Package 'eyeris'

August 1, 2026

Type Package

Title Flexible, Extensible, & Reproducible Pupillometry Preprocessing

Version 3.3.0

Date 2026-07-30

Language en-US

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, splines, graphics, grDevices, tidyr, progress, data.table, withr, lifecycle, MASS, viridis, jsonlite, rmarkdown, DBI, glue, base64enc, reaborn, ggplot2, patchwork

Suggests arrow, duckdb, knitr, testthat (>= 3.0.0), devtools

VignetteBuilder knitr

License MIT + file LICENSE

Config/testthat/edition 3

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

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

Config/roxygen2/version 8.0.0

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 2026-07-31 22:20:02 UTC

Contents

`bidsify` *Save out pupil time series 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 time series, and formats the directory and filename structures based on the metadata you provide.

Usage

```
bidsify(
  eyeris,
  save_all = TRUE,
  epochs_list = NULL,
  bids_dir = NULL,
  participant_id = NULL,
  session_num = NULL,
  task_name = NULL,
  run_num = NULL,
  save_raw = TRUE,
  html_report = TRUE,
  report_seed = 0,
  report_epoch_grouping_var_col = "matched_event",
  verbose = TRUE,
  csv_enabled = TRUE,
  db_enabled = FALSE,
  db_path = "my-project",
  parallel_processing = FALSE,
  merge_epochs = deprecated(),
  merge_runs = deprecated(),
  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
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)
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 data frames 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
csv_enabled	Logical flag indicating whether to write CSV output files. Defaults to TRUE. Set to FALSE to disable CSV file generation, useful for large-scale cloud compute environments when using database storage only
db_enabled	Logical flag indicating whether to write data to a DuckDB database. Defaults to FALSE. When TRUE, creates or connects to a database for centralized data storage and querying
db_path	Database filename or path. Defaults to "eyeris-proj.eyerisdb". If just a filename, the database will be created in the derivatives/ directory. If a full path is provided, it will be used as specified

parallel_processing
: Logical flag to manually enable parallel database processing. When TRUE, uses temporary databases to avoid concurrency issues. Defaults to FALSE (auto-detect based on environment variables)

merge_epochs
: **(Deprecated)** This parameter is deprecated and will be ignored. All epochs are now saved as separate files following BIDS conventions. This parameter will be removed in a future version

merge_runs
: **(Deprecated)** This parameter is deprecated and will be ignored. All runs are now saved as separate files following BIDS conventions. This parameter will be removed in a future version

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 time series
    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
  )

# example with database storage enabled
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",
    db_enabled = TRUE,  # enable eyerisdb database storage
    db_path = "my-project"  # custom project database name
  )

# example for large-scale cloud compute (database only, no CSV files)
demo_data |>
  eyeris::glassbox() |>
  eyeris::bidsify(
    bids_dir = tempdir(),
    participant_id = "001",
```

```
  session_num = "01",
  task_name = "assocret",
  csv_enabled = FALSE,  # disable CSV files
  db_enabled = TRUE     # database storage only
)
```

---

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

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

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")) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))
```

`boilerplate` *Generate a reproducible, copy-and-paste-ready methods boilerplate*

Description

Auto-generates human-readable, "fMRIPrep-style" methods-section boilerplate text that describes – in plain prose – the exact preprocessing workflow that was run on an `eyeris` object. Because every `eyeris` preprocessing step records its own call stack and parameters (in `eyeris$params`), the pipeline is able to self-document precisely what it did to your data, ready to paste directly into the methods section of a manuscript.

This reinforces the workflow-level transparency at the heart of the `eyeris` "glassbox" philosophy: the code that ran and the prose that describes it are generated from one and the same source of truth.

Usage

```
boilerplate(
  eyeris,
  version = NULL,
  include_citation = TRUE,
  include_license = TRUE
)
```

Arguments

`eyeris`	An object of class `eyeris` (e.g., the output of `glassbox()`). Binocular objects are supported.

`version` Optional character string giving the `eyeris` version to cite in the boilerplate. Defaults to `NULL`, in which case the currently installed `eyeris` version is used.

`include_citation` Logical. Whether to append a formatted citation for `eyeris`. Defaults to `TRUE`.

`include_license` Logical. Whether to append the Creative Commons CC BY 4.0 license note telling users it is safe to paste the text into a manuscript as long as they cite `eyeris` for attribution. Defaults to `TRUE`.

Details

The generated text walks the recorded pipeline steps in the canonical order in which `eyeris` applies them (`load_asc` -> `deblink` -> `detransient` -> `interpolate` -> `lpfilt` -> `downsample/bin` -> `detrend` -> `zscore` -> `epoch`) and emits one sentence per step, substituting in the actual parameter values that were used. Steps that were skipped are omitted, and any custom (user-supplied) pipeline extensions are appended at the end so that multi-step and non-default pipelines are described faithfully. Multi-block (multi-run) and binocular recordings are handled automatically.

The boilerplate also references the per-run, machine-readable `.json` metadata sidecar that `eyeris` writes for every run (`source/logs/run-XX_metadata.json`), which together with the reported package version is sufficient to reproduce the pipeline exactly.

License & attribution. The auto-generated boilerplate text is licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0) license (https://creativecommons.org/licenses/by/4.0/). It is explicitly safe to copy and paste the generated Markdown content directly into your manuscript, provided that you give appropriate credit by citing `eyeris` (run `citation("eyeris")` for the reference); including that citation in your references satisfies the attribution requirement. This mirrors the approach taken by fMRIPrep and other reproducible-pipeline tools.

When you run `bidsify()` (or `glassbox()` with reporting enabled), this same boilerplate is embedded in the diagnostic HTML report and written to `derivatives/.../source/logs/` as a standalone Markdown file.

Value

A length-one character string of Markdown-formatted methods boilerplate (invisibly classed as `eyeris_boilerplate` so it prints nicely at the console). The raw string can be passed to `base::cat()`, `base::writeLines()`, or embedded directly in a report.

See Also

`glassbox()`, `bidsify()`

Examples

```
demo_data <- eyelink_asc_demo_dataset()

eyeris_obj <- demo_data |>
  eyeris::glassbox(verbose = FALSE)

# print the copy-and-paste-ready methods boilerplate
boilerplate(eyeris_obj)
```

```
# capture it as plain Markdown text (e.g., to write to a file)
methods_md <- boilerplate(eyeris_obj)
cat(methods_md)
```

`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

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# 50 ms in both directions (the default)
demo_data |>
  eyeris::glassbox(deblink = list(extend = 50)) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))

# 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, preview_window = c(10, 12))
```

`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 `time series`: `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.

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(
    detransient = list(n = 16) # set to FALSE to skip step (not recommended)
  ) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))
```

`detrend` *Detrend the pupil time series*

Description

Detrend pupil data by fitting a model of `pupil_data ~ time` and returning the fitted values (the estimated trend) together with the residuals (`pupil_data - fitted_values`). Two trend models are supported via the `method` argument: `"linear"` (the default) removes a straight-line drift by fitting an ordinary linear model, while `"spline"` removes a smooth, potentially nonlinear drift by fitting a natural cubic spline basis of `time` (`splines::ns(time, df = spline_df)`).

Usage

```
detrend(
  eyeris,
  method = c("linear", "spline"),
  spline_df = 5,
  call_info = NULL
)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`method`	A string indicating the detrending model to fit. Either `"linear"` (the default) to remove a straight-line trend by regressing pupil size on time, or `"spline"` to remove a smooth, potentially nonlinear trend by fitting a natural cubic spline basis of time (`splines::ns(time, df = spline_df)`)
`spline_df`	The degrees of freedom for the natural cubic spline basis used when `method = "spline"`. Higher values allow the fitted trend to follow more rapid, nonlinear drift; lower values enforce a smoother trend. Must be a single whole number `>= 1`. Defaults to `5`. Ignored when `method = "linear"`
`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 `time series`: `detrend_fitted_values`, 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)` for linear detrending, or `glassbox(detrend = list(method = "spline"))` for spline detrending.

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

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

# (a) linear detrending (the default)
demo_data |>
  eyeris::glassbox(detrend = TRUE) |>  # set to FALSE to skip step (default)
  # a wider `preview_window` makes the (slow) fitted trend easier to see
  plot(seed = 0, preview_window = c(5, 20))

# (b) spline detrending (removes a smooth, nonlinear trend)
demo_data |>
  eyeris::glassbox(detrend = list(method = "spline", spline_df = 5)) |>
  plot(seed = 0, preview_window = c(5, 20))
```

`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.

When the input contains gaps longer than the interpolation limit (see interpolate()'s max_gap_ms) that were left as NA, those gaps are temporarily filled so the anti-aliasing filter can run and then masked back to NA. This can slightly bias the valid samples immediately adjacent to each gap toward the interpolated values, so a warning is emitted in this case. If that bias is a concern, consider disabling this step (glassbox(downsample = FALSE)).

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.

For a complete, end-to-end reference pipeline that demonstrates how all eyeris preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the

Anatomy of an `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

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)) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))
```

`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 data frames that manually specify start/end event timestamp-message pairs to pull out of the raw time series 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 time series 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)
```

`eyeris_db_collect` *Extract and aggregate eyeris data across subjects from database*

Description

A comprehensive wrapper function that simplifies extracting `eyeris` data from the database. Provides easy one-liner access to aggregate data across multiple subjects for each data type, without requiring SQL knowledge.

Usage

```
eyeris_db_collect(
  bids_dir,
  db_path = "my-project",
  subjects = NULL,
  data_types = NULL,
  sessions = NULL,
  tasks = NULL,
  epoch_labels = NULL,
  eye_suffixes = NULL,
  verbose = TRUE
)
```

Arguments

`bids_dir`	Path to the BIDS directory containing the database
`db_path`	Database name (defaults to "my-project", becomes "my-project.eyerisdb")
`subjects`	Vector of subject IDs to include. If NULL (default), includes all subjects
`data_types`	Vector of data types to extract. If NULL (default), extracts all available types. Valid types: "blinks", "events", "timeseries", "epochs", "epoch_summary", "run_confounds", "confounds_events", "confounds_summary"
`sessions`	Vector of session IDs to include. If NULL (default), includes all sessions
`tasks`	Vector of task names to include. If NULL (default), includes all tasks
`epoch_labels`	Vector of epoch labels to include. If NULL (default), includes all epochs. Only applies to epoch-related data types
`eye_suffixes`	Vector of eye suffixes to include. If NULL (default), includes all eyes. Typically c("eye-L", "eye-R") for binocular data
`verbose`	Logical. Whether to print progress messages (default TRUE)

Value

A named list of data frames, one per data type

Examples

```
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
  db_enabled = TRUE # enable database storage
)

# extract all data for all subjects (returns list of data frames)
all_data <- eyeris_db_collect(tempdir())

# view available data types
names(all_data)

# access specific data type
blinks_data <- all_data$blinks
epochs_data <- all_data$epochs

# extract specific subjects and data types
subset_data <- eyeris_db_collect(
  bids_dir = tempdir(),
  subjects = c("001"),
  data_types = c("blinks", "epochs", "timeseries")
)

# extract epoch data for specific epoch label
epoch_data <- eyeris_db_collect(
  bids_dir = tempdir(),
  data_types = "epochs",
  epoch_labels = "prepostprobe"
)
```

`eyeris_db_connect` *Connect to eyeris project database (user-facing)*

Description

User-friendly function to connect to an existing `eyeris` project database. This function provides easy access for users to query their `eyeris` data.

Usage

```
eyeris_db_connect(bids_dir, db_path = "my-project")
```

Arguments

`bids_dir`	Path to the BIDS directory containing the database
`db_path`	Database name (defaults to "my-project", becomes "my-project.eyerisdb") If just a filename, will look in `derivatives/` directory. If includes path, will use as provided.

Value

Database connection object for use with other eyeris database functions

Examples

```
# step 1: create a database using bidsify with db_enabled = TRUE
# (This example assumes you have already run bidsify to create a database)

# temp dir for testing
temp_dir <- tempdir()

# step 2: connect to eyeris DB (will fail gracefully if no DB exists)
tryCatch({
  con <- eyeris_db_connect(temp_dir)

  tables <- eyeris_db_list_tables(con)

  # read timeseries data for a specific subject
  data <- eyeris_db_read(con, data_type = "timeseries", subject = "001")

  # close connection when done
  eyeris_db_disconnect(con)
}, error = function(e) {
  message("No eyeris DB found - create one first with bidsify(db_enabled = TRUE)")
})
```

eyeris_db_disconnect *Disconnect from eyeris database (user-facing)*

Description

User-friendly function to disconnect from the `eyeris` project database.

Usage

```
eyeris_db_disconnect(con)
```

Arguments

`con`	Database connection object

Value

Logical indicating success

eyeris_db_list_tables *List available tables in eyeris database*

Description

Lists all tables in the `eyeris` project database with optional filtering.

Usage

```
eyeris_db_list_tables(con, data_type = NULL, subject = NULL)
```

Arguments

`con`	Database connection
`data_type`	Optional filter by data type
`subject`	Optional filter by subject ID

Value

Character vector of table names

`eyeris_db_read` *Read eyeris data from database*

Description

Reads `eyeris` data from the project database with dplyr-style interface.

Usage

```
eyeris_db_read(
  con,
  data_type = NULL,
  subject = NULL,
  session = NULL,
  task = NULL,
  run = NULL,
  eye_suffix = NULL,
  epoch_label = NULL,
  table_name = NULL
)
```

Arguments

`con`	Database connection
`data_type`	Type of data to read ("timeseries", "epochs", "epoch_timeseries", "epoch_summary", "events", "blinks")
`subject`	Optional subject ID filter
`session`	Optional session ID filter
`task`	Optional task name filter
`run`	Optional run number filter
`eye_suffix`	Optional eye suffix filter
`epoch_label`	Optional epoch label filter (for epoched data)
`table_name`	Exact table name (overrides other parameters)

Value

Data frame with requested data

`eyeris_db_reconstruct_from_chunks` *Reconstruct eyerisdb from chunked files*

Description

Merges multiple chunked eyerisdb files back into a single database file. Uses the reconstruction metadata file created by `eyeris_db_split_for_sharing()` to ensure proper reconstruction.

Usage

```
eyeris_db_reconstruct_from_chunks(
  chunked_dir,
  output_path,
  reconstruction_file = NULL,
  verbose = TRUE
)
```

Arguments

`chunked_dir`	Directory containing the chunked database files and reconstruction metadata
`output_path`	Full path for the reconstructed database (e.g., "/path/to/reconstructed.eyerisdb")
`reconstruction_file`	Path to the reconstruction metadata JSON file. If NULL (default), searches for "*_reconstruction_info.json" in chunked_dir
`verbose`	Whether to print progress messages (default: TRUE)

Value

List containing information about the reconstruction process

Examples

```
## Not run:
# Reconstruct database from chunked files
reconstruction_info <- eyeris_db_reconstruct_from_chunks(
  chunked_dir = "/path/to/chunked_db/project-name",
  output_path = "/path/to/reconstructed-project.eyerisdb"
)

# Specify custom reconstruction file location
reconstruction_info <- eyeris_db_reconstruct_from_chunks(
  chunked_dir = "/path/to/chunked_db/project-name",
  output_path = "/path/to/reconstructed-project.eyerisdb",
  reconstruction_file = "/path/to/custom_reconstruction_info.json"
)

## End(Not run)
```

`eyeris_db_split_for_sharing` *Split eyerisdb for data sharing and distribution*

Description

Creates multiple smaller eyerisdb files from a single large database for easier distribution via platforms with file size limits (GitHub, OSF, data repositories, etc.). Data can be chunked by data type, by number of chunks, or by maximum file size. Includes metadata to facilitate reconstruction of the original database.

Usage

```
eyeris_db_split_for_sharing(
  bids_dir,
  db_path = "my-project",
  output_dir = NULL,
  chunk_strategy = "by_data_type",
  n_chunks = 4,
  max_chunk_size_mb = 100,
  data_types = NULL,
  group_by_epoch_label = TRUE,
  include_metadata = TRUE,
  verbose = TRUE
)
```

Arguments

`bids_dir`	Path to the BIDS directory containing the source database
`db_path`	Source database name (defaults to "my-project", becomes "my-project.eyerisdb")
`output_dir`	Directory to save chunked databases (defaults to bids_dir/derivatives/chunked_db)
`chunk_strategy`	Strategy for chunking: "by_data_type", "by_count", or "by_size" (default: "by_data_type")
`n_chunks`	Number of chunks to create when chunk_strategy = "by_count" (default: 4)
`max_chunk_size_mb`	Maximum size per chunk in MB when chunk_strategy = "by_size" (default: 100)
`data_types`	Vector of data types to include. If NULL (default), includes all available
`group_by_epoch_label`	If TRUE (default), processes epoch-related data types separately by epoch label
`include_metadata`	Whether to include eyeris metadata columns in chunked databases (default: TRUE)
`verbose`	Whether to print progress messages (default: TRUE)

Value

List containing information about created chunked databases and reconstruction instructions

Examples

```
## Not run:
# These examples require an existing eyeris database

# Chunk by data type (each data type gets its own database file)
chunk_info <- eyeris_db_split_for_sharing(
  bids_dir = "/path/to/bids",
  db_path = "large-project",
  chunk_strategy = "by_data_type"
)

# Chunk into 6 files by count
chunk_info <- eyeris_db_split_for_sharing(
  bids_dir = "/path/to/bids",
  db_path = "large-project",
  chunk_strategy = "by_count",
  n_chunks = 6
)

# Chunk by size (max 50MB per file)
chunk_info <- eyeris_db_split_for_sharing(
  bids_dir = "/path/to/bids",
  db_path = "large-project",
  chunk_strategy = "by_size",
  max_chunk_size_mb = 50
)

## End(Not run)
```

---

eyeris_db_summary *Get summary statistics for eyeris database*

Description

Provides a quick overview of the contents of an `eyeris` database, including available subjects, sessions, tasks, and data types.

Usage

```
eyeris_db_summary(bids_dir, db_path = "my-project", verbose = TRUE)
```

Arguments

bids_dir	Path to the BIDS directory containing the database
db_path	Database name (defaults to "my-project", becomes "my-project.eyerisdb")
verbose	Logical. Whether to print detailed output (default TRUE)

Value

A named list containing summary information about the database contents

Examples

```
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 = file.path(tempdir(), "my-cool-memory-project"),
    participant_id = "001",
    session_num = "01",
    task_name = "assocret",
    run_num = "03", # override default run-01 (block_1) to use run-03 instead
    db_enabled = TRUE,
    db_path = "my-cool-memory-study",
  )

# get database summary
summary <- eyeris_db_summary(
             file.path(
               tempdir(),
               "my-cool-memory-project"
             ),
             db_path = "my-cool-memory-study"
           )

# view available subjects
summary$subjects

# view available data types
summary$data_types

# view table counts
summary$table_counts
```

`eyeris_db_to_chunked_files` *Export eyeris database to chunked files*

Description

High-level wrapper function to export large eyeris databases to chunked CSV or Parquet files by data type. Uses chunked processing to handle very large datasets without memory issues.

Usage

```
eyeris_db_to_chunked_files(
  bids_dir,
  db_path = "my-project",
  output_dir = NULL,
  chunk_size = 1e+06,
  file_format = "csv",
  data_types = NULL,
  subjects = NULL,
  max_file_size_mb = 50,
  group_by_epoch_label = TRUE,
  verbose = TRUE
)
```

Arguments

`bids_dir`	Path to the BIDS directory containing the database
`db_path`	Database name (defaults to "my-project", becomes "my-project.eyerisdb")
`output_dir`	Directory to save output files (defaults to bids_dir/derivatives/eyerisdb_export)
`chunk_size`	Number of rows to process per chunk (default: 1000000)
`file_format`	Output format: "csv" or "parquet" (default: "csv")
`data_types`	Vector of data types to export. If NULL (default), exports all available
`subjects`	Vector of subject IDs to include. If NULL (default), includes all subjects
`max_file_size_mb`	Maximum file size in MB per output file (default: 50). When exceeded, automatically creates numbered files (e.g., data_01-of-03.csv, data_02-of-03.csv)
`group_by_epoch_label`	If TRUE (default), processes epoch-related data types separately by epoch label to reduce memory footprint and produce label-specific files. When FALSE, epochs with different labels are merged into single large files (not recommended).
`verbose`	Whether to print progress messages (default: TRUE)

Value

List containing information about exported files

Examples

```
## Not run:
# These examples require an existing eyeris database

# Export entire database to CSV files
if (file.exists(file.path(tempdir(), "derivatives", "large-project.eyerisdb"))) {
  export_info <- eyeris_db_to_chunked_files(
    bids_dir = tempdir(),
    db_path = "large-project",
    chunk_size = 50000,
    file_format = "csv"
  )
}

# Export specific data types to Parquet
if (file.exists(file.path(tempdir(), "derivatives", "large-project.eyerisdb"))) {
  export_info <- eyeris_db_to_chunked_files(
    bids_dir = tempdir(),
    db_path = "large-project",
    data_types = c("timeseries", "events"),
    file_format = "parquet",
    chunk_size = 75000
  )
}

## End(Not run)
```

---

eyeris_db_to_parquet *Split eyeris database into N parquet files by data type*

Description

Utility function that takes an eyerisdb DuckDB database and splits it into N reasonably sized parquet files for easy management with GitHub, downloading, and distribution. Data is first grouped by table type (timeseries, epochs, events, etc.) since each has different columnar structures, then each group is split into the specified number of files. Files are organized in folders matching the database name for easy identification.

Usage

```
eyeris_db_to_parquet(
  bids_dir,
  db_path = "my-project",
  n_files_per_type = 1,
  output_dir = NULL,
  max_file_size = 512,
  data_types = NULL,
  verbose = TRUE,
```

```
  include_metadata = TRUE,
  epoch_labels = NULL,
  group_by_epoch_label = TRUE
)
```

Arguments

bids_dir	Path to the BIDS directory containing the database
db_path	Database name (defaults to "my-project", becomes "my-project.eyerisdb")
n_files_per_type	Number of parquet files to create per data type (default: 1)
output_dir	Directory to save parquet files (defaults to bids_dir/derivatives/parquet)
max_file_size	Maximum file size in MB per parquet file (default: 512) Used as a constraint when n_files_per_type would create files larger than this
data_types	Vector of data types to include. If NULL (default), includes all available. Valid types: "timeseries", "epochs", "epoch_summary", "events", "blinks", "confounds_*"
verbose	Whether to print progress messages (default: TRUE)
include_metadata	Whether to include eyeris metadata columns in output (default: TRUE)
epoch_labels	Optional character vector of epoch labels to include (e.g., "prepostprobe"). Only applies to epoch-related data types. If NULL, includes all labels.
group_by_epoch_label	If TRUE, processes epoch-related data types separately by epoch label to reduce memory footprint and produce label-specific parquet files (default: TRUE).

Value

List containing information about created parquet files

Database Safety

This function creates temporary tables during parquet export when the arrow package is not available. All temporary tables are automatically cleaned up, but if the process crashes, leftover tables may remain. The function checks for and warns about existing temporary tables before starting.

Examples

```
# create demo database
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 = "memory",
    db_enabled = TRUE,
    db_path = "memory-task"
  )

# split into 3 parquet files per data type - creates memory-task/ folder
split_info <- eyeris_db_to_parquet(
  bids_dir = tempdir(),
  db_path = "memory-task",
  n_files_per_type = 3
)

# split with size constraint and specific data types using the same database
split_info <- eyeris_db_to_parquet(
  bids_dir = tempdir(),
  db_path = "memory-task",
  n_files_per_type = 5,
  max_file_size = 50,  # max 50MB per file
  data_types = c("timeseries", "epochs", "events")
)
```

`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	Either an SR Research EyeLink .asc file generated by the official EyeLink edf2asc command, or a pre-constructed eyeris object. When an .asc path is supplied, the load_asc step reads and parses it. When a pre-loaded eyeris object is supplied instead (for example, the output of load_generic() for non-EyeLink trackers, or an existing load_asc() object), the load step is skipped and the remaining pipeline runs on the object as-is – enabling load_generic(...) |> glassbox()
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 time series
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 time series 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 time series 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 time series data.

Value

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

What `glassbox()` does, step by step

In plain language, `glassbox()` runs the following `eyeris` steps in order on your pupil time series. Steps marked **(default: on)** run automatically; steps marked **(default: off)** are skipped unless you explicitly enable them. Most preprocessing steps can be turned off by passing `<step> = FALSE` (except `load_asc`, which always runs when `file` is an `.asc` path). Steps that accept parameters can be customized by passing `<step> = list(...)` with values you want to override (for example, `deblink = list(extend = 40)`).

1. **Load the data** (`load_asc`, default: on) – Reads and parses the EyeLink `.asc` file into an `eyeris` object, automatically splitting the recording into blocks and, for binocular recordings, handling each eye separately. See `load_asc()`. This step is skipped when a pre-loaded `eyeris` object is passed as `file` (see the `file` parameter).
2. **Resample onto a uniform grid** (`resample`, default: on) – Places each block on the expected uniform sampling grid. For hardware that *drops* samples (instead of zero-filling) when pupil data is missing, this interpolates local sub-period timing jitter and inserts `NA` rows at the dropped timestamps, so the rate-dependent steps that follow stay valid. A guaranteed no-op for already-uniform data (e.g., EyeLink). See `resample()`.
3. **Remove blinks** (`deblink`, default: on) – Replaces the missing data around blinks with `NA`s, extending each gap by `50` ms on either side so that the rapid dips and spikes that surround a blink are removed too. See `deblink()`.
4. **Remove transient artifacts** (`detransient`, default: on) – Rejects pupil samples that change faster than is physiologically plausible, using a speed-based median absolute deviation (MAD) threshold. See `detransient()`.
5. **Interpolate missing samples** (`interpolate`, default: on) – Fills the `NA` gaps left by the resample, deblink, and detransient steps using linear interpolation, producing a continuous, gap-free time series. See `interpolate()`.
6. **Smooth the signal** (`lpfilt`, default: on) – Applies a low-pass filter (default `4` Hz passband) to remove high-frequency noise while preserving the slower pupil dynamics of interest. See `lpfilt()`.

7. **Downsample** (`downsample`, default: off) – Optionally lowers the sampling rate using an anti-aliasing filter, which preserves the temporal dynamics of the signal. Cannot be combined with `bin`. See `downsample()`.
8. **Bin** (`bin`, default: off) – Optionally lowers the sampling rate by averaging samples within equal-width time bins. Cannot be combined with `downsample`. See `bin()`.
9. **Detrend** (`detrend`, default: off) – Optionally fits a model of `pupil ~ time` and returns the residuals (along with the fitted trend) to remove slow drift. By default (`detrend = TRUE`) a straight-line (`method = "linear"`) trend is removed; pass `detrend = list(method = "spline", spline_df = 5)` to instead remove a smooth, potentially nonlinear trend via a natural cubic spline of time. Use with care – see `detrend()` for when this is appropriate.
10. **Z-score** (`zscore`, default: on) – Rescales the pupil time series to a mean of `0` and a standard deviation of `1`, making values comparable across participants and recordings. See `zscore()`.

After preprocessing, `glassbox()` calls `summarize_confounds()` to compute per-step confound metrics (e.g., missingness and gaze statistics) and store them in `$confounds`.

Crucially, each step *adds a new column* to the time series rather than overwriting the previous one, so every intermediate stage is preserved inside the returned `eyeris` object. This is what makes the pipeline a "glass box": you can inspect, plot, and compare the data before and after each transformation (for example, `plot(output, steps = c(1, 5))`).

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),
  # only interpolate gaps up to 100 ms; longer gaps are left as NA
  interpolate = list(max_gap_ms = 100),
  lpfilt = list(plot_freqz = TRUE) # overrides verbose parameter
)

# to suppress messages, set verbose = FALSE in plot():
plot(output, seed = 0, verbose = FALSE, preview_window = c(10, 12))

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

plot(output, seed = 0, preview_window = c(10, 12))
```

`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, max_gap_ms = 250, verbose = TRUE, call_info = NULL)
```

Arguments

`eyeris`	An object of class `eyeris` derived from `load_asc()`
`max_gap_ms`	The maximum duration (in milliseconds) of a gap of missing (`NA`) pupil samples that will be interpolated. Gaps longer than this threshold are left as `NA` (not interpolated). Must be greater than `0`; defaults to `250` ms, the value recommended by Kret & Sjak-Shie (2018). Set to `Inf` (or `NULL`) to interpolate across all gaps regardless of duration. The threshold is converted to a whole number of samples (rounded down) using each recording's sampling rate. To skip interpolation entirely, set `interpolate = FALSE` in `glassbox()` instead

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

By default, only gaps shorter than or equal to max_gap_ms milliseconds are interpolated. Any gap longer than this threshold is left as NA rather than being interpolated over. This follows the recommendation of Kret & Sjak-Shie (2018) to avoid interpolating across long stretches of missing data, where linear interpolation is unlikely to reflect the true underlying pupil signal. The default of 250 ms matches the value used in that paper.

Set max_gap_ms = Inf (or NULL) to disable the limit and interpolate across all gaps, restoring the behavior of eyeris versions <= 3.2.0.

Downstream glassbox() steps that cannot operate on missing data (low-pass filtering, downsampling, and binning) automatically work *around* these retained gaps – filtering/resampling over a temporarily filled copy and then restoring the gaps as NA – so the gaps are preserved through to the final preprocessed output. Because of this temporary fill, the filtering steps (lpfilt() and the anti-aliasing filter in downsample()) can slightly bias the valid samples immediately adjacent to a long retained gap toward the interpolated values. These steps therefore emit a warning when they operate over such gaps, so you can choose to disable them (e.g. lpfilt = FALSE and/or downsample = FALSE in glassbox()) if this bias is a concern for your analysis.

Note: Prior to eyeris version 3.3.0, all gaps were interpolated regardless of duration. Enforcing max_gap_ms is a change in default behavior and may affect downstream results.

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), or provide parameters via glassbox(interpolate = list(max_gap_ms = ...)).

Advanced users may call it directly if needed.

References

Kret, M. E., & Sjak-Shie, E. E. (2018). Preprocessing pupil size data: Guidelines and code. *Behavior Research Methods, 51*(3), 1336-1342. doi:10.3758/s134280181075y

See Also

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

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  # set to FALSE to skip (not recommended)
  eyeris::glassbox(interpolate = TRUE) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))

# only interpolate gaps up to 100 ms; leave longer gaps as NA
demo_data |>
  eyeris::glassbox(interpolate = list(max_gap_ms = 100)) |>
  plot(seed = 0, preview_window = c(10, 12))
```

---

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"),
  verbose = TRUE
)
```

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.

`verbose` Logical. Whether to print verbose output (default TRUE).

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.

For a complete, end-to-end reference pipeline that demonstrates how all `eyeris` preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")` — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: `vignette("complete-pipeline", package = "eyeris")`.

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)
```

`load_generic` *Load pupillometry data from a non-EyeLink eye tracker*

Description

Construct a valid `eyeris` S3 object from standardized data frames so that data from eye trackers other than SR Research EyeLink can enter the `eyeris` preprocessing pipeline. While `load_asc()` parses EyeLink `.asc` files specifically, `load_generic()` provides a tracker-agnostic ingestion path: you supply the raw samples (and, optionally, event messages, gaze coordinates, and blink intervals) as plain R data frames, and `load_generic()` assembles them into the same object structure that the rest of `eyeris` (including `glassbox()`) expects.

Usage

```
load_generic(
  pupil,
  events = NULL,
  blinks = NULL,
  gaze = NULL,
  sample_rate = NULL,
  time_unit = c("ms", "s"),
  block = "auto",
  eye = c("L", "R", "LR"),
  pupil_type = c("area", "diameter"),
  screen_width = NA_real_,
  screen_height = NA_real_,
  tracker = "generic",
  model = NA_character_,
  mapping = NULL,
  path = NULL,
  verbose = TRUE
)
```

Arguments

pupil	A data frame of raw samples. Must contain a timestamp column and a pupil-size column (see mapping). May optionally contain gaze coordinate and block columns.
events	An optional data frame of event messages with a timestamp column (on the same clock as pupil) and a message-text column. If NULL (default), the object is created with empty event tables.
blinks	An optional data frame of blink intervals with start and end timestamp columns. If NULL (default), empty blink tables are created. Note that blink padding via deblink() does not depend on this table.
gaze	An optional data frame of gaze coordinate samples (timestamp, x, y), used only when gaze is exported separately from pupil size. It is left-joined onto pupil by timestamp. If NULL (default), gaze is taken from pupil if those columns are present, otherwise filled with NA.
sample_rate	Numeric sampling rate of the tracker in Hz. If NULL (default), it is inferred from the median spacing of the pupil timestamps; inference is reported and we recommend supplying the true rate explicitly.
time_unit	Unit of all timestamp columns (in pupil, events, gaze, and blinks). Either "ms" (milliseconds, the default, matching EyeLink) or "s" (seconds). Timestamps are stored internally in milliseconds.
block	Block specification, mirroring load_asc(): • "auto" (default): if the pupil data contains a block column with more than one unique value, the data are split into multiple blocks; otherwise a single block (block_1) is created. • NULL: omit the block column and create a single block.

- Numeric value: assign this block number to all samples.

`eye` Which eye the data correspond to: `"L"` (left, default), `"R"` (right), or `"LR"` (both, e.g., averaged). Recorded as metadata.

`pupil_type` Pupil measurement units: `"area"` (default) or `"diameter"`. Recorded as metadata (the `type` column).

`screen_width, screen_height`
Optional screen dimensions in pixels, used for gaze heatmaps and gaze-based confounds. Leave as `NA` (default) if unknown; the gaze heatmap is simply skipped.

`tracker` Character label for the source tracker/system (default `"generic"`). Recorded in `info$version`.

`model` Optional character label for the specific tracker model. Recorded in `info$model`.

`mapping` An optional named list remapping the default column names to the ones present in your data frames. Recognized names are `time`, `pupil`, `eye_x`, `eye_y`, `text`, `stime`, `etime`, and `block`. For example, `mapping = list(time = "t_ms", pupil = "pup_size")`.

`path` Optional character path/identifier stored in the object's `file` slot (used in report titles). Defaults to the value of `tracker`.

`verbose` Logical. Whether to print verbose output (default `TRUE`).

Details

`eyeris` was designed to be extensible, but historically `load_asc()` was the only implemented loader. `load_generic()` closes that gap. Native support for specific tracker formats (e.g., Tobii, SMI, Pupil Labs, GazePoint) can be layered on top of this function incrementally: a format-specific reader only needs to produce the standardized data frames documented below and then call `load_generic()`.

The resulting object is structurally identical to one returned by `load_asc()`, so it is a drop-in input to `glassbox()` and the individual preprocessing steps (`deblink()`, `detransient()`, `interpolate()`, `lpfilt()`, `downsample()`, `bin()`, `detrend()`, `zscore()`), `epoch()`, `bidsify()`, and the plotting methods.

The three standardized data frames:

Following the conceptual model that most trackers export, `load_generic()` accepts three core data frames (plus an optional fourth for gaze that is exported separately):

1. `pupil` (required) – the raw sample stream. Must contain a timestamp column and a pupil-size column. May *also* carry gaze coordinate columns (`eye_x`, `eye_y`) if your tracker exports samples as one wide table.
2. `events` (optional) – experimental event messages. Must contain a timestamp column (on the same clock as `pupil`) and a message-text column. Required only if you intend to epoch on event messages later.
3. `blinks` (optional) – blink intervals reported by the tracker. Must contain blink start and end timestamp columns. Note that blink padding via `deblink()` does *not* depend on this table – it reconstructs missing/blink regions directly from `NA` (and `0`) values in the pupil column – so this table is purely for record-keeping and export.

A fourth, gaze, data frame is accepted for the less common case where gaze coordinates are exported separately from pupil size (timestamp + x/y columns); it is joined onto pupil by timestamp. If your pupil data frame already contains gaze columns, leave this NULL.

If a column in your data frames does not use the default name expected by load_generic(), remap it via the mapping argument (see below) rather than renaming your data by hand.

Column requirements and defaults:

By default load_generic() looks for these columns (override any of them with mapping):

data frame	role	default column
pupil	timestamp	time
pupil	pupil size	pupil
pupil	gaze x (optional)	eye_x
pupil	gaze y (optional)	eye_y
pupil	block (optional)	block
events	timestamp	time
events	message text	text
blinks	blink start	stime
blinks	blink end	etime
gaze	timestamp	time
gaze	gaze x	eye_x
gaze	gaze y	eye_y

Handling tracker quirks:

eyeris assumes EyeLink-style regularly-sampled data. Two practical notes for other trackers:

- **Missing samples.** deblink() reconstructs missing/blink regions directly from NA (and 0) values in the pupil column – it does *not* require the blinks table. If your tracker drops samples or encodes missing pupil data some other way, set those samples to NA in the pupil column so deblinking and the confound calculations behave correctly.
- **Irregular sampling.** If consecutive timestamps are not uniformly spaced, load_generic() emits a warning, because several downstream steps assume a fixed sampling interval. (A fuller guardrail is tracked separately.)

Value

An object of S3 class eyeris with the same structure as load_asc():

1. file: The path identifier for the source data.
2. timeseries: A named list of per-block data frames of raw time series data (time_orig, time_secs, time_scaled, eye_x, eye_y, eye, hz, type, pupil_raw).
3. events: A named list of per-block event-message data frames.
4. blinks: A named list of per-block blink data frames.
5. info: Tracker metadata (sample.rate, mono, left, right, pupil.dtype, version, model, screen.x, screen.y).
6. latest: eyeris pointer for tracking pipeline run history.
7. binocular, binocular_mode, decimated.sample.rate, params.

See Also

load_asc() for loading SR Research EyeLink .asc files.

glassbox() for running the full eyeris preprocessing pipeline on the object returned by this function.

Examples

```
# build three small standardized data frames from any non-EyeLink tracker
set.seed(1)
n <- 1000
samples <- data.frame(
  time = seq(0, by = 1, length.out = n), # 1000 Hz -> 1 ms spacing
  pupil = 1000 + cumsum(rnorm(n, 0, 5)),
  eye_x = 960 + rnorm(n, 0, 10),
  eye_y = 540 + rnorm(n, 0, 10)
)

events <- data.frame(
  time = c(100, 500),
  text = c("TRIALID 1", "TRIALID 2")
)

# construct a valid eyeris object
eye <- eyeris::load_generic(
  pupil = samples,
  events = events,
  sample_rate = 1000,
  screen_width = 1920,
  screen_height = 1080,
  tracker = "my-tracker"
)

# ...and run it straight through the glassbox pipeline
eye |>
  eyeris::glassbox(lpfilt = list(plot_freqz = FALSE))
```

---

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.

When the input contains gaps longer than the interpolation limit (see interpolate()'s max_gap_ms) that were left as NA, those gaps are temporarily filled so the Butterworth filter can run and then masked back to NA. This can slightly bias the valid samples immediately adjacent to each gap toward the interpolated values, so a warning is emitted in this case. If that bias is a concern, consider disabling this step (glassbox(lpfilt = FALSE)).

Value

An eyeris object with a new column in time series: 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

For a complete, end-to-end reference pipeline that demonstrates how all eyeris preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* eyeris *Object* vignette — vignette("anatomy", package = "eyeris") — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: vignette("complete-pipeline", package = "eyeris").

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)) |>
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))
```

---

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 time series data in a list of data frames (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 time series data. This custom function should accept a data frame 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 data frame(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 time series data frame.

Value

An updated eyeris object with the new column added to the timeseries data frame 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 time series 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 time series
`preview_n`	Number of random example "epochs" to generate for previewing the effect of each preprocessing step on the pupil time series
`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 time series 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 time series *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 time series 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 time series range (i.e., preview window) in your plots:
#  (note: omitting `preview_window` uses the default 10000 to 20000 ms
#   subset; the examples below pass shorter windows explicitly so that they
#   run quickly)

## example 1: a 2-second time subset (i.e., 10000 to 12000 ms)
plot(
  my_eyeris_data,
  preview_window = c(10, 12),
  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
)

# controlling which block of data you would like to plot:
#  (`block` defaults to 1, i.e., the first block; here it is combined with a
#   custom preview window of 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)
```

`print.eyeris_sim_params` *Print a synthetic pupil parameter list*

Description

Print a synthetic pupil parameter list

Usage

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

Arguments

`x`	An object of class `eyeris_sim_params` from `sim_params()`
`...`	Unused; included for S3 consistency

Value

`x`, invisibly

`process_chunked_query` *Process large database query in chunks*

Description

Handles really large databases by processing queries in reasonably sized chunks to avoid memory issues. Data can be written to CSV or Parquet files as it's processed.

Usage

```
process_chunked_query(
  con,
  query,
  chunk_size = 1e+06,
  output_file = NULL,
  process_chunk = NULL,
  verbose = TRUE
)
```

Arguments

con	Database connection
query	SQL query string to execute
chunk_size	Number of rows to fetch per chunk (default: 1000000)
output_file	Optional output file path for writing chunks. If provided, chunks will be appended to this file. File format determined by extension (.csv or .parquet)
process_chunk	Optional function to process each chunk. Function should accept a data.frame and return logical indicating success. If not provided and output_file is specified, chunks are written to file.
verbose	Whether to print progress messages (default: TRUE)

Value

List containing summary information about the chunked processing

Examples

```
## Not run:
# These examples require an existing eyeris database

con <- eyeris_db_connect("/path/to/bids", "my-project")

# Process large query and write to CSV
process_chunked_query(
  con,
  "SELECT * FROM large_table WHERE condition = 'something'",
  chunk_size = 50000,
  output_file = "large_export.csv"
)

# Process large query with custom chunk processing
process_chunked_query(
  con,
  "SELECT * FROM large_table",
  chunk_size = 25000,
  process_chunk = function(chunk) {
    # Custom processing here
    processed_data <- some_analysis(chunk)
    return(TRUE)
  }
)

eyeris_db_disconnect(con)

## End(Not run)
```

`read_eyeris_parquet` *Read parquet files back into R*

Description

Convenience function to read the parquet files created by eyeris_db_to_parquet back into a single data frame or list of data frames by data type.

Usage

```
read_eyeris_parquet(
  parquet_dir,
  db_name = NULL,
  data_type = NULL,
  return_list = FALSE,
  pattern = "*.parquet",
  verbose = TRUE
)
```

Arguments

`parquet_dir`	Directory containing the parquet files, or path to database-specific folder
`db_name`	Optional database name to read from (if parquet_dir contains multiple database folders)
`data_type`	Optional data type to read (if NULL, reads all data types)
`return_list`	Whether to return a list by data type (TRUE) or combined data frame (FALSE, default)
`pattern`	Pattern to match parquet files (default: "*.parquet")
`verbose`	Whether to print progress messages (default: TRUE)

Value

Combined data frame from all parquet files, or list of data frames by data type

Examples

```
# Minimal self-contained example that avoids database creation
if (requireNamespace("arrow", quietly = TRUE)) {
  # create a temporary folder structure: parquet/<db_name>
  base_dir <- file.path(tempdir(), "derivatives", "parquet")
  db_name <- "example-db"
  dir.create(file.path(base_dir, db_name), recursive = TRUE, showWarnings = FALSE)

  # write two small parquet parts for a single data type
  part1 <- data.frame(time = 1:5, value = 1:5)
  part2 <- data.frame(time = 6:10, value = 6:10)
  arrow::write_parquet(
```

```
    part1,
    file.path(
      base_dir, db_name, paste0(db_name, "_timeseries_part-01-of-02.parquet")
    )
  )
  arrow::write_parquet(
    part2,
    file.path(
      base_dir, db_name, paste0(db_name, "_timeseries_part-02-of-02.parquet")
    )
  )

  # read them back as combined data frame
  data <- read_eyeris_parquet(base_dir, db_name = db_name)

  # read as list by data type
  data_by_type <- read_eyeris_parquet(base_dir, db_name = db_name, return_list = TRUE)

  # read specific data type only
  timeseries_data <- read_eyeris_parquet(base_dir, db_name = db_name, data_type = "timeseries")
}
```

`resample` *Resample an irregularly-sampled pupil timeseries onto a uniform grid*

Description

Places each recording segment onto the expected uniform (consistently-spaced) sampling grid: local sub-period timing jitter is smoothed by linear interpolation, and longer-than-expected gaps (dropped samples) become explicit `NA` rows for the later `interpolate()` step to handle.

Usage

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

Details

Most of the eyeris pipeline (e.g., detransient(), lpfilt(), downsample()) assumes a fixed sampling rate. EyeLink trackers honor that assumption by zero-filling missing pupil samples, but some hardware instead *drops* samples entirely when pupil data is missing, leaving holes in the otherwise evenly-spaced time vector. Those holes silently distort any rate-dependent step.

resample() repairs the **time axis** in two stages. Whether a block needs repair is decided by the robust check_uniform_sampling_intervals() detector, which distinguishes genuine dropped samples from data that only *looks* irregular – notably high-rate trackers that report integer-millisecond timestamps for sub-millisecond samples (these are left untouched, so genuine samples are never collapsed). For blocks it does repair:

1. **Build the target grid.** The uniform grid is anchored on the first *reliable* regular interval – the first observed interval that matches the expected sampling period – rather than on the first timestamp. This keeps early sub-period jitter (e.g., intervals of 3, 3, 4, 4 ms at a 4 ms period) from offsetting the whole grid. The grid is then extended in both directions at the expected period so that it spans every observed timestamp, including any samples that precede the anchor (back-extension).
2. **Resample onto the grid.** Observed samples are placed on the grid by linear interpolation: samples that land on a grid point are kept verbatim, and short/jittered intervals are interpolated across so their values contribute to the regular grid. Any observed interval longer than the expected period is treated as a real gap: the missing grid sample(s) inside it are inserted as NA rather than interpolated across, and flagged in a new logical is_resampled column so they can be tracked downstream.

resample() does **not** fill the inserted NA values itself; that is the job of interpolate(), which decides how much of a missing span to fill according to its own missing-data policy. Running resample() therefore turns the "dropped-sample" problem into the ordinary "missing-value" (NA) problem that the rest of the pipeline already handles.

For data that is already uniformly sampled (e.g., EyeLink), resample() is a guaranteed no-op: no rows are inserted, no is_resampled column is added, and the data is returned unchanged. The same is true of a block whose surviving samples already form a uniform (coarser) grid – e.g., pure systematic decimation – where there is nothing to insert without fabricating data.

Value

An eyeris object whose timeseries blocks have been placed on a uniform time grid, with a new logical is_resampled column marking inserted (gap) rows.

Note

This step is part of the glassbox() preprocessing pipeline and runs **automatically by default** (it is a no-op unless irregular sampling is detected). Opt out with glassbox(resample = FALSE). Advanced users may call it directly if needed.

See Also

interpolate() for filling the gaps left by dropped samples, and glassbox() for the recommended way to run this step as part of the full eyeris glassbox preprocessing pipeline.

`simulate_eyeris` *Generate a synthetic* `eyeris` *object with realistic pupil characteristics*

Description

Simulates a pupil time series whose statistical and morphological characteristics resemble real EyeLink recordings — a tonic baseline with slow drift and hippus, canonical task-evoked phasic dilations, blinks with partial-occlusion flank spikes, isolated transient artifacts, measurement noise, and optional high-frequency line noise — and wraps it in a fully valid S3 `eyeris` object.

Because the returned object is byte-compatible with the output of `load_asc()`, it flows unchanged through the entire `eyeris` pipeline: `deblink()`, `detransient()`, `interpolate()`, `lpfilt()`, `downsample()`, `detrend()`, `zscore()`, `plot.eyeris()`, `epoch()`, and `summarize_confounds()`.

Usage

```
simulate_eyeris(seed = 1L, params = sim_params(), block = 1L, verbose = TRUE)
```

Arguments

`seed`	Integer random seed for reproducible generation. Defaults to `1`
`params`	A parameter list from `sim_params()`. Defaults to `sim_params()`
`block`	Numeric block label written into the time series. Defaults to `1`
`verbose`	Logical; print a short status message. Defaults to `TRUE`

Details

The signal is generated deterministically given `seed`: the same `seed` and `params` always yield an identical object, and the global random number generator state is left untouched (generation is confined via `withr::with_seed()`). See `sim_params()` for the full generative model and every tunable component.

The clean latent signal (before artifacts and noise) and the indices of the injected artifacts are attached to the returned object as the attribute `"sim_truth"` for use as ground truth in quantitative demonstrations; note that this attribute does **not** survive the pipeline step functions (which rebuild the time series data frame), so downstream tooling recomputes ground truth from a retained copy rather than relying on the attribute.

Value

An object of S3 class `eyeris` (see the *Anatomy of an* `eyeris` *Object* vignette — `vignette("anatomy", package = "eyeris")`), with a `"sim_truth"` attribute describing the ground-truth signal.

See Also

`sim_params()` to configure the synthetic signal.

Examples

```
# generate a synthetic recording and run it through the pipeline
sim <- simulate_eyeris(seed = 1, params = sim_params(duration_secs = 20))


out <- sim |>
  eyeris::deblink() |>
  eyeris::detransient() |>
  eyeris::interpolate() |>
  eyeris::lpfilt() |>
  eyeris::zscore()

pdf(tempfile(fileext = ".pdf"))
plot(out, seed = 1)
dev.off()
```

`sim_params` *Configure the synthetic pupil signal model*

Description

Builds the parameter list that controls `simulate_eyeris()`. Every component of the synthetic pupil time series (tonic baseline, slow drift, hippus, task-evoked phasic responses, blinks, transient spikes, measurement noise, and optional line noise) is independently toggle-able and parameterized here, so a single scenario can isolate exactly the feature it needs to teach.

Usage

```
sim_params(
  fs = 1000L,
  duration_secs = 60,
  baseline = TRUE,
  baseline_mean = 5000,
  drift = FALSE,
  drift_slope = 8,
  drift_curv = 0,
  hippus = TRUE,
  hippus_amp = 40,
  hippus_freq = 0.12,
  phasic = TRUE,
  phasic_n = 10.1,
  phasic_tmax = 0.93,
  phasic_amp = c(120, 300),
  phasic_isi = c(6, 8),
  phasic_onsets_ms = NULL,
  blinks = TRUE,
```

```
  n_blinks = 6,
  blink_dur_ms = c(100, 300),
  blink_flank_ms = 40,
  blink_depth = 400,
  blink_overshoot = 300,
  blink_flank_shape = c("linear", "cosine"),
  transients = TRUE,
  n_transients = 3,
  transient_amp = c(200, 350),
  transient_width_ms = 1.5,
  noise = TRUE,
  noise_sd = 3,
  noise_ar = 0,
  line = FALSE,
  line_freq = 96,
  line_amp = 20,
  dropout_frac = 0.003,
  dropout_run_ms = c(1, 5),
  clip = c(3200, 7200),
  eye = c("R", "L"),
  pupil_dtype = c("DIAMETER", "AREA")
)
```

Arguments

`fs`	Sampling rate in Hz. Must divide 1000 evenly (e.g. `1000`, `500`, `250`) so sample timestamps remain integer milliseconds. Defaults to `1000`
`duration_secs`	Recording duration in seconds. Defaults to `60`
`baseline`	Logical; include the tonic baseline offset. Defaults to `TRUE`
`baseline_mean`	Tonic baseline pupil size in a.u. Defaults to `5000`
`drift`	Logical; include a slow linear drift. Defaults to `FALSE`
`drift_slope`	Drift slope magnitude in a.u. per second (applied as a decline). Defaults to `8` (i.e. a 480 a.u. decline over 60 s)
`drift_curv`	Quadratic drift coefficient in a.u. per second squared. Defaults to `0`
`hippus`	Logical; include the slow hippus oscillation. Defaults to `TRUE`
`hippus_amp`	Hippus amplitude in a.u. Defaults to `40`
`hippus_freq`	Hippus frequency in Hz. Defaults to `0.12`
`phasic`	Logical; include task-evoked phasic responses. Defaults to `TRUE`
`phasic_n`	Erlang shape parameter (unitless). Defaults to `10.1`
`phasic_tmax`	Time-to-peak of the phasic response in seconds. Defaults to `0.930`
`phasic_amp`	Length-2 numeric `c(min, max)`; peak phasic amplitudes are drawn uniformly from this range (a.u.). Defaults to `c(120, 300)`
`phasic_isi`	Length-2 numeric `c(min, max)`; inter-stimulus intervals (seconds) are drawn uniformly from this range when `phasic_onsets_ms` is `NULL`. Defaults to `c(6, 8)`

phasic_onsets_ms
: Optional numeric vector of explicit stimulus onset times in milliseconds. When NULL (default), onsets are generated from phasic_isi

blinks
: Logical; include blink artifacts. Defaults to TRUE

n_blinks
: Number of blinks. Defaults to 6

blink_dur_ms
: Length-2 numeric c(min, max); blink core (missing-data) durations in milliseconds are drawn uniformly from this range. Defaults to c(100, 300)

blink_flank_ms
: Duration in milliseconds of the occlusion spike on each side of a blink core. Keep below the deblink() extend value (default 50 ms) so deblinking fully removes it. Defaults to 40

blink_depth
: Depth in a.u. of the leading occlusion dip. Defaults to 400

blink_overshoot
: Height in a.u. of the trailing recovery overshoot. Defaults to 300

blink_flank_shape
: Shape of the occlusion ramp: "linear" or "cosine". Defaults to "linear"

transients
: Logical; include isolated transient spikes. Defaults to TRUE

n_transients
: Number of transient spikes. Defaults to 3

transient_amp
: Length-2 numeric c(min, max); transient amplitudes are drawn uniformly from this range (a.u.) with random sign. Defaults to c(200, 350)

transient_width_ms
: Gaussian width (standard deviation) of each transient in milliseconds. Defaults to 1.5

noise
: Logical; include broadband measurement noise. Should remain TRUE for any signal that will be passed to detransient(). Defaults to TRUE

noise_sd
: Standard deviation of the white measurement noise in a.u. Defaults to 3

noise_ar
: Lag-1 autocorrelation for optional AR(1) noise coloring in [0, 1). Defaults to 0 (white noise)

line
: Logical; include a high-frequency line-noise tone. Defaults to FALSE

line_freq
: Line-noise frequency in Hz. Defaults to 96

line_amp
: Line-noise amplitude in a.u. Defaults to 20

dropout_frac
: Fraction of (non-blink) samples to mark missing as short scattered dropout runs. Defaults to 0.003

dropout_run_ms
: Length-2 numeric c(min, max); dropout run lengths in milliseconds. Defaults to c(1, 5)

clip
: Length-2 numeric c(min, max); observed (non-missing) samples are clamped to this a.u. range. Defaults to c(3200, 7200)

eye
: Which eye to label the data as ("L" or "R"). Defaults to "R"

pupil_dtype
: Pupil data type label ("DIAMETER" or "AREA"). Defaults to "DIAMETER"

Details

The synthetic signal is composed additively in arbitrary units (a.u.) matched to real EyeLink **pupil diameter** data (which typically ranges from roughly 3600–7000 a.u.):

$$L(t) = B_0 + \text{drift}(t) + \text{hippus}(t) + \sum_k \text{phasic}_k(t)$$

$$\text{pupil}(t) = L(t) + \sum_j \text{transient}_j(t) + \text{line}(t) + \text{noise}(t)$$

after which blink and dropout artifacts are stamped on (see `simulate_eyeris()`).

Components (each independently toggle-able):

- **Tonic baseline** (`baseline`): a constant offset `baseline_mean`.
- **Linear drift** (`drift`): a slow tonic decline $-\text{drift\_slope} \cdot t$ (a.u. per second), plus an optional quadratic term `drift_curv`. This is precisely what `detrend()` removes.
- **Hippus** (`hippus`): a slow arousal oscillation $A_h \sin(2\pi f_h t + \phi)$ that sits below the low-pass cutoff and therefore survives `lpfilt()`.
- **Phasic response** (`phasic`): the canonical task-evoked pupil response modeled with the Hoeks & Levelt (1993) Erlang gamma kernel $h(\tau) = \tau^n e^{-n\tau/t_{max}}$, peak-normalized and scaled by `phasic_amp`, time-locked to stimulus onsets.
- **Blinks** (`blinks`): runs of **missing data** (`NA`) flanked by the rapid partial-occlusion down-then-up spikes that surround real blinks — the artifact `deblink()` is designed to remove.
- **Transients** (`transients`): isolated, physiologically implausible fast spikes (tracker glitches) that `detransient()` targets.
- **Noise** (`noise`): broadband white measurement noise. Always keep `noise_sd > 0` — a perfectly noiseless signal makes the median absolute deviation of the pupil speed zero, which aborts `detransient()`.
- **Line noise** (`line`): an optional high-frequency tone used to demonstrate aliasing when decimating without an anti-alias filter.

Value

A named list of class `eyeris_sim_params`.

See Also

`simulate_eyeris()` to generate data from these parameters.

Examples

```
# default parameters
p <- sim_params()

# a short recording with a strong linear drift for a detrend demo
p2 <- sim_params(duration_secs = 20, drift = TRUE, drift_slope = 12)
```

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 time series 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:

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

Missing data is reported as both a count (`n_missing`) and a proportion (`prop_missing`, `0` to `1`; multiply by `100` for a percentage) of `NA` samples. It is computed at two levels so you can choose the granularity appropriate to your design:

- **Block level**: `confounds$unepoched_timeseries[[block]][[step]]` reports `prop_missing` across the entire recording block.
- **Epoch/trial level**: `confounds$epoched_timeseries[[epoch]][[block]]` reports `prop_missing` for each epoched event (e.g., per trial).

`eyeris` intentionally does not enforce a fixed missing-data exclusion cutoff. Instead, `prop_missing` is exposed so you can define your own exclusion thresholds at whichever level (trial, epoch, or block) is appropriate for your study.

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 time series for a specific block and step
confounds$confounds$unepoched_timeseries

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

`zscore` *Z-score pupil time series data*

Description

The intended use of this method is to scale the arbitrary units of the pupil size time series 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 time series from each outcome variable group, separately). Instead, you should compute the z-score on the entire pupil time series (before epoching the data), and then split and take the mean of the z-scored time series as a function of condition variable.

Value

An eyeris object with a new column in time series: 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

For a complete, end-to-end reference pipeline that demonstrates how all eyeris preprocessing functions are chained together in practice, see the "Building Blocks Under the Hood" section of the *Anatomy of an* eyeris *Object* vignette — vignette("anatomy", package = "eyeris") — as well as the *Complete Pupillometry Pipeline Walkthrough* vignette: vignette("complete-pipeline", package = "eyeris").

Examples

```
demo_data <- eyelink_asc_demo_dataset()

demo_data |>
  eyeris::glassbox(zscore = TRUE) |> # set to FALSE to skip (not recommended)
  # `preview_window` zooms in on a 2-second subset of the time series
  plot(seed = 0, preview_window = c(10, 12))
```

Index