep.onset = NULL
)
```

Arguments

`MEDI.vector` Numeric vector containing the melanopic EDI data.

`Illuminance.vector`
Numeric vector containing the Illuminance data.

Time.vector	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
epoch	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either "dominant.epoch" (the default) for a guess based on the data, or a valid duration string, e.g., "1 day" or "10 sec".
sleep.onset	The time of habitual sleep onset. Can be HMS, numeric, or NULL. If NULL (the default), then the data is assumed to start at habitual sleep onset. If Time.vector is HMS or POSIXct, sleep.onset must be HMS. Likewise, if Time.vector is numeric, sleep.onset must be numeric.

Details

The timeseries is assumed to be regular. Missing values in the light data will be replaced by 0.

Value

A numeric vector containing the nvRC data. The output has the same length as Time.vector.

References

Amundadottir, M.L. (2016). Light-driven model for identifying indicators of non-visual health potential in the built environment [Doctoral dissertation, EPFL]. EPFL infoscience. doi:10.5075/epfl-thesis7146

See Also

Other metrics: bright_dark_period(), centroidLE(), disparity_index(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("B", 60 * 48),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*48-1)),
    Illuminance = c(rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60),
                    rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60)),
    MEDI = Illuminance * rep(sample(0.5:1.5, 48, replace = TRUE), each = 60)
  )
# Time.vector as POSIXct
dataset1.nvRC <- dataset1 %>%
  dplyr::mutate(
    nvRC = nvRC(MEDI, Illuminance, Datetime, sleep.onset = hms::as_hms("22:00:00"))
  )

# Time.vector as difftime
dataset2 <- dataset1 %>%
  dplyr::mutate(Datetime = Datetime - lubridate::as_datetime(lubridate::dhours(22)))
dataset2.nvRC <- dataset2 %>%
  dplyr::mutate(
```

```
  nvRC = nvRC(MEDI, Illuminance, Datetime, sleep.onset = lubridate::dhours(0))
)
```

---

nvRC_metrics *Performance metrics for circadian response*

Description

These functions compare the non-visual circadian response (see nvRC) for measured personal light exposure to the nvRC for a reference light exposure pattern, such as daylight.

Usage

```
nvRC_circadianDisturbance(nvRC, nvRC.ref, as.df = FALSE)

nvRC_circadianBias(nvRC, nvRC.ref, as.df = FALSE)

nvRC_relativeAmplitudeError(nvRC, nvRC.ref, as.df = FALSE)
```

Arguments

nvRC	Time series of non-visual circadian response (see nvRC.
nvRC.ref	Time series of non-visual circadian response circadian response (see nvRC for a reference light exposure pattern (e.g., daylight). Must be the same length as nvRC.
as.df	Logical. Should the output be returned as a data frame? Defaults to TRUE.

Details

`nvRC_circadianDisturbance()` calculates the circadian disturbance (CD). It is expressed as

$$CD(i,T) = \frac{1}{T}\int_{t_i}^{t_i+T} |r_C(t) - r_C^{ref}(t)|dt,$$

and quantifies the total difference between the measured circadian response and the circadian response to a reference profile.

`nvRC_circadianBias()` calculates the circadian bias (CB). It is expressed as

$$CB(i,T) = \frac{1}{T}\int_{t_i}^{t_i+T} (r_C(t) - r_C^{ref}(t))dt,$$

and provides a measure of the overall trend for the difference in circadian response, i.e. positive values for overestimating and negative for underestimating between the measured circadian response and the circadian response to a reference profile.

`nvRC_relativeAmplitudeError()` calculates the relative amplitude error (RAE). It is expressed as

$$RAE(i,T) = r_{C,max} - r_{C,max}^{ref},$$

and quantifies the difference between the maximum response achieved in a period to the reference signal.

Value

A numeric value or single column data frame.

References

Amundadottir, M.L. (2016). Light-driven model for identifying indicators of non-visual health potential in the built environment [Doctoral dissertation, EPFL]. EPFL infoscience. doi:10.5075/epflthesis7146

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("B", 60 * 24),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*24-1)),
    Illuminance = c(rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60)),
    MEDI = Illuminance * rep(sample(0.5:1.5, 24, replace = TRUE), each = 60),
  ) %>%
  dplyr::mutate(
    nvRC = nvRC(MEDI, Illuminance, Datetime, sleep.onset = hms::as_hms("22:00:00"))
  )

dataset.reference <-
  tibble::tibble(
    Id = rep("Daylight", 60 * 24),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*24-1)),
    Illuminance = c(rep(0, 60*6), rep(10000, 12*60), rep(0, 60*6)),
    MEDI = Illuminance
  ) %>%
  dplyr::mutate(
    nvRC = nvRC(MEDI, Illuminance, Datetime, sleep.onset = hms::as_hms("22:00:00"))
  )

# Circadian disturbance
nvRC_circadianDisturbance(dataset1$nvRC, dataset.reference$nvRC)

# Circadian bias
nvRC_circadianBias(dataset1$nvRC, dataset.reference$nvRC)

# Relative amplitude error
nvRC_relativeAmplitudeError(dataset1$nvRC, dataset.reference$nvRC)
```

nvRD — *Non-visual direct response*

Description

This function calculates the non-visual direct response (nvRD). It takes into account the assumed response dynamics of the non-visual system and processes the light exposure signal to quantify the effective direct input to the non-visual system (see Details).

Usage

```
nvRD(MEDI.vector, Illuminance.vector, Time.vector, epoch = "dominant.epoch")
```

Arguments

`MEDI.vector`	Numeric vector containing the melanopic EDI data.
`Illuminance.vector`	Numeric vector containing the Illuminance data.
`Time.vector`	Vector containing the time data. Can be `POSIXct()`,`hms::hms()`, `lubridate::duration()`, `difftime()`.
`epoch`	The epoch at which the data was sampled. Can be either a `lubridate::duration()` or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid `lubridate::duration()` string, e.g., `"1 day"` or `"10 sec"`.

Details

The timeseries is assumed to be regular. Missing values in the light data will be replaced by 0.

Value

A numeric vector containing the nvRD data. The output has the same length as `Time.vector`.

References

Amundadottir, M.L. (2016). Light-driven model for identifying indicators of non-visual health potential in the built environment [Doctoral dissertation, EPFL]. EPFL infoscience. doi:10.5075/epflthesis7146

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
# Dataset 1 with 24h measurement
dataset1 <-
  tibble::tibble(
    Id = rep("A", 60 * 24),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*24-1)),
    Illuminance = c(rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60)),
    MEDI = Illuminance * rep(sample(0.5:1.5, 24, replace = TRUE), each = 60)
  )
# Dataset 2 with 48h measurement
dataset2 <-
  tibble::tibble(
    Id = rep("B", 60 * 48),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*48-1)),
    Illuminance = c(rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60),
                    rep(0, 60*8), rep(sample(1:1000, 16, replace = TRUE), each = 60)),
    MEDI = Illuminance * rep(sample(0.5:1.5, 48, replace = TRUE), each = 60)
  )
# Combined datasets
dataset.combined <- rbind(dataset1, dataset2)

# Calculate nvRD per ID
dataset.combined.nvRD <- dataset.combined %>%
  dplyr::group_by(Id) %>%
  dplyr::mutate(
    nvRD = nvRD(MEDI, Illuminance, Datetime)
  )
```

---

nvRD_cumulative_response

Cumulative non-visual direct response

Description

This function calculates the cumulative non-visual direct response (nvRD). This is basically the integral of the nvRD over the provided time period in hours. The unit of the resulting value thus is "nvRD*h".

Usage

```
nvRD_cumulative_response(
  nvRD,
  Time.vector,
  epoch = "dominant.epoch",
  as.df = FALSE
)
```

Arguments

`nvRD`	Numeric vector containing the non-visual direct response. See nvRD.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.
`as.df`	Logical. Should a data frame with be returned? If `TRUE`, a data frame with a single column named `nvRD_cumulative` will be returned. Defaults to `FALSE`.

Value

A numeric value or single column data frame.

References

Amundadottir, M.L. (2016). Light-driven model for identifying indicators of non-visual health potential in the built environment [Doctoral dissertation, EPFL]. EPFL infoscience. doi:10.5075/epflthesis7146

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`, `timing_above_threshold()`

Examples

```
dataset1 <-
  tibble::tibble(
    Id = rep("A", 60 * 24),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(0:(60*24-1)),
   Illuminance = c(rep(0, 60*8), rep(sample(1:1000, 14, replace = TRUE), each = 60), rep(0, 60*2)),
    MEDI = Illuminance * rep(sample(0.5:1.5, 24, replace = TRUE), each = 60)
  ) %>%
  dplyr::mutate(
    nvRD = nvRD(MEDI, Illuminance, Datetime)
  )
dataset1 %>%
  dplyr::summarise(
    "cumulative nvRD" = nvRD_cumulative_response(nvRD, Datetime)
  )
```

period_above_threshold

Length of longest continuous period above/below threshold

Description

This function finds the length of the longest continous period above/below a specified threshold light level or within a specified range of light levels.

Usage

```
period_above_threshold(
  Light.vector,
  Time.vector,
  comparison = c("above", "below"),
  threshold,
  epoch = "dominant.epoch",
  loop = FALSE,
  na.replace = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

Light.vector	Numeric vector containing the light data.
Time.vector	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
comparison	String specifying whether the period of light levels above or below threshold should be calculated. Can be either "above" (the default) or "below". If two values are provided for threshold, this argument will be ignored.
threshold	Single numeric value or two numeric values specifying the threshold light level(s) to compare with. If a vector with two values is provided, the period of light levels within the two thresholds will be calculated.
epoch	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either "dominant.epoch" (the default) for a guess based on the data, or a valid duration string, e.g., "1 day" or "10 sec".
loop	Logical. Should the data be looped? Defaults to FALSE.
na.replace	Logical. Should missing values (NA) be replaced for the calculation? If TRUE missing values will not be removed but will result in FALSE when comparing Light.vector with threshold. Defaults to FALSE.
na.rm	Logical. Should missing values (NA) be removed for the calculation? If TRUE, this argument will override na.replace. Defaults to FALSE.
as.df	Logical. Should a data frame be returned? If TRUE, a data frame with a single column named period_{comparison}_{threshold} will be returned. Defaults to FALSE.

Value

A duration object (see duration) as single value, or single column data frame.

See Also

Other metrics: bright_dark_period(), centroidLE(), disparity_index(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), pulses_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
N <- 60
# Dataset with continous period of >250lx for 35min
dataset1 <-
  tibble::tibble(
    Id = rep("A", N),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(1:N),
    MEDI = c(sample(1:249, N-35, replace = TRUE),
             sample(250:1000, 35, replace = TRUE))
  )

dataset1 %>%
 dplyr::reframe("Period >250lx" = period_above_threshold(MEDI, Datetime, threshold = 250))

dataset1 %>%
 dplyr::reframe("Period <250lx" = period_above_threshold(MEDI, Datetime, "below", threshold = 250))

# Dataset with continous period of 100-250lx for 20min
dataset2 <-
  tibble::tibble(
    Id = rep("B", N),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(1:N),
    MEDI = c(sample(c(1:99, 251-1000), N-20, replace = TRUE),
             sample(100:250, 20, replace = TRUE)),
  )
dataset2 %>%
 dplyr::reframe("Period 250lx" = period_above_threshold(MEDI, Datetime, threshold = c(100,250)))

# Return data frame
dataset1 %>%
  dplyr::reframe(period_above_threshold(MEDI, Datetime, threshold = 250, as.df = TRUE))
```

---

photoperiod — *Calculate photoperiod and boundary times*

Description

A family of functions to extract and add photoperiod information. `photoperiod()` creates a `tibble` with the calculated times of dawn and dusk for the given location and date. The function is a convenience wrapper for `suntools::crepuscule()` to calculate the times of dawn and dusk. By default, civil dawn and dusk are calculated, but the function can be used to calculate other times by changing the `solarDep` parameter (e.g., 0 for sunrise/sunset, 12 for nautical, and 18 for astronomical).

Taking a light exposure dataset as input, `extract_photoperiod()` calculates the photoperiods and their boundary times for each unique day in the dataset, given a location and boundary condition (i.e., the solar depression angle). Basically, this is a convenience wrapper for `photoperiod()` that takes a light logger dataset and extracts unique dates and the time zone from the dataset.

`add_photoperiod()` adds photoperiod information to a light logger dataset. Beyond the photoperiod information, it will categorize the `photoperiod.state` as `"day"` or `"night"`. If `overwrite` is set to `TRUE`, the function will overwrite any columns with the same name.

`solar_noon()` calculates the solar noon for a given location and date. The function is a convenience wrapper for `suntools::solarnoon()`. The function has no companions like `extract_photoperiod()` or `add_photoperiod()`, but will be extended, if there is sufficient interest.

Usage

```
photoperiod(coordinates, dates, tz, solarDep = 6)

extract_photoperiod(
  dataset,
  coordinates,
  Datetime.colname = Datetime,
  solarDep = 6
)

add_photoperiod(
  dataset,
  coordinates,
  Datetime.colname = Datetime,
  solarDep = 6,
  overwrite = FALSE
)

solar_noon(coordinates, dates, tz)
```

Arguments

`coordinates`	A two element numeric vector representing the latitude and longitude of the location. *Important note:* **Latitude** *is the first element and* **Longitude** *is the second element.*
`dates`	A date of format `Date`, or coercible to `Date` through `lubridate::as_date()`
`tz`	Timezone of the data. Expects a `character`. You can look up the supported timezones with `OlsonNames()`.

solarDep	A numerical value representing the solar depression angle between 90 and -90. This means a value of 6 equals **-6** degrees above the horizon. Default is 6, equalling `Civil dawn/dusk`. Other common values are 12 degrees for `Nautical dawn/dusk`, 18 degrees for `Astronomical dawn/dusk`, and 0 degrees for `Sunrise/Sunset`. Note that the output columns will always be named `dawn` and `dusk`, regardless of the `solarDep` value.
dataset	A light logger dataset. Expects a `dataframe`. If not imported by LightLogR, take care to choose a sensible variable for the `Datetime.colname`.
Datetime.colname	column name that contains the datetime. Defaults to `"Datetime"` which is automatically correct for data imported with LightLogR. Expects a `symbol`. Needs to be part of the `dataset`. Must be of type `POSIXct`.
overwrite	Logical scalar. If `TRUE`, the function will overwrite any columns with the same name. If `FALSE` (default), the function will stop if any of the columns already exist in the dataset.

Details

Please note that all functions of the `photoperiod` family work with one coordinate pair at a time. If you have multiple locations (and multiple time zones), you need to run the function for each location separately. We suggest using a nested dataframe structure, and employ the `purrr` package to iterate over the locations.

Value

`photoperiod()` returns a `tibble` with the calculated times of dawn and dusk for the given location and date, with the length equal to the `dates` input parameter . The tibble contains the following columns:

- `date` with the date of the calculation, stored as class `Date`
- `tz` with the timezone of the output, stored as class `character`
- `lat` and `lon` with the latitude and longitude of the location, stored as class `numeric`
- `solar.angle` with the negative solar depression angle, i.e. the sun elevation above the horizon. stored as class `numeric`
- `dawn` and `dusk` with the calculated datetimes, stored as class `POSIXct`
- `photoperiod` with the calculated photoperiod, stored as class `difftime`.

`extract_photoperiod()` returns a `tibble` of the same structure as `photoperiod()`, but with a length equal to the number of unique dates in the dataset.

add_photoperiod returns the input dataset with the added photoperiod information. The information is appended with the following columns: `dawn`, `dusk`, `photoperiod`, and `photoperiod.state`.

`solar_noon()` returns a `tibble` with the calculated solar noon

See Also

Other photoperiod: `gg_photoperiod()`

Examples

```
#example für Tübingen, Germany
coordinates <- c(48.521637, 9.057645)
dates <- c("2023-06-01", "2025-08-23")
tz <- "Europe/Berlin"

#civil dawn/dusk
photoperiod(coordinates, dates, tz)
#sunrise/sunset
photoperiod(coordinates, dates, tz, solarDep = 0)
#extract_photoperiod
sample.data.environment |>
  extract_photoperiod(coordinates)


#add_photoperiod
added_photoperiod <-
 sample.data.environment |>
 add_photoperiod(coordinates)

added_photoperiod |> head()

added_photoperiod |>
  filter_Date(length = "3 days") |>
  gg_days(aes_col = photoperiod.state,
          group = dplyr::consecutive_id(photoperiod.state),
          jco_color = TRUE
          )

added_photoperiod |>
  filter_Date(length = "3 days") |>
  gg_day(aes_col = Id) +
  ggplot2:: geom_rect(
 data = \(x) x |> dplyr::ungroup(Id) |> dplyr::summarize(dawn = mean(dawn) |> hms::as_hms()),
  ggplot2::aes(xmin = 0, xmax = dawn, ymin = -Inf, ymax = Inf),
  alpha = 0.1
  ) +
  ggplot2:: geom_rect(
 data = \(x) x |> dplyr::ungroup(Id) |> dplyr::summarize(dusk = mean(dusk) |> hms::as_hms()),
  ggplot2::aes(xmin = dusk, xmax = 24*60*60, ymin = -Inf, ymax = Inf),
  alpha = 0.1
  )


 added_photoperiod |> dplyr::summarize(dawn = mean(dawn) |> hms::as_hms())

 #solar_noon()
 solar_noon(coordinates, dates, tz)
```

`pulses_above_threshold` *Pulses above threshold*

Description

This function clusters the light data into continuous clusters (pulses) of light above/below a given threshold. Clustering may be fine-tuned by setting the minimum length of the clusters and by allowing brief interruptions to be included in a single cluster, with a specified maximum length of interruption episodes and proportion of total amount of interruptions to light above threshold.

Usage

```
pulses_above_threshold(
  Light.vector,
  Time.vector,
  comparison = c("above", "below"),
  threshold,
  min.length = "2 mins",
  max.interrupt = "8 mins",
  prop.interrupt = 0.25,
  epoch = "dominant.epoch",
  return.indices = FALSE,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data. Missing values will be considered as `FALSE` when comparing light levels against the threshold.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`comparison`	String specifying whether the time above or below threshold should be calculated. Can be either `"above"` (the default) or `"below"`. If two values are provided for `threshold`, this argument will be ignored.
`threshold`	Single numeric value or two numeric values specifying the threshold light level(s) to compare with. If a vector with two values is provided, the timing corresponding to light levels between the two thresholds will be calculated.
`min.length`	The minimum length of a pulse. Can be either a duration or a string. If it is a string, it needs to be a valid duration string, e.g., `"1 day"` or `"10 sec"`. Defaults to `"2 mins"` as in Wilson et al. (2018).
`max.interrupt`	Maximum length of each episode of interruptions. Can be either a duration or a string. If it is a string, it needs to be a valid duration string, e.g., `"1 day"` or `"10 sec"`. Defaults to `"8 mins"` as in Wilson et al. (2018).
`prop.interrupt`	Numeric value between `0` and `1` specifying the maximum proportion of the total number of interruptions. Defaults to `0.25` as in Wilson et al. (2018).

epoch	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either "dominant.epoch" (the default) for a guess based on the data, or a valid duration string, e.g., "1 day" or "10 sec".
return.indices	Logical. Should the cluster indices be returned? Only works if as.df is FALSE. Defaults to FALSE.
na.rm	Logical. Should missing values be removed for the calculation of pulse metrics? Defaults to FALSE.
as.df	Logical. Should a data frame be returned? If TRUE, a data frame with seven columns ("n", "mean_level", "mean_duration", "total_duration", "mean_onset", "mean_midpoint", "mean_offset") and the threshold (e.g., _{threshold}) will be returned. Defaults to FALSE.

Details

The timeseries is assumed to be regular. Missing values in the light data will be replaced by 0.

Value

List or data frame with calculated values.

References

Wilson, J., Reid, K. J., Braun, R. I., Abbott, S. M., & Zee, P. C. (2018). Habitual light exposure relative to circadian timing in delayed sleep-wake phase disorder. *Sleep*, 41(11). doi:10.1093/sleep/zsy166

See Also

Other metrics: bright_dark_period(), centroidLE(), disparity_index(), dose(), duration_above_threshold(), exponential_moving_average(), frequency_crossing_threshold(), interdaily_stability(), intradaily_variability(), midpointCE(), nvRC(), nvRD(), nvRD_cumulative_response(), period_above_threshold(), threshold_for_duration(), timing_above_threshold()

Examples

```
# Sample data
data = sample.data.environment %>%
  dplyr::filter(Id == "Participant") %>%
  filter_Datetime(length = lubridate::days(1)) %>%
  dplyr::mutate(
    Time = hms::as_hms(Datetime),
  )

# Time vector as datetime
data %>%
  dplyr::reframe(pulses_above_threshold(MEDI, Datetime, threshold = 250, as.df = TRUE))

# Time vector as hms time
data %>%
  dplyr::reframe(pulses_above_threshold(MEDI, Time, threshold = 250, as.df = TRUE))
```

```
# Pulses below threshold
data %>%
  dplyr::reframe(pulses_above_threshold(MEDI, Datetime, "below", threshold = 250, as.df = TRUE))

# Pulses within threshold range
data %>%
  dplyr::reframe(pulses_above_threshold(MEDI, Datetime, threshold = c(250,1000), as.df = TRUE))
```

`remove_partial_data` *Remove groups that have too few data points*

Description

This function removes groups from a dataframe that do not have sufficient data points. Groups of one data point will automatically be removed. Single data points are common after using `aggregate_Datetime()`.

Usage

```
remove_partial_data(
  dataset,
  Variable.colname = Datetime,
  threshold.missing = 0.2,
  by.date = FALSE,
  Datetime.colname = Datetime,
  show.result = FALSE,
  handle.gaps = FALSE
)
```

Arguments

`dataset` A light logger dataset. Expects a dataframe. If not imported by LightLogR, take care to choose sensible variables for the Datetime.colname and Variable.colname.

`Variable.colname` Column name that contains the variable for which to assess sufficient datapoints. Expects a symbol. Needs to be part of the dataset. Default is `Datetime`, which makes only sense in the presence of single data point groups that need to be removed.

`threshold.missing` either

- percentage of missing data, before that group gets removed. Expects a numeric scalar.
- duration of missing data, before that group gets removed. Expects either a `lubridate::duration()` or a character that can be converted to one, e.g., "30 mins".

by.date	Logical. Should the data be (additionally) grouped by day? Defaults to FALSE. Additional grouping is not persitant beyond the function call.
Datetime.colname	Column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol. Needs to be part of the dataset. Must be of type POSIXct.
show.result	Logical, whether the output of the function is summary of the data (TRUE), or the reduced dataset (FALSE, the default)
handle.gaps	Logical, whether the data shall be treated with gap_handler(). Is set to FALSE by default. If TRUE, it will be used with the argument full.days = TRUE.

Value

if show.result = FALSE(default), a reduced dataframe without the groups that did not have sufficient data

Examples

```
#create sample data with gaps
gapped_data <-
  sample.data.environment |>
  dplyr::filter(MEDI < 30000)

#check their status, based on the MEDI variable
gapped_data |> remove_partial_data(MEDI, handle.gaps = TRUE, show.result = TRUE)

#the function will produce a warning if implicit gaps are present
gapped_data |> remove_partial_data(MEDI, show.result = TRUE)

#one group (Environment) does not make the cut of 20% missing data
gapped_data |> remove_partial_data(MEDI, handle.gaps = TRUE) |> dplyr::count(Id)
#for comparison
gapped_data |> dplyr::count(Id)
#If the threshold is set differently, e.g., to 2 days allowed missing, results vary
gapped_data |>
  remove_partial_data(MEDI, handle.gaps = TRUE, threshold.missing = "2 days") |>
  dplyr::count(Id)

#The removal can be automatically switched to daily detections within groups
gapped_data |>
 remove_partial_data(MEDI, handle.gaps = TRUE, by.date = TRUE, show.result = TRUE) |>
 head()
```

---

reverse2_trans *Create a reverse transformation function specifically for date scales*

Description

This helper function is exclusive for gg_heatmap(), to get a reversed date sequence.

Usage

```
reverse2_trans()
```

Value

a transformation function

Source

from https://github.com/tidyverse/ggplot2/issues/4014

Examples

```
reverse2_trans()
```

`sample.data.environment` *Sample of wearable data combined with environmental data*

Description

A subset of data from a study at the TSCN-Lab using the ActLumus light logger. This dataset contains personal light exposure information for one participant over the course of six full days. The dataset is measured with a 10 second epoch and is complete (no missing values). Additionally environmental light data was captured with a second light logger mounted horizontally at the TUM university roof, without any obstructions (besides a transparent plastic halfdome). The epoch for this data is 30 seconds. This dataset allows for some interesting calculations based on *available* daylight at a given point in time.

Usage

```
sample.data.environment
```

Format

`sample.data.environment` A tibble with 69,120 rows and 3 columns:

Datetime POSIXct Datetime

MEDI melanopic EDI measurement data. Unit is lux.

Id A `character` vector indicating whether the data is from the `Participant` or from the `Environment`.

Source

https://www.tscnlab.org

`sample.data.irregular` *Sample of highly irregular wearable data*

Description

A dataset collected with a wearable device that has a somewhat irregular recording pattern. Overall, the data are recorded every 15 seconds. Every tenth or so measurement takes 16 seconds, every hundredths 17 seconds, every thousandths 18 seconds, and so on. This makes the dataset a prime example for handling and dealing with irregular data.

Usage

```
sample.data.irregular
```

Format

`sample.data.irregular` A tibble with 11,422 rows and 13 columns:

Id A `character` vector indicating the participant (only `P1`).

Datetime POSIXct Datetime

lux numeric Illuminance. Unit is lux.

kelvin numeric correlated colour temperature (CCT). Unit is Kelvin.

rgbR numeric red sensor channel output. Unit is W/m2/nm.

rgbG numeric green sensor channel output. Unit is W/m2/nm.

rgbB numeric blue sensor channel output. Unit is W/m2/nm.

rgbIR numeric infrared sensor channel output. Unit is W/m2/nm.

movement numeric indicator for movement (intensity) of the device. Movement is given in discrete counts correlating to the number of instances the accelerometer records instances greater than 0.1875g per 15s sampling interval.

MEDI melanopic EDI measurement data. Unit is lux.

R. Unknown, but likely direct or derived output from the red sensor channel

G. Unknown, but likely direct or derived output from the green sensor channel

B. Unknown, but likely direct or derived output from the blue sensor channel

sc2interval — *Statechange (sc) Timestamps to Intervals*

Description

Takes an input of datetimes and Statechanges and creates a column with Intervals. If full = TRUE, it will also create intervals for the day prior to the first state change and after the last. If output.dataset = FALSE it will give a named vector, otherwise a tibble. The state change info requires a description or name of the state (like "sleep" or "wake", or "wear") that goes into effect at the given Datetime. Works for grouped data so that it does not mix up intervals between participants. Missing data should be explicit if at all possible. Also, the maximum allowed length of an interval can be set, so that implicit missing timestamps after a set period of times can be enforced.

Usage

```
sc2interval(
  dataset,
  Datetime.colname = Datetime,
  Statechange.colname = State,
  State.colname = State,
  Interval.colname = Interval,
  full = TRUE,
  starting.state = NA,
  output.dataset = TRUE,
  Datetime.keep = FALSE,
  length.restriction = 60 * 60 * 24
)
```

Arguments

`dataset` A light logger dataset. Expects a dataframe. If not imported by LightLogR, take care to choose a sensible variable for the Datetime.colname.

`Datetime.colname` column name that contains the datetime. Defaults to "Datetime" which is automatically correct for data imported with LightLogR. Expects a symbol. Needs to be part of the dataset. Must be of type POSIXct.

`Statechange.colname, Interval.colname, State.colname` Column names that do contain the name/description of the state change and that will contain the Interval and State (which are also the default). Expects a symbol. The Statechange column needs do be part of the dataset.

`full, starting.state` These arguments handle the state on the first day before the first state change and after the last state change on the last day. If full = TRUE(the default, expects a logical), it will create an interval on the first day from 00:00:00 up until the state change. This interval will be given the state specified in starting.state, which is NA by default, but can be any character scalar. It will further extend the interval for the last state change until the end of the last given day (more specifically until 00:00:00 the next day).

output.dataset should the output be a data.frame (Default TRUE) or a vector with hms (FALSE) times? Expects a logical scalar.

Datetime.keep If TRUE, the original Datetime column will be kept.

length.restriction
If the length between intervals is too great, the interval state can be set to NA, which effectively produces a gap in the data. This makes sense when intervals are implausibly wrong (e.g. someone slept for 50 hours), because when this data is combined with light logger data, e.g., through interval2state(), metrics and visualizations will remove the interval.

Value

One of

- a data.frame object identical to dataset but with the interval instead of the datetime. The original Statechange column now indicates the State during the Interval.
- a named vector with the intervals, where the names are the states

Examples

```
library(tibble)
library(lubridate)
library(dplyr)
sample <- tibble::tibble(Datetime = c("2023-08-15 6:00:00",
                                      "2023-08-15 23:00:00",
                                      "2023-08-16 6:00:00",
                                      "2023-08-16 22:00:00",
                                      "2023-08-17 6:30:00",
                                      "2023-08-18 1:00:00"),
                         State = rep(c("wake", "sleep"), 3),
                         Id = "Participant")
#intervals from sample
sc2interval(sample)

#compare sample (y) and intervals (x)
sc2interval(sample) %>%
 mutate(Datetime = int_start(Interval)) %>%
 dplyr::left_join(sample, by = c("Id", "State"),
                  relationship = "many-to-many") %>%
 head()
```

sleep_int2Brown *Recode Sleep/Wake intervals to Brown state intervals*

Description

Takes a dataset with sleep/wake intervals and recodes them to Brown state intervals. Specifically, it recodes the `sleep` intervals to `night`, reduces `wake` intervals by a specified `evening.length` and recodes them to `evening` and `day` intervals. The `evening.length` is the time between `day` and `night`. The result can be used as input for `interval2state()` and might be used subsequently with `Brown2reference()`.

Usage

```
sleep_int2Brown(
  dataset,
  Interval.colname = Interval,
  Sleep.colname = State,
  wake.state = "wake",
  sleep.state = "sleep",
  Brown.day = "day",
  Brown.evening = "evening",
  Brown.night = "night",
  evening.length = lubridate::dhours(3),
  Brown.state.colname = State.Brown,
  output.dataset = TRUE
)
```

Arguments

`dataset` A dataset with sleep/wake intervals.

`Interval.colname`
The name of the column with the intervals. Defaults to `Interval`.

`Sleep.colname` The name of the column with the sleep/wake states. Defaults to `State`.

`wake.state, sleep.state`
The names of the wake and sleep states in the `Sleep.colname`. Default to `"wake"` and `"sleep"`. Expected to be a `character` scalar and must be an exact match.

`Brown.day, Brown.evening, Brown.night`
The names of the Brown states that will be used. Defaults to `"day"`, `"evening"` and `"night"`.

`evening.length` The length of the evening interval in seconds. Can also use **lubridate** duration or period objects. Defaults to 3 hours.

`Brown.state.colname`
The name of the column with the newly created Brown states. Works as a simple renaming of the `Sleep.colname`.

`output.dataset` Whether to return the whole `dataset` or a `vector` with the Brown states.

Details

The function will filter out any non-sleep intervals that are shorter than the specified `evening.length`. This prevents problematic behaviour when the `evening.length` is longer than the `wake` intervals or, e.g., when the first state is sleep after midnight and there is a prior `NA` interval from midnight till sleep. This behavior might, however, result in problematic results for specialized experimental setups with ultra short wake/sleep cycles. The `sleep_int2Brown()` function would not be applicable in those cases anyways.

Value

A dataset with the Brown states or a vector with the Brown states. The Brown states are created in a new column with the name specified in `Brown.state.colname`. The dataset will have more rows than the original dataset, because the `wake` intervals are split into `day` and `evening` intervals.

References

https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.3001571

See Also

Other Brown: `Brown2reference()`, `Brown_check()`, `Brown_cut()`, `Brown_rec()`

Examples

```
#create a sample dataset
sample <- tibble::tibble(Datetime = c("2023-08-15 6:00:00",
                                      "2023-08-15 23:00:00",
                                      "2023-08-16 6:00:00",
                                      "2023-08-16 22:00:00",
                                      "2023-08-17 6:30:00",
                                      "2023-08-18 1:00:00"),
                         State = rep(c("wake", "sleep"), 3),
                         Id = "Participant")
#intervals from sample
sc2interval(sample)
#recoded intervals
sc2interval(sample) %>% sleep_int2Brown()
```

`spectral_integration` *Integrate spectral irradiance with optional weighting*

Description

Integrates over a given spectrum, optionally over only a portion of the spectrum, optionally with a weighing function. Can be used to calculate spectral contributions in certain wavelength ranges, or to calculate (alphaopically equivalent daylight) illuminance.

Usage

```
spectral_integration(
  spectrum,
  wavelength.range = NULL,
  action.spectrum = NULL,
  general.weight = 1
)
```

Arguments

`spectrum` Tibble with spectral data (1st col: wavelength, 2nd col: SPD values)

`wavelength.range` Optional integration bounds (length-2 numeric)

`action.spectrum` Either:

- Tibble with wavelength and weighting columns
- Name of built-in spectrum: "photopic", "melanopic", "rhodopic", "l_cone_opic", "m_cone_opic", "s_cone_opic"

`general.weight` Scalar multiplier or "auto" for built-in efficacies

Details

The function uses trapezoidal integration and recognizes differing step-widths in the spectrum. If an action spectrum is used, values of the action spectrum at the spectral wavelenghts are interpolated with `stats::approx()`.

The used efficacies for for the auto-weighting are:

- photopic: 683.0015478
- melanopic: 1/0.0013262
- rhodopic: 1/0.0014497
- l_cone_opic: 1/0.0016289
- m_cone_opic: 1/0.0014558
- s_cone_opic: 1/0.0008173

This requires input values in W/(m^2) for the spectrum. If it is provided in other units, the result has to be rescaled afterwards.

Value

Numeric integrated value

See Also

Other Spectrum: `normalize_counts()`, `spectral_reconstruction()`

Examples

```
# creating an equal energy spectrum of value 1
spd <- data.frame(wl = 380:780, values = 1)

#integrating over the full spectrum
spectral_integration(spd)

#integrating over wavelengths 400-500 nm
spectral_integration(spd, wavelength.range = c(400, 500))

#calculating the photopic illuminance of an equal energy spectrum with 1 W/(m^2*nm)
spectral_integration(spd, action.spectrum = "photopic", general.weight = "auto")

#calculating the melanopic EDI of an equal energy spectrum with 1 W/(m^2*nm)
spectral_integration(spd, action.spectrum = "melanopic", general.weight = "auto")

# Custom action spectrum
custom_act <- data.frame(wavelength = 400:700, weight = 0.5)
spectral_integration(spd, wavelength.range = c(400,700),
                     action.spectrum = custom_act, general.weight = 2)

#using a spectrum that is broader then the action spectrum will not change the
#output, as the action spectrum will use zeros beyond its range
```

`spectral_reconstruction`

Reconstruct spectral irradiance from sensor counts

Description

This function takes sensor data in the form of (normalized) counts and reconstructs a spectral power distribution (SPD) through a calibration matrix. The matrix takes the form of `sensor channel x wavelength`, and the spectrum results form a linear combination of `counts x calibration-value` for any wavelength in the matrix. Handles multiple sensor readings by returning a list of spectra

Usage

```
spectral_reconstruction(
  sensor_channels,
  calibration_matrix,
  format = c("long", "wide")
)
```

Arguments

`sensor_channels`
: Named numeric vector or dataframe with sensor readings. Names must match calibration matrix columns.

`calibration_matrix`
: Matrix or dataframe with sensor-named columns and wavelength-indexed rows

`format`
: Output format: "long" (list of tibbles) or "wide" (dataframe)

Details

Please note that calibration matrices are not provided by LightLogR, but can be provided by a wearable device manufacturer. Counts can be normalized with the `normalize_counts()` function, provided that the output also contains a `gain` column.

Value

- "long": List of tibbles (wavelength, irradiance)
- "wide": Dataframe with wavelength columns and one row per spectrum

See Also

Other Spectrum: `normalize_counts()`, `spectral_integration()`

Examples

```
# Calibration matrix example
calib <- matrix(1:12, ncol=3, dimnames = list(400:403, c("R", "G", "B")))

# Named vector input
spectral_reconstruction(c(R=1, G=2, B=3), calib)

# Dataframe input
df <- data.frame(R=1, G=2, B=3, other_col=10)
spectral_reconstruction(dplyr::select(df, R:B), calib)

# Multiple spectra: as list columns
df <- data.frame(Measurement = c(1,2), R=c(1,2), G=c(2,4), B=c(3,6))
df <-
df |>
  dplyr::mutate(
      Spectrum = spectral_reconstruction(dplyr::pick(R:B), calib)
      )
df |> tidyr::unnest(Spectrum)

# Multiple spectra: as extended dataframes
df |>
  dplyr::mutate(
      Spectrum = spectral_reconstruction(dplyr::pick(R:B), calib, "wide"))
```

summarize_numeric *Summarize numeric columns in dataframes to means*

Description

This simple helper function was created to summarize episodes of gaps, clusters, or states, focusing on numeric variables. It calculates mean values for all numeric columns and handles Duration objects appropriately.

Despite its name, the function actually summarizes all double columns, which is more inclusive compared to just numeric columns.

Usage

```
summarize_numeric(
  data,
  remove = NULL,
  prefix = "mean_",
  na.rm = TRUE,
  complete.groups.on = NULL,
  add.total.duration = TRUE,
  durations.dec = 0,
  Datetime2Time = TRUE
)

summarise_numeric(
  data,
  remove = NULL,
  prefix = "mean_",
  na.rm = TRUE,
  complete.groups.on = NULL,
  add.total.duration = TRUE,
  durations.dec = 0,
  Datetime2Time = TRUE
)
```

Arguments

`data` A dataframe containing numeric data, typically from `extract_clusters()` or `extract_gaps()`.

`remove` Character vector of columns removed from the summary.

`prefix` A prefix to add to the column names of summarized metrics. Defaults to "mean_".

`na.rm` Whether to remove NA values when calculating means. Defaults to TRUE.

`complete.groups.on`
Column name that, together with grouping variables, can be used to provide a complete set. For example, with `extract_clusters()`, some days might

not have clusters. They do not show up in the summary output then. If it is important however, to consider that there are zero instances, one could extract the complete set of clusters and non-clusters, and then set `is.cluster` in this argument, which would then show zero clusters for those days.

`add.total.duration`
: Logical, whether the total duration for a given group should be calculated. Only relevant if a column `duration` is part of the input data.

`durations.dec`
: Numeric of number of decimals for the mean calculation of durations and times. Defaults to 0.

`Datetime2Time`
: Logical of whether POSIXct columns should be transformed into hms(time) columns, which is usually sensible for averaging (default is `TRUE`). Calls `Datetime2Time()` with default settings (all POSIXct are affected).

Value

A dataframe containing the summarized metrics.

Examples

```
# Extract clusters and summarize them
dataset <-
sample.data.environment %>%
aggregate_Datetime(unit = "15 mins") |>
extract_clusters(MEDI > 1000)

#input to summarize_numeric
dataset |> utils::head()
#output of summarize_numeric (removing state.count and epoch from the summary)
dataset |> summarize_numeric(c("state.count", "epoch"))
```

`supported_devices` *Get all the supported devices in LightLogR*

Description

Returns a vector of all the supported devices in LightLogR.

Usage

```
supported_devices()
```

Details

These are all supported devices where there is a dedicated import function. Import functions can be called either through `import_Dataset()` with the respective `device = "device"` argument, or directly, e.g., `import$ActLumus()`.

Value

A character vector of all supported devices

See Also

import_Dataset

Examples

```
supported_devices()
```

`symlog_trans` *Scale positive and negative values on a log scale*

Description

To create a plot with positive and negative (unscaled) values on a log-transformed axis, the values need to be scaled accordingly. R or **ggplot2** do not have a built-in function for this, but the following function can be used to create a transformation function for this purpose. The function was coded based on a post on stack overflow. The `symlog` transformation is the standard transformation used e.g., in `gg_day()`.

Usage

```
symlog_trans(base = 10, thr = 1, scale = 1)
```

Arguments

`base`	Base for the logarithmic transformation. The default is 10.
`thr`	Threshold after which a logarithmic transformation is applied. If the absolute value is below this `threshold`, the value is not transformed. The default is 1.
`scale`	Scaling factor for logarithmically transformed values above the `threshold`. The default is 1.

Details

The `symlog` transformation can be accessed either via the `trans = "symlog"` argument in a scaling function, or via `trans = symlog_trans()`. The latter allows setting the individual arguments.

Value

a transformation function that can be used in **ggplot2** or **plotly** to scale positive and negative values on a log scale.

References

This function‘s code is a straight copy from a post on stack overflow. The author of the answer is Julius Vainora, and the author of the question Brian B

Examples

```
dataset <-
sample.data.environment %>%
filter_Date(end = "2023-08-29") %>%
dplyr::mutate(MEDI = dplyr::case_when(
                                      Id == "Environment" ~ -MEDI,
                                      .default = MEDI))
#basic application where transformation, breaks and labels are set manually
dataset %>%
gg_day(aes_col = Id) +
ggplot2::scale_y_continuous(
trans = "symlog")

#the same plot, but with breaks and labels set manually
dataset %>%
gg_day(aes_col = Id) +
ggplot2::scale_y_continuous(
trans = "symlog",
breaks = c(-10^(5:0), 0, 10^(0:5)),
labels = function(x) format(x, scientific = FALSE, big.mark = " "))

#setting individual arguments of the symlog function manually allows
#e.g., to emphasize values smaller than 1
dataset %>%
gg_day(aes_col = Id) +
ggplot2::scale_y_continuous(
trans = symlog_trans(thr = 0.01),
breaks = c(-10^(5:-1), 0, 10^(-1:5)),
labels = function(x) format(x, scientific = FALSE, big.mark = " "))
```

---

threshold_for_duration

Find threshold for given duration

Description

This function finds the threshold for which light levels are above/below for a given duration. This function can be considered as the inverse of duration_above_threshold.

Usage

```
threshold_for_duration(
  Light.vector,
  Time.vector,
  duration,
  comparison = c("above", "below"),
  epoch = "dominant.epoch",
  na.rm = FALSE,
```

```
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`duration`	The duration for which the threshold should be found. Can be either a duration or a string. If it is a string, it needs to be a valid duration string, e.g., `"1 day"` or `"10 sec"`.
`comparison`	String specifying whether light levels above or below the threshold should be considered. Can be either `"above"` (the default) or `"below"`.
`epoch`	The epoch at which the data was sampled. Can be either a duration or a string. If it is a string, it needs to be either `"dominant.epoch"` (the default) for a guess based on the data, or a valid duration string, e.g., `"1 day"` or `"10 sec"`.
`na.rm`	Logical. Should missing values (NA) be removed for the calculation? Defaults to `FALSE`.
`as.df`	Logical. Should a data frame with be returned? If `TRUE`, a data frame with a single column named `threshold_{comparison}_for_{duration}` will be returned. Defaults to `FALSE`.

Value

Single numeric value or single column data frame.

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `timing_above_threshold()`

Examples

```
N <- 60
# Dataset with 30 min < 250lx and 30min > 250lx
dataset1 <-
  tibble::tibble(
    Id = rep("A", N),
    Datetime = lubridate::as_datetime(0) + lubridate::minutes(1:N),
    MEDI = sample(c(sample(1:249, N / 2, replace = TRUE),
                    sample(250:1000, N / 2, replace = TRUE))),
  )

dataset1 %>%
  dplyr::reframe("Threshold above which for 30 mins" =
                   threshold_for_duration(MEDI, Datetime, duration = "30 mins"))

dataset1 %>%
```

```
  dplyr::reframe("Threshold below which for 30 mins" =
                   threshold_for_duration(MEDI, Datetime, duration = "30 mins",
                                          comparison = "below"))

dataset1 %>%
  dplyr::reframe(threshold_for_duration(MEDI, Datetime, duration = "30 mins",
                                        as.df = TRUE))
```

`timing_above_threshold` *Mean/first/last timing above/below threshold.*

Description

This function calculates the mean, first, and last timepoint (MLiT, FLiT, LLiT) where light levels are above or below a given threshold intensity within the given time interval.

Usage

```
timing_above_threshold(
  Light.vector,
  Time.vector,
  comparison = c("above", "below"),
  threshold,
  na.rm = FALSE,
  as.df = FALSE
)
```

Arguments

`Light.vector`	Numeric vector containing the light data.
`Time.vector`	Vector containing the time data. Can be POSIXct, hms, duration, or difftime.
`comparison`	String specifying whether the time above or below threshold should be calculated. Can be either `"above"` (the default) or `"below"`. If two values are provided for `threshold`, this argument will be ignored.
`threshold`	Single numeric value or two numeric values specifying the threshold light level(s) to compare with. If a vector with two values is provided, the timing corresponding to light levels between the two thresholds will be calculated.
`na.rm`	Logical. Should missing values be removed for the calculation? Defaults to `FALSE`.
`as.df`	Logical. Should a data frame be returned? If `TRUE`, a data frame with three columns (MLiT, FLiT, LLiT) and the threshold (e.g., `MLiT_{threshold}`) will be returned. Defaults to `FALSE`.

Value

List or dataframe with the three values: `mean`, `first`, and `last` timing above threshold. The output type corresponds to the type of `Time.vector`, e.g., if `Time.vector` is HMS, the timing metrics will be also HMS, and vice versa for POSIXct and numeric.

References

Reid, K. J., Santostasi, G., Baron, K. G., Wilson, J., Kang, J., & Zee, P. C. (2014). Timing and Intensity of Light Correlate with Body Weight in Adults. *PLOS ONE*, 9(4), e92251. doi:10.1371/journal.pone.0092251

Hartmeyer, S.L., Andersen, M. (2023). Towards a framework for light-dosimetry studies: Quantification metrics. *Lighting Research & Technology*. doi:10.1177/14771535231170500

See Also

Other metrics: `bright_dark_period()`, `centroidLE()`, `disparity_index()`, `dose()`, `duration_above_threshold()`, `exponential_moving_average()`, `frequency_crossing_threshold()`, `interdaily_stability()`, `intradaily_variability()`, `midpointCE()`, `nvRC()`, `nvRD()`, `nvRD_cumulative_response()`, `period_above_threshold()`, `pulses_above_threshold()`, `threshold_for_duration()`

Examples

```
# Dataset with light > 250lx between 06:00 and 18:00
dataset1 <-
  tibble::tibble(
    Id = rep("A", 24),
    Datetime = lubridate::as_datetime(0) + lubridate::hours(0:23),
    MEDI = c(rep(1, 6), rep(250, 13), rep(1, 5))
  )

# Above threshold
dataset1 %>%
  dplyr::reframe(timing_above_threshold(MEDI, Datetime, "above", 250, as.df = TRUE))

# Below threshold
dataset1 %>%
  dplyr::reframe(timing_above_threshold(MEDI, Datetime, "below", 10, as.df = TRUE))

# Input = HMS -> Output = HMS
dataset1 %>%
 dplyr::reframe(timing_above_threshold(MEDI, hms::as_hms(Datetime), "above", 250, as.df = TRUE))
```

Index