Package ‘eyeris’

July 26, 2025

Type Package

Title Flexible, Extensible, & Reproducible Pupillometry Preprocessing

Version 2.1.1

Date 2025-07-25

Description Pupillometry offers a non-invasive window into the mind and has been used extensively as a psychophysiological readout of arousal signals linked with cognitive processes like attention, stress, and emotional states [Clewett et al. (2020) <doi:10.1038/s41467-020-17851-9>; Kret & Sjak-Shie (2018) <doi:10.3758/s13428-018-1075-y>; Strauch (2024) <doi:10.1016/j.tins.2024.06.002>]. Yet, despite decades of pupillometry research, many established packages and workflows to date lack design patterns based on Findability, Accessibility, Interoperability, and Reusability (FAIR) principles [see Wilkinson et al. (2016) <doi:10.1038/sdata.2016.18>]. 'eyeris' provides a modular, performant, and extensible preprocessing framework for pupillometry data with BIDS-like organization and interactive output reports [Esteban et al. (2019) <doi:10.1038/s41592-018-0235-4>; Gorgolewski et al. (2016) <doi:10.1038/sdata.2016.44>]. Development was supported, in part, by the Stanford Wu Tsai Human Performance Alliance, Stanford Ric Weiland Graduate Fellowship, Stanford Center for Mind, Brain, Computation and Technology, NIH National Institute on Aging Grants (R01-AG065255, R01-AG079345), NSF GRFP (DGE-2146755), McKnight Brain Research Foundation Clinical Translational Research Scholarship in Cognitive Aging and Age-Related Memory Loss, American Brain Foundation, and the American Academy of Neurology.

Encoding UTF-8

Depends R (>= 4.1)

Imports eyelinker, dplyr, gsignal, purrr, zoo, cli, rlang, stringr, utils, stats, graphics, grDevices, tidyr, progress, data.table, withr, lifecycle, MASS, viridis, fields, jsonlite, rmarkdown

RoxygenNote 7.3.2

Suggests knitr, testthat (>= 3.0.0), devtools

VignetteBuilder knitr

License MIT + file LICENSE

Config/testthat/edition 3

URL https://shawnschwartz.com/eyeris/,

https://github.com/shawntz/eyeris/

BugReports https://github.com/shawntz/eyeris/issues

NeedsCompilation no

Author Shawn Schwartz [aut, cre] (ORCID:
<https://orcid.org/0000-0001-6444-8451>),
Mingjian He [ctb],
Haopei Yang [ctb],
Alice Xue [ctb],
Gustavo Santiago-Reyes [ctb]

Maintainer Shawn Schwartz <shawn.t.schwartz@gmail.com>

Repository CRAN

Date/Publication 2025-07-26 16:10:02 UTC

Contents

`bidsify` *Save out pupil timeseries data in a BIDS-like structure*

Description

This method provides a structured way to save out pupil data in a BIDS-like structure. The method saves out epoched data as well as the raw pupil timeseries, and formats the directory and filename structures based on the metadata you provide.

Usage

```
bidsify(
  eyeris,
  save_all = TRUE,
  epochs_list = NULL,
  merge_epochs = FALSE,
  bids_dir = NULL,
  participant_id = NULL,
  session_num = NULL,
  task_name = NULL,
  run_num = NULL,
  merge_runs = FALSE,
  save_raw = TRUE,
  html_report = TRUE,
  report_seed = 0,
  report_epoch_grouping_var_col = "matched_event",
  verbose = TRUE,
  pdf_report = deprecated()
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`save_all`	Logical flag indicating whether all epochs are to be saved or only a subset of them. Defaults to `TRUE`
`epochs_list`	List of epochs to be saved. Defaults to `NULL`
`merge_epochs`	Logical flag indicating whether epochs should be saved as one file or as separate files. Defaults to `FALSE` (no merge)
`bids_dir`	Base bids_directory. Defaults to `NULL`
`participant_id`	BIDS subject ID. Defaults to `NULL`
`session_num`	BIDS session ID. Defaults to `NULL`
`task_name`	BIDS task ID. Defaults to `NULL`

`run_num`	BIDS run ID. Optional override for the run number when there's only one block of data present in a given `.asc` file. This allows you to manually specify a run number (e.g., "03") instead of using the default block number in `.asc` files (1). This is especially useful if you have a single `.asc` file for a single run of a task and want your BIDSified derivatives to be labeled correctly. However, for files with multiple recording blocks embedded within the **same** `.asc` file, this parameter is ignored and blocks are automatically numbered as runs (block 1 = run-01, block 2 = run-02, etc.) in the order they appeared/were recorded. Defaults to `NULL` (no override)
`merge_runs`	Logical flag indicating whether multiple runs (either from multiple recording blocks existing within the **same** `.asc` file (see above), or manually specified) should be combined into a single output file. When `TRUE`, adds a 'run' column to identify the source run Defaults to `FALSE` (i.e., separate files per block/run – the standard BIDS-like-behavior)
`save_raw`	Logical flag indicating whether to save_raw pupil data in addition to epoched data. Defaults to `TRUE`
`html_report`	Logical flag indicating whether to save out the `eyeris` preprocessing summary report as an HTML file. Defaults to `TRUE`
`report_seed`	Random seed for the plots that will appear in the report Defaults to `0`. See `plot()` for a more detailed description
`report_epoch_grouping_var_col`	String name of grouping column to use for epoch-by-epoch diagnostic plots in an interactive rendered HTML report. Column name must exist (i.e., be a custom grouping variable name set within the metadata template of your `epoch()` call). Defaults to `"matched_event"`, which all epoched dataframes have as a valid column name. To disable these epoch-level diagnostic plots, set to `NULL`
`verbose`	A flag to indicate whether to print detailed logging messages. Defaults to `TRUE`. Set to `FALSE` to suppress messages about the current processing step and run silently
`pdf_report`	**(Deprecated)** Use `html_report = TRUE` instead

Details

In the future, we intend for this function to save out the data in an official BIDS format for eyetracking data (see the proposal currently under review here). At this time, however, this function instead takes a more BIDS-inspired approach to organizing the output files for preprocessed pupil data.

Value

Invisibly returns `NULL`. Called for its side effects

See Also

`lifecycle::deprecate_warn()`

Examples

```
# Bleed around blink periods just long enough to remove majority of
#  deflections due to eyelid movements

demo_data <- eyelink_asc_demo_dataset()

# example with unepoched data
demo_data |>
  eyeris::glassbox() |>
  eyeris::bidsify(
    bids_dir = tempdir(), # <- MAKE SURE TO UPDATE TO YOUR DESIRED LOCAL PATH
    participant_id = "001",
    session_num = "01",
    task_name = "assocret",
    run_num = "01",
    save_raw = TRUE, # save out raw timeseries
    html_report = TRUE, # generate interactive report document
    report_seed = 0 # make randomly selected plot epochs reproducible
  )

# example with epoched data
demo_data |>
  eyeris::glassbox() |>
  eyeris::epoch(
    events = "PROBE_{startstop}_{trial}",
    limits = c(-1, 1), # grab 1 second prior to and 1 second post event
    label = "prePostProbe" # custom epoch label name
  ) |>
  eyeris::bidsify(
    bids_dir = tempdir(), # <- MAKE SURE TO UPDATE TO YOUR DESIRED LOCAL PATH
    participant_id = "001",
    session_num = "01",
    task_name = "assocret",
    run_num = "01"
  )

# example with run_num for single block data
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox() |>
  eyeris::epoch(
    events = "PROBE_{startstop}_{trial}",
    limits = c(-1, 1),
    label = "prePostProbe"
  ) |>
  eyeris::bidsify(
    bids_dir = tempdir(),
    participant_id = "001",
    session_num = "01",
    task_name = "assocret",
    run_num = "03" # override default run-01 (block_1) to use run-03 instead
```

```
)
```

---

bin — *Bin pupil time series by averaging within time bins*

Description

This function bins pupillometry data by dividing time into equal intervals and averaging the data within each bin. Unlike downsampling, binning averages data points within each time bin.

Usage

```
bin(eyeris, bins_per_second, method = "mean", call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`bins_per_second`	The number of bins to create per second of data
`method`	The binning method: "mean" (default) or "median"
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call. Defaults to `NULL`

Details

Binning divides one second of pupillary data into X bins and averages pupillometry data around each bin center. The resulting time points will be: 1/2X, 3/2X, 5/2X, ..., etc. where X is the number of bins per second.

This approach is commonly used in pupillometry research to study temporal dynamics of pupil dilatory response; however, it should be used with caution (as averaging within bins can distort the pupillary dynamics).

Value

An `eyeris` object with binned data and updated sampling rate

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `bin = list(...)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline `downsample()` for downsampling functionality

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# bin data into 10 bins per second using the (default) "mean" method
demo_data |>
  eyeris::glassbox(bin = list(bins_per_second = 10, method = "mean")) |>
  plot(seed = 0)
```

`deblink` *NA-pad blink events / missing data*

Description

Deblinking (a.k.a. NA-padding) of time series data. The intended use of this method is to remove blink-related artifacts surrounding periods of missing data. For instance, when an individual blinks, there are usually rapid decreases followed by increases in pupil size, with a chunk of data missing in-between these 'spike'-looking events. The deblinking procedure here will NA-pad each missing data point by your specified number of ms.

Usage

```
deblink(eyeris, extend = 50, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`extend`	Either a single number indicating the number of milliseconds to pad forward/backward around each missing sample, or, a vector of length two indicating different numbers of milliseconds pad forward/backward around each missing sample, in the format `c(backward, forward)`
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call

Details

This function is automatically called by `glassbox()` by default. If needed, customize the parameters for `deblink` by providing a parameter list. Use `glassbox(deblink = FALSE)` to disable this step as needed.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Value

An `eyeris` object with a new column: `pupil_raw_{...}_deblink`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `deblink = list(...)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# 50 ms in both directions (the default)
demo_data |>
  eyeris::glassbox(deblink = list(extend = 50)) |>
  plot(seed = 0)

# 40 ms backward, 50 ms forward
demo_data |>
  # set deblink to FALSE (instead of a list of params)
  #  to skip step (not recommended)
  eyeris::glassbox(deblink = list(extend = c(40, 50))) |>
  plot(seed = 0)
```

`detransient` *Remove pupil samples that are physiologically unlikely*

Description

The intended use of this method is for removing pupil samples that emerge more quickly than would be physiologically expected. This is accomplished by rejecting samples that exceed a "speed"-based threshold (i.e., median absolute deviation from sample-to-sample). This threshold is computed based on the constant `n`, which defaults to the value `16`.

Usage

```
detransient(eyeris, n = 16, mad_thresh = NULL, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`n`	A constant used to compute the median absolute deviation (MAD) threshold. Defaults to `16`

`mad_thresh` Default `NULL`. This parameter provides alternative options for handling edge cases where the computed properties here within `detransient()` mad_val and $median\_speed$ are very small. For example, if

$$mad\_val = 0 \quad \text{and} \quad median\_speed = 1,$$

then, with the default multiplier $n = 16$,

$$mad\_thresh = median\_speed + (n \times mad\_val) = 1 + (16 \times 0) = 1.$$

In this situation, any speed $p_i \geq 1$ would be flagged as a transient, which might be overly sensitive. To reduce this sensitivity, two possible adjustments are available:

1. If $mad\_thresh = 1$, the transient detection criterion is modified from

   $$p_i \geq mad\_thresh$$

   to

   $$p_i > mad\_thresh.$$

2. If mad_thresh is very small, the user may manually adjust the sensitivity by supplying an alternative threshold value here directly via this `mad_thresh` parameter.

`call_info` A list of call information and parameters. If not provided, it will be generated from the function call. Defaults to `NULL`

Details

This function is automatically called by `glassbox()` by default. If needed, customize the parameters for `detransient` by providing a parameter list. Use `glassbox(detransient = FALSE)` to disable this step as needed.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Computed properties:

- `pupil_speed`**:** Compute speed of pupil by approximating the derivative of `x` (pupil) with respect to `y` (time) using finite differences.
  - Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ be two numeric vectors with $n \geq 2$; then, the finite differences are computed as:

    $$\delta_i = \frac{x_{i+1} - x_i}{y_{i+1} - y_i}, \quad i = 1, 2, \ldots, n-1.$$

  - This produces an output vector $p = (p_1, p_2, \ldots, p_n)$ defined by:
    - For the first element:

      $$p_1 = |\delta_1|,$$

    - For the last element:

      $$p_n = |\delta_{n-1}|,$$

  - $*$ For the intermediate elements ($i = 2, 3, \ldots, n-1$):

$$p_i = \max\{|\delta_{i-1}|, |\delta_i|\}.$$

- `median_speed`**:** The median of the computed `pupil_speed`:

$$median\_speed = median(p)$$

- `mad_val`**:** The median absolute deviation (MAD) of `pupil_speed` from the median:

$$mad\_val = median(|p - median\_speed|)$$

- `mad_thresh`**:** A threshold computed from the median speed and the MAD, using a constant multiplier n (default value: 16):

$$mad\_thresh = median\_speed + (n \times mad\_val)$$

Value

An `eyeris` object with a new column in `timeseries`: `pupil_raw_{...}_detransient`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `detransient = list(...)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(
    detransient = list(n = 16) # set to FALSE to skip step (not recommended)
  ) |>
  plot(seed = 0)
```

`detrend` *Detrend the pupil time series*

Description

Linearly detrend_pupil data by fitting a linear model of `pupil_data ~ time`, and return the fitted betas and the residuals (`pupil_data - fitted_values`).

Usage

```
detrend(eyeris, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call. Defaults to `NULL`

Details

This function is automatically called by `glassbox()` if `detrend = TRUE`.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Value

An `eyeris` object with two new columns in `timeseries`: `detrend_fitted_betas`, and `pupil_raw_{...}_detrend`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Use `glassbox(detrend = TRUE)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(detrend = TRUE) |>  # set to FALSE to skip step (default)
  plot(seed = 0)
```

`downsample` *Downsample pupil time series with anti-aliasing filtering*

Description

This function downsamples pupillometry data by applying an anti-aliasing filter before decimation. Unlike binning, downsampling preserves the original temporal dynamics without averaging within bins.

Usage

```
downsample(
  eyeris,
  target_fs,
  plot_freqz = FALSE,
  rp = 1,
  rs = 35,
  call_info = NULL
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`.
`target_fs`	The target sampling frequency in Hz after downsampling.
`plot_freqz`	Boolean flag for displaying filter frequency response (default FALSE).
`rp`	Passband ripple in dB (default 1).
`rs`	Stopband attenuation in dB (default 35).
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call.

Details

Downsampling reduces the sampling frequency by decimating data points. The function automatically designs an anti-aliasing filter using the `lpfilt()` function with carefully chosen parameters:

- `ws` (stopband frequency) = Fs_new / 2 (Nyquist freq of new sampling rate)
- `wp` (passband frequency) = ws - max(5, Fs_nq * 0.2)
- An error is raised if `wp` < 4 to prevent loss of pupillary responses

The resulting time points will be: 0, 1/X, 2/X, 3/X, ..., etc. where X is the new sampling frequency.

Value

An `eyeris` object with downsampled data and updated sampling rate.

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `downsample = list(...)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline. `bin()` for binning functionality.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# downsample pupil data recorded at 1000 Hz to 100 Hz with the default params
demo_data |>
  eyeris::glassbox(downsample = list(target_fs = 100)) |>
  plot(seed = 0)
```

`epoch` *Epoch (and baseline) pupil data based on custom event message structure*

Description

Intended to be used as the final preprocessing step. This function creates data epochs of either fixed or dynamic durations with respect to provided `events` and time `limits`, and also includes an intuitive metadata parsing feature where additional trial data embedded within event messages can easily be identified and joined into the resulting epoched data frames.

Usage

```
epoch(
  eyeris,
  events,
  limits = NULL,
  label = NULL,
  baseline = FALSE,
  baseline_type = c("sub", "div"),
  baseline_events = NULL,
  baseline_period = NULL,
  hz = NULL,
  verbose = TRUE,
  call_info = NULL,
  calc_baseline = deprecated(),
  apply_baseline = deprecated()
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`events`	Either (1) a single string representing the event message to perform trial extraction around, using specified `limits` to center the epoch around or no `limits` (which then just grabs the data epochs between each subsequent event string of the same type); (2) a vector containing both `start` and `end` event message strings – here, `limits` will be ignored and the duration of each trial epoch will be the number of samples between each matched `start` and `end` event message pair; or (3) a list of 2 dataframes that manually specify start/end event timestamp-message pairs to pull out of the raw timeseries data – here, it is required that each raw timestamp and event message be provided in the following format: list(data.frame(time = c(...), msg = c(...)), # start events data.frame(time = c(...), msg = c(...)), # end events 1 # block number) where the first data.frame indicates the `start` event timestamp and message string pairs, and the second data.frame indicates the `end` event timestamp and message string pairs. Additionally, manual epoching only words with 1 block at a time for event-modes `2` and `3`; thus, please be sure to explicitly indicate the block number in your input list (for examples, see above as well as example #9 below for more details) For event-modes `1` and `2`, the way in which you pass in the event message string must conform to a standardized protocol so that `eyeris` knows how to find your events and (optionally) parse any included metadata into the tidy epoch data outputs. You have two primary choices: either (a) specify a string followed by a `*` wildcard expression (e.g., `"PROBE_START*"`), which will match any messages that have "PROBE_START ..." (... referring to potential metadata, such as trial number, stim file, etc.); or (b) specify a string using the `eyeris` syntax: (e.g., `"PROBE_{type}_{trial}"`), which will match the messages that follow a structure like this "PROBE_START_1" and "PROBE_STOP_1", and generate two additional metadata columns: `type` and `trial`, which would contain the following values based on these two example strings: `type`: `('START', 'STOP')`, and `trial`: `(1, 1)`
`limits`	A vector of 2 values (start, end) in seconds, indicating where trial extraction should occur centered around any given `start` message string in the `events` parameter
`label`	An (optional) string you can provide to customize the name of the resulting `eyeris` class object containing the epoched data frame. If left as `NULL` (default), then list item will be called `epoch_xyz`, where `xyz` will be a sanitized version of the original `start` event string you provided for matching. If you choose to specify a `label` here, then the resulting list object name will take the form: `epoch_label`. **Warning: if no `label` is specified and there are no event message strings to sanitize, then you may obtain a strange-looking epoch list element in your output object (e.g., `$epoch_`, or `$epoch_nana`, etc.). The data should still be accessible within this nested lists, however, to avoid ambiguous list objects, we recommend you provide an `epoch` label here to be safe**

`baseline`	**(New)** A single parameter that controls baseline correction. Set to `TRUE` to both calculate and apply baseline correction, or `FALSE` to skip it. This replaces the deprecated `calc_baseline` and `apply_baseline` parameters
`baseline_type`	Whether to perform *subtractive* (`sub`) or *divisive* (`div`) baseline correction. Defaults to `sub`
`baseline_events`	Similar to `events`, `baseline_events`, you can supply either (1) a single string representing the event message to center the baseline calculation around, as indicated by `baseline_period`; or (2) a single vector containing both a `start` and an `end` event message string – here, `baseline_period` will be ignored and the duration of each baseline period that the mean will be calculated on will be the number of samples between each matched `start` and `end` event message pair, as opposed to a specified fixed duration (as described in 1). Please note, providing a list of trial-level start/end message pairs (like in the `events` parameter) to manually indicate unique start/end chunks for baselining is currently unsupported. Though, we intend to add this feature in a later version of `eyeris`, given it likely won't be a heavily utilized / in demand feature.
`baseline_period`	A vector of 2 values (start, end) in seconds, indicating the window of data that will be used to perform the baseline correction, which will be centered around the single string "start" message string provided in `baseline_events`. Again, `baseline_period` will be ignored if both a "start" **and** "end" message string are provided to the `baseline_events` argument
`hz`	Data sampling rate. If not specified, will use the value contained within the tracker's metadata
`verbose`	A flag to indicate whether to print detailed logging messages Defaults to `TRUE`. Set to `False` to suppress messages about the current processing step and run silently
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call
`calc_baseline`	**(Deprecated)** Use `baseline` instead
`apply_baseline`	**(Deprecated)** Use `baseline` instead

Value

An `eyeris` object with a new nested list of data frames: `$epoch_*`. The epochs are organized hierarchically by block and preprocessing step. Each epoch contains the pupil timeseries data for the specified time window around each event message, along with metadata about the event.

When using `bidsify()` to export the data, filenames will include both epoch and baseline event information for clarity.

See Also

`lifecycle::deprecate_warn()`

Examples

```
demo_data <- eyelink_asc_demo_dataset()
eye_preproc <- eyeris::glassbox(demo_data)

# example 1: select 1 second before/after matched event message "PROBE*"
eye_preproc |>
  eyeris::epoch(events = "PROBE*", limits = c(-1, 1))

# example 2: select all samples between each trial
eye_preproc |>
  eyeris::epoch(events = "TRIALID {trial}")

# example 3: grab the 1 second following probe onset
eye_preproc |>
  eyeris::epoch(
    events = "PROBE_START_{trial}",
    limits = c(0, 1)
  )

# example 4: 1 second prior to and 1 second after probe onset
eye_preproc |>
  eyeris::epoch(
    events = "PROBE_START_{trial}",
    limits = c(-1, 1),
    label = "prePostProbe" # custom epoch label name
  )

# example 5: manual start/end event pairs
# note: here, the `msg` column of each data frame is optional
eye_preproc |>
  eyeris::epoch(
    events = list(
      data.frame(time = c(11334491), msg = c("TRIALID 22")), # start events
      data.frame(time = c(11337158), msg = c("RESPONSE_22")), # end events
      1 # block number
    ),
    label = "example5"
  )

# example 6: manual start/end event pairs
# note: set `msg` to NA if you only want to pass in start/end timestamps
eye_preproc |>
  eyeris::epoch(
    events = list(
      data.frame(time = c(11334491), msg = NA), # start events
      data.frame(time = c(11337158), msg = NA), # end events
      1 # block number
    ),
    label = "example6"
  )

## examples with baseline arguments enabled
```

```
# example 7: use mean of 1-s preceding "PROBE_START" (i.e. "DELAY_STOP")
# to perform subtractive baselining of the 1-s PROBE epochs.
eye_preproc |>
  eyeris::epoch(
    events = "PROBE_START_{trial}",
    limits = c(0, 1), # grab 0 seconds prior to and 1 second post PROBE event
    label = "prePostProbe", # custom epoch label name
    baseline = TRUE, # Calculate and apply baseline correction
    baseline_type = "sub", # "sub"tractive baseline calculation is default
    baseline_events = "DELAY_STOP_*",
    baseline_period = c(-1, 0)
  )

# example 8: use mean of time period between set start/end event messages
# (i.e. between "DELAY_START" and "DELAY_STOP"). In this case, the
# `baseline_period` argument will be ignored since both a "start" and "end"
# message string are provided to the `baseline_events` argument.
eye_preproc |>
  eyeris::epoch(
    events = "PROBE_START_{trial}",
    limits = c(0, 1), # grab 0 seconds prior to and 1 second post PROBE event
    label = "prePostProbe", # custom epoch label name
    baseline = TRUE, # Calculate and apply baseline correction
    baseline_type = "sub", # "sub"tractive baseline calculation is default
    baseline_events = c(
      "DELAY_START_*",
      "DELAY_STOP_*"
    )
  )

# example 9: additional (potentially helpful) example
start_events <- data.frame(
  time = c(11334491, 11338691),
  msg = c("TRIALID 22", "TRIALID 23")
)
end_events <- data.frame(
  time = c(11337158, 11341292),
  msg = c("RESPONSE_22", "RESPONSE_23")
)
block_number <- 1

eye_preproc |>
  eyeris::epoch(
    events = list(start_events, end_events, block_number),
    label  = "example9"
  )
```

`eyelink_asc_binocular_demo_dataset` *Access example EyeLink .asc binocular mock dataset file provided by the eyeris package.*

Description

Returns the file path to the demo binocular `.asc` EyeLink pupil data file included in the `eyeris` package.

Usage

```
eyelink_asc_binocular_demo_dataset()
```

Details

This dataset is a mock dataset trimmed from a larger data file. The original data file was obtained from: https://github.com/scott-huberty/eyelinkio/blob/main/src/eyelinkio/tests/data/test_raw_binocular.edf

Value

A character string giving the full file path to the demo `.asc` EyeLink pupil data file

Examples

```
path_to_binocular_demo_dataset <- eyelink_asc_binocular_demo_dataset()
print(path_to_binocular_demo_dataset)
```

`eyelink_asc_demo_dataset` *Access example EyeLink .asc demo dataset file provided by the eyeris package.*

Description

Returns the file path to the demo `.asc` EyeLink pupil data file included in the `eyeris` package.

Usage

```
eyelink_asc_demo_dataset()
```

Value

A character string giving the full file path to the demo `.asc` EyeLink pupil data file

Examples

```
path_to_demo_dataset <- eyelink_asc_demo_dataset()
print(path_to_demo_dataset)
```

`eyelogger` *Run* `eyeris` *commands with automatic logging of R console's stdout and stderr*

Description

This utility function evaluates `eyeris` commands while automatically capturing and recording both standard output (`stdout`) and standard error (`stderr`) to timestamped log files in your desired log directory.

Usage

```
eyelogger(
  eyeris_cmd,
  log_dir = file.path(tempdir(), "eyeris_logs"),
  timestamp_format = "%Y%m%d_%H%M%S"
)
```

Arguments

`eyeris_cmd`	An `eyeris` command, wrapped in `{}` if multiline
`log_dir`	Character path to the desired log directory. Is set to the temporary directory given by `tempdir()` by default
`timestamp_format`	Format string passed to `format(Sys.time())` for naming the log files. Defaults to `"%Y%m%d_%H%M%S"`

Details

Each run produces two log files:

- `<timestamp>.out`: records all console output
- `<timestamp>.err`: records all warnings and errors

Value

The result of the evaluated `eyeris` command (invisibly)

Examples

```
eyelogger({
  message("eyeris `glassbox()` completed successfully.")
  warning("eyeris `glassbox()` completed with warnings.")
  print("some eyeris-related information.")
})

eyelogger({
  glassbox(eyelink_asc_demo_dataset(), interactive_preview = FALSE)
}, log_dir = file.path(tempdir(), "eyeris_logs"))
```

`eyeris_color_palette` *Default color palette for eyeris plotting functions*

Description

A custom color palette designed for visualizing pupil data preprocessing steps. This palette is based on the RColorBrewer Set1 palette and provides distinct, visually appealing colors for different preprocessing stages.

Usage

```
eyeris_color_palette()
```

Details

The palette includes 7 colors optimized for:

- High contrast and visibility
- Colorblind-friendly design
- Consistent visual hierarchy across preprocessing steps
- Professional appearance in reports and publications

Colors are designed to work well with both light and dark backgrounds and maintain readability when overlaid in time series plots.

Value

A character vector of 7 hex color codes representing the default eyeris color palette

Examples

```
# get the default color palette
colors <- eyeris_color_palette()
print(colors)

# use in a plot
plot(1:7, 1:7, col = colors, pch = 19, cex = 3)
```

`glassbox` *The opinionated "glass box"* `eyeris` *pipeline*

Description

This `glassbox` function (in contrast to a "black box" function where you run it and get a result but have no (or little) idea as to how you got from input to output) has a few primary benefits over calling each exported function from `eyeris` separately.

Usage

```
glassbox(
  file,
  interactive_preview = FALSE,
  preview_n = 3,
  preview_duration = 5,
  preview_window = NULL,
  verbose = TRUE,
  ...,
  confirm = deprecated(),
  num_previews = deprecated(),
  detrend_data = deprecated(),
  skip_detransient = deprecated()
)
```

Arguments

`file`	An SR Research EyeLink `.asc` file generated by the official EyeLink `edf2asc` command
`interactive_preview`	A flag to indicate whether to run the `glassbox` pipeline autonomously all the way through (set to `FALSE` by default), or to interactively provide a visualization after each pipeline step, where you must also indicate "(y)es" or "(n)o" to either proceed or cancel the current `glassbox` pipeline operation (set to `TRUE`)
`preview_n`	Number of random example "epochs" to generate for previewing the effect of each preprocessing step on the pupil timeseries
`preview_duration`	Time in seconds of each randomly selected preview
`preview_window`	The start and stop raw timestamps used to subset the preprocessed data from each step of the `eyeris` workflow for visualization. Defaults to NULL, meaning random epochs as defined by `preview_n` and `preview_duration` will be plotted. To override the random epochs, set `preview_window` here to a vector with relative start and stop times (in seconds), for example – `c(5,6)` – to indicate the raw data from 5-6 secs on data that were recorded at 1000 Hz). Note, the start/stop time values indicated here are in seconds because `eyeris` automatically computes the indices for the supplied range of seconds using the `$info$sample.rate` metadata in the `eyeris` S3 class object

`verbose`	A logical flag to indicate whether to print status messages to the console. Defaults to `TRUE`. Set to `FALSE` to suppress messages about the current processing step and run silently
`...`	Additional arguments to override the default, prescribed settings
`confirm`	**(Deprecated)** Use `interactive_preview` instead
`num_previews`	**(Deprecated)** Use `preview_n` instead
`detrend_data`	**(Deprecated)** A flag to indicate whether to run the `detrend` step (set to `FALSE` by default). Detrending your pupil timeseries can have unintended consequences; we thus recommend that users understand the implications of detrending – in addition to whether detrending is appropriate for the research design and question(s) – before using this function
`skip_detransient`	**(Deprecated)** A flag to indicate whether to skip the `detransient` step (set to `FALSE` by default). In most cases, this should remain `FALSE`. For a more detailed description about likely edge cases that would prompt you to set this to `TRUE`, see the docs for `detransient()`

Details

First, this `glassbox` function provides a highly opinionated prescription of steps and starting parameters we believe any pupillometry researcher should use as their defaults when preprocessing pupillometry data.

Second, and not mutually exclusive from the first point, using this function should ideally reduce the probability of accidental mishaps when "reimplementing" the steps from the preprocessing pipeline both within and across projects. We hope to streamline the process in such a way that you could collect a pupillometry dataset and within a few minutes assess the quality of those data while simultaneously running a full preprocessing pipeline in 1-ish line of code!

Third, `glassbox` provides an "interactive" framework where you can evaluate the consequences of the parameters within each step on your data in real time, facilitating a fairly easy-to-use workflow for parameter optimization on your particular dataset. This process essentially takes each of the opinionated steps and provides a pre-/post-plot of the timeseries data for each step so you can adjust parameters and re-run the pipeline until you are satisfied with the choices of your parameters and their consequences on your pupil timeseries data.

Value

Preprocessed pupil data contained within an object of class `eyeris`

See Also

`lifecycle::deprecate_warn()`

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# (1) examples using the default prescribed parameters and pipeline recipe
```

```
## (a) run an automated pipeline with no real-time inspection of parameters
output <- eyeris::glassbox(demo_data)

start_time <- min(output$timeseries$block_1$time_secs)
end_time <- max(output$timeseries$block_1$time_secs)

# by default, verbose = TRUE. To suppress messages, set verbose = FALSE.
plot(
  output,
  steps = c(1, 5),
  preview_window = c(start_time, end_time),
  seed = 0
)

## (b) run a interactive workflow (with confirmation prompts after each step)

output <- eyeris::glassbox(demo_data, interactive_preview = TRUE, seed = 0)


# (2) examples of overriding the default parameters
output <- eyeris::glassbox(
  demo_data,
  interactive_preview = FALSE, # TRUE to visualize each step in real-time
  deblink = list(extend = 40),
  lpfilt = list(plot_freqz = TRUE) # overrides verbose parameter
)

# to suppress messages, set verbose = FALSE in plot():
plot(output, seed = 0, verbose = FALSE)

# (3) examples of disabling certain steps
output <- eyeris::glassbox(
  demo_data,
  detransient = FALSE,
  detrend = FALSE,
  zscore = FALSE
)

plot(output, seed = 0)
```

`interpolate` *Interpolate missing pupil samples*

Description

Linear interpolation of time series data. The intended use of this method is for filling in missing pupil samples (NAs) in the time series. This method uses "na.approx()" function from the zoo package, which implements linear interpolation using the "approx()" function from the stats package. Currently, NAs at the beginning and the end of the data are replaced with values on either end, respectively, using the "rule = 2" argument in the `approx()` function.

Usage

```
interpolate(eyeris, verbose = TRUE, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`verbose`	A flag to indicate whether to print detailed logging messages. Defaults to `TRUE`. Set to `FALSE` to suppress messages about the current processing step and run silently
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call

Details

This function is automatically called by `glassbox()` by default. Use `glassbox(interpolate = FALSE)` to disable this step as needed.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Value

An `eyeris` object with a new column in `timeseries`: `pupil_raw_{...}_interpolate`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Use `glassbox(interpolate = TRUE)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  # set to FALSE to skip (not recommended)
  eyeris::glassbox(interpolate = TRUE) |>
  plot(seed = 0)
```

`load_asc` *Load and parse SR Research EyeLink `.asc` files*

Description

This function builds upon the `eyelinker::read.asc()` function to parse the messages and metadata within the EyeLink `.asc` file. After loading and additional processing, this function returns an S3 `eyeris` class for use in all subsequent `eyeris` pipeline steps and functions.

Usage

```
load_asc(
  file,
  block = "auto",
  binocular_mode = c("average", "left", "right", "both")
)
```

Arguments

`file`	An SR Research EyeLink `.asc` file generated by the official EyeLink `edf2asc` command
`block`	Optional block number specification. The following are valid options: • "auto" (default): Automatically handles multiple recording segments embedded within the same `.asc` file. We recommend using this default as this is likely the safer choice then assuming a single-block recording (unless you know what you're doing). • `NULL`: Omits block column. Suitable for single-block recordings. • Numeric value: Manually sets block number based on the value provided here.
`binocular_mode`	Optional binocular mode specification. The following are valid options: • "average" (default): Averages the left and right eye pupil sizes. • "left": Uses only the left eye pupil size. • "right": Uses only the right eye pupil size. • "both": Uses both the left and right eye pupil sizes independently.

Details

This function is automatically called by `glassbox()` by default. If needed, customize the parameters for `load_asc` by providing a parameter list.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Value

An object of S3 class `eyeris` with the following attributes:

1. `file`: Path to the original `.asc` file.
2. `timeseries`: Data frame of all raw time series data from the tracker.
3. `events`: Data frame of all event messages and their time stamps.
4. `blinks`: Data frame of all blink events.
5. `info`: Data frame of various metadata parsed from the file header.
6. `latest`: `eyeris` variable for tracking pipeline run history.

For binocular data with `binocular_mode = "both"`, returns a list containing:

1. `left`: An `eyeris` object for the left eye data.
2. `right`: An `eyeris` object for the right eye data.
3. `original_file`: Path to the original `.asc` file.

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `load_asc = list(...)`.

Advanced users may call it directly if needed.

See Also

`eyelinker::read.asc()` which this function wraps.

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(load_asc = list(block = 1))

# Other useful parameter configurations
## (1) Basic usage (no block column specified)
demo_data |>
  eyeris::load_asc()

## (2) Manual specification of block number
demo_data |>
  eyeris::load_asc(block = 3)

## (3) Auto-detect multiple recording segments embedded within the same
##  file (i.e., the default behavior)
demo_data |>
  eyeris::load_asc(block = "auto")
```

```
## (4) Omit block column
demo_data |>
  eyeris::load_asc(block = NULL)
```

`lpfilt` *Lowpass filtering of time series data*

Description

The intended use of this method is for smoothing, although by specifying `wp` and `ws` differently one can achieve highpass or bandpass filtering as well. However, only lowpass filtering should be done on pupillometry data.

Usage

```
lpfilt(
  eyeris,
  wp = 4,
  ws = 8,
  rp = 1,
  rs = 35,
  plot_freqz = FALSE,
  call_info = NULL
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`wp`	The end of passband frequency in Hz (desired lowpass cutoff). Defaults to `4`
`ws`	The start of stopband frequency in Hz (required lowpass cutoff). Defaults to `8`
`rp`	Required maximal ripple within passband in dB. Defaults to `1`
`rs`	Required minimal attenuation within stopband in dB. Defaults to `35`
`plot_freqz`	A flag to indicate whether to display the filter frequency response. Defaults to `FALSE`
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call. Defaults to `NULL`

Details

This function is automatically called by `glassbox()` by default. If needed, customize the parameters for `lpfilt` by providing a parameter list. Use `glassbox(lpfilt = FALSE)` to disable this step as needed.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

Value

An `eyeris` object with a new column in `timeseries`: `pupil_raw_{...}_lpfilt`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Provide parameters via `lpfilt = list(...)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  # set lpfilt to FALSE (instead of a list of params) to skip step
  eyeris::glassbox(lpfilt = list(plot_freqz = TRUE)) |>
  plot(seed = 0)
```

`pipeline_handler` *Build a generic operation (extension) for the* `eyeris` *pipeline*

Description

`pipeline_handler` enables flexible integration of custom data processing functions into the `eyeris` pipeline. Under the hood, each preprocessing function in `eyeris` is a wrapper around a core operation that gets tracked, versioned, and stored using this `pipeline_handler` method. As such, custom pipeline steps must conform to the `eyeris` protocol for maximum compatibility with the downstream functions we provide.

Usage

```
pipeline_handler(eyeris, operation, new_suffix, ...)
```

Arguments

`eyeris`	An object of class `eyeris` containing timeseries data in a list of dataframes (one per block), various metadata collected by the tracker, and `eyeris` specific pointers for tracking the preprocessing history for that specific instance of the `eyeris` object

`operation`	The name of the function to apply to the timeseries data. This custom function should accept a dataframe `x`, a string `prev_op` (i.e., the name of the previous pupil column – which you DO NOT need to supply as a literal string as this is inferred from the `latest` pointer within the `eyeris` object), and any custom parameters you would like
`new_suffix`	A character string indicating the suffix you would like to be appended to the name of the previous operation's column, which will be used for the new column name in the updated preprocessed dataframe(s)
`...`	Additional (optional) arguments passed to the `operation` method

Details

Following the `eyeris` protocol also ensures:

- all operations follow a predictable structure, and
- that new pupil data columns based on previous operations in the chain are able to be dynamically constructed within the core timeseries data frame.

Value

An updated `eyeris` object with the new column added to the `timeseries` dataframe and the `latest` pointer updated to the name of the most recently added column plus all previous columns (ie, the history "trace" of preprocessing steps from start-to-present)

See Also

For more details, please check out the following vignettes:

- Anatomy of an eyeris Object

`vignette("anatomy", package = "eyeris")`

- Building Your Own Custom Pipeline Extensions

`vignette("custom-extensions", package = "eyeris")`

Examples

```
# first, define your custom data preprocessing function
winsorize_pupil <- function(x, prev_op, lower = 0.01, upper = 0.99) {
  vec <- x[[prev_op]]
  q <- quantile(vec, probs = c(lower, upper), na.rm = TRUE)
  vec[vec < q[1]] <- q[1]
  vec[vec > q[2]] <- q[2]
  vec
}

# second, construct your `pipeline_handler` method wrapper
winsorize <- function(eyeris, lower = 0.01, upper = 0.99, call_info = NULL) {
  # create call_info if not provided
  call_info <- if (is.null(call_info)) {
```

```
    list(
      call_stack = match.call(),
      parameters = list(lower = lower, upper = upper)
    )
  } else {
    call_info
  }

  # handle binocular objects
  if (eyeris:::is_binocular_object(eyeris)) {
    # process left and right eyes independently
    left_result <- eyeris$left |>
      pipeline_handler(
        winsorize_pupil,
        "winsorize",
        lower = lower,
        upper = upper,
        call_info = call_info
      )

    right_result <- eyeris$right |>
      pipeline_handler(
        winsorize_pupil,
        "winsorize",
        lower = lower,
        upper = upper,
        call_info = call_info
    )

    # return combined structure
    list_out <- list(
      left = left_result,
      right = right_result,
      original_file = eyeris$original_file,
      raw_binocular_object = eyeris$raw_binocular_object
    )

    class(list_out) <- "eyeris"

    return(list_out)
  } else {
    # regular eyeris object, process normally
    eyeris |>
      pipeline_handler(
        winsorize_pupil,
        "winsorize",
        lower = lower,
        upper = upper,
        call_info = call_info
      )
  }
}
```

```
# and voilà, you can now connect your custom extension
# directly into your custom `eyeris` pipeline definition!
custom_eye <- system.file("extdata", "memory.asc", package = "eyeris") |>
  eyeris::load_asc(block = "auto") |>
  eyeris::deblink(extend = 50) |>
  winsorize()

plot(custom_eye, seed = 1)
```

`plot.eyeris` *Plot pre-processed pupil data from* `eyeris`

Description

S3 plotting method for objects of class `eyeris`. Plots a single-panel timeseries for a subset of the pupil timeseries at each preprocessing step. The intended use of this function is to provide a simple method for qualitatively assessing the consequences of the preprocessing recipe and parameters on the raw pupillary signal.

Usage

```
## S3 method for class 'eyeris'
plot(
  x,
  ...,
  steps = NULL,
  preview_n = NULL,
  preview_duration = NULL,
  preview_window = NULL,
  seed = NULL,
  block = 1,
  plot_distributions = FALSE,
  suppress_prompt = TRUE,
  verbose = TRUE,
  add_progressive_summary = FALSE,
  eye = c("left", "right", "both"),
  num_previews = deprecated()
)
```

Arguments

`x`	An object of class `eyeris` derived from `load_asc()`
`...`	Additional arguments to be passed to `plot`
`steps`	Which steps to plot; defaults to `all` (i.e., plot all steps). Otherwise, pass in a vector containing the index of the step(s) you want to plot, with index `1` being the original raw pupil timeseries

`preview_n`	Number of random example "epochs" to generate for previewing the effect of each preprocessing step on the pupil timeseries
`preview_duration`	Time in seconds of each randomly selected preview
`preview_window`	The start and stop raw timestamps used to subset the preprocessed data from each step of the `eyeris` workflow for visualization Defaults to NULL, meaning random epochs as defined by `preview_n` and `preview_duration` will be plotted. To override the random epochs, set `preview_window` here to a vector with relative start and stop times (in seconds), for example – `c(5,6)` – to indicate the raw data from 5-6 secs on data that were recorded at 1000 Hz). Note, the start/stop time values indicated here are in seconds because `eyeris` automatically computes the indices for the supplied range of seconds using the `$info$sample.rate` metadata in the `eyeris` S3 class object
`seed`	Random seed for current plotting session. Leave NULL to select `preview_n` number of random preview "epochs" (of `preview_duration`) each time. Otherwise, choose any seed-integer as you would normally select for `base::set.seed()`, and you will be able to continue re-plotting the same random example pupil epochs each time – which is helpful when adjusting parameters within and across `eyeris` workflow steps
`block`	For multi-block recordings, specifies which block to plot. Defaults to 1. When a single `.asc` data file contains multiple recording blocks, this parameter determines which block's timeseries to visualize. Must be a positive integer not exceeding the total number of blocks in the recording
`plot_distributions`	Logical flag to indicate whether to plot both diagnostic pupil timeseries *and* accompanying histograms of the pupil samples at each processing step. Defaults to `FALSE`
`suppress_prompt`	Logical flag to disable interactive confirmation prompts during plotting. Defaults to `TRUE`, which avoids hanging behavior in non-interactive or automated contexts (e.g., RMarkdown, scripts) Set to `FALSE` only when running inside `glassbox()` with `interactive_preview = TRUE`, where prompting after each step is desired, as well as in the generation of interactive HTML reports with bidsify
`verbose`	A logical flag to indicate whether to print status messages to the console. Defaults to `TRUE`. Set to `FALSE` to suppress messages about the current processing step and run silently
`add_progressive_summary`	Logical flag to indicate whether to add a progressive summary plot after plotting. Defaults to `FALSE`. Set to `TRUE` to enable the progressive summary plot (useful for interactive exploration). Set to `FALSE` to disable the progressive summary plot (useful in automated contexts like bidsify reports)
`eye`	For binocular data, specifies which eye to plot: "left", "right", or "both". Defaults to "left". For "both", currently plots left eye data (use eye="right" for right eye data)
`num_previews`	**(Deprecated)** Use `preview_n` instead

Value

No return value; iteratively plots a subset of the pupil timeseries from each preprocessing step run

See Also

`lifecycle::deprecate_warn()`

Examples

```
# first, generate the preprocessed pupil data
my_eyeris_data <- system.file("extdata", "memory.asc", package = "eyeris") |>
  eyeris::load_asc() |>
  eyeris::deblink(extend = 50) |>
  eyeris::detransient() |>
  eyeris::interpolate() |>
  eyeris::lpfilt(plot_freqz = TRUE) |>
  eyeris::zscore()

# controlling the timeseries range (i.e., preview window) in your plots:

## example 1: using the default 10000 to 20000 ms time subset
plot(my_eyeris_data, seed = 0, add_progressive_summary = TRUE)

## example 2: using a custom time subset (i.e., 1 to 500 ms)
plot(
  my_eyeris_data,
  preview_window = c(0.01, 0.5),
  seed = 0,
  add_progressive_summary = TRUE
)

# controlling which block of data you would like to plot:

## example 1: plots first block (default)
plot(my_eyeris_data, seed = 0)

## example 2: plots a specific block
plot(my_eyeris_data, block = 1, seed = 0)

## example 3: plots a specific block along with a custom preview window
##   (i.e., 1000 to 2000 ms)
plot(
  my_eyeris_data,
  block = 1,
  preview_window = c(1, 2),
  seed = 0
)
```

`plot_binocular_correlation` *Plot binocular correlation between left and right eye data*

Description

Creates correlation plots showing the relationship between left and right eye measurements for pupil size, x-coordinates, and y-coordinates. This function is useful for validating binocular data quality and assessing the correlation between the two eyes.

Usage

```
plot_binocular_correlation(
  eyeris,
  block = 1,
  variables = c("pupil", "x", "y"),
  main = "",
  col_palette = "viridis",
  sample_rate = NULL,
  verbose = TRUE
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()` with binocular data, or a list containing `left` and `right` eyeris objects (from `binocular_mode = "both"`)
`block`	Block number to plot (default: 1)
`variables`	Variables to plot correlations for. Defaults to `c("pupil", "x", "y")` for pupil size, x-coordinates, and y-coordinates
`main`	Title for the overall plot (default: "Binocular Correlation")
`col_palette`	Color palette for the plots (default: "viridis")
`sample_rate`	Sample rate in Hz (optional, for time-based sampling)
`verbose`	Logical flag to indicate whether to print status messages (default: TRUE)

Value

No return value; creates correlation plots

Examples

```
# For binocular data loaded with binocular_mode = "both"
binocular_data <- load_asc(eyelink_asc_binocular_demo_dataset(), binocular_mode = "both")
plot_binocular_correlation(binocular_data)

# For binocular data loaded with binocular_mode = "average"
# (correlation plot will show original left vs right before averaging)
```

```
avg_data <- load_asc(eyelink_asc_binocular_demo_dataset(), binocular_mode = "average")
plot_binocular_correlation(avg_data$raw_binocular_object)
```

`plot_gaze_heatmap` *Create gaze heatmap of eye coordinates*

Description

Creates a heatmap showing the distribution of eye_x and eye_y coordinates across the entire screen area. The heatmap shows where the participant looked most frequently during the recording period.

Usage

```
plot_gaze_heatmap(
  eyeris,
  block = 1,
  screen_width = NULL,
  screen_height = NULL,
  n_bins = 50,
  col_palette = "viridis",
  main = "Gaze Heatmap",
  xlab = "Screen X (pixels)",
  ylab = "Screen Y (pixels)",
  sample_rate = NULL,
  eye_suffix = NULL
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`block`	Block number to plot (default: 1)
`screen_width`	Screen width in pixels from eyeris$info$screen.x
`screen_height`	Screen height in pixels from eyeris$info$screen.y
`n_bins`	Number of bins for the heatmap grid (default: 50)
`col_palette`	Color palette for the heatmap (default: "viridis")
`main`	Title for the plot (default: "Fixation Heatmap")
`xlab`	X-axis label (default: "Screen X (pixels)")
`ylab`	Y-axis label (default: "Screen Y (pixels)")
`sample_rate`	Sample rate in Hz (optional)
`eye_suffix`	Eye suffix for binocular data (default: NULL)

Value

No return value; creates a heatmap plot

Examples

```
demo_data <- eyelink_asc_demo_dataset()
eyeris_preproc <- glassbox(demo_data)
plot_gaze_heatmap(eyeris = eyeris_preproc, block = 1)
```

`summarize_confounds` *Extract confounding variables calculated separately for each pupil data file*

Description

Calculates various confounding variables for pupil data, including blink statistics, gaze position metrics, and pupil size characteristics. These confounds are calculated separately for each preprocessing step, recording block, and epoched timeseries in the eyeris object.

Usage

```
summarize_confounds(eyeris)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`

Value

An `eyeris` object with a new nested list of data frames: `$confounds` The confounds are organized hierarchically by block and preprocessing step. Each step contains metrics such as:

- Blink rate and duration statistics
- Gaze position (x,y) mean and standard deviation
- Pupil size mean, standard deviation, and range
- Missing data percentage

Examples

```
# load demo dataset
demo_data <- eyelink_asc_demo_dataset()

# calculate confounds for all blocks and preprocessing steps
confounds <- demo_data |>
  eyeris::glassbox() |>
  eyeris::epoch(
    events = "PROBE_{type}_{trial}",
    limits = c(-1, 1), # grab 1 second prior to and 1 second post event
    label = "prePostProbe" # custom epoch label name
  ) |>
  eyeris::summarize_confounds()
```

```
# access confounds for entire timeseries for a specific block and step
confounds$confounds$unepoched_timeseries

# access confounds for a specific epoched timeseries
# for a specific block and step
confounds$confounds$epoched_timeseries
confounds$confounds$epoched_epoch_wide
```

`zscore` *Z-score pupil timeseries data*

Description

The intended use of this method is to scale the arbitrary units of the pupil size timeseries to have a mean of `0` and a standard deviation of `1`. This is accomplished by mean centering the data points and then dividing them by their standard deviation (i.e., z-scoring the data, similar to `base::scale()`). Opting to z-score your pupil data helps with trial-level and between-subjects analyses where arbitrary units of pupil size recorded by the tracker do not scale across participants, and therefore make analyses that depend on data from more than one participant difficult to interpret.

Usage

```
zscore(eyeris, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`call_info`	A list of call information and parameters. If not provided, it will be generated from the function call

Details

This function is automatically called by `glassbox()` by default. Use `glassbox(zscore = FALSE)` to disable this step as needed.

Users should prefer using `glassbox()` rather than invoking this function directly unless they have a specific reason to customize the pipeline manually.

In general, it is common to z-score pupil data within any given participant, and furthermore, z-score that participant's data as a function of block number (for tasks/experiments where participants complete more than one block of trials) to account for potential time-on-task effects across task/experiment blocks.

As such, if you use the `eyeris` package as intended, you should NOT need to specify any groups for the participant/block-level situations described above. This is because `eyeris` is designed to preprocess a single block of pupil data for a single participant, one at a time. Therefore, when you later merge all of the preprocessed data from `eyeris`, each individual, preprocessed block of data for each participant will have already been independently scaled from the others.

Additionally, if you intend to compare mean z-scored pupil size across task conditions, such as that for memory successes vs. memory failures, then do NOT set your behavioral outcome (i.e., success/failure) variable as a grouping variable within your analysis. If you do, you will consequently obtain a mean pupil size of 0 and standard deviation of 1 within each group (since the scaled pupil size would be calculated on the timeseries from each outcome variable group, separately). Instead, you should compute the z-score on the entire pupil timeseries (before epoching the data), and then split and take the mean of the z-scored timeseries as a function of condition variable.

Value

An `eyeris` object with a new column in `timeseries`: `pupil_raw_{...}_z`

Note

This function is part of the `glassbox()` preprocessing pipeline and is not intended for direct use in most cases. Use `glassbox(zscore = TRUE)`.

Advanced users may call it directly if needed.

See Also

`glassbox()` for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(zscore = TRUE) |> # set to FALSE to skip (not recommended)
  plot(seed = 0)
```

Index