

Package: GeoPressureR (via r-universe)

October 8, 2026

Title Global Positioning by Atmospheric Pressure

Version 3.6.3

Description R package to determine the position and trajectory of a bird based on light-weight data-logger measuring at lease atmospheric pressure.

License GPL (≥ 3)

URL <https://github.com/GeoPressure/GeoPressureR>,
<https://geopressure.org/GeoPressureR/>

BugReports <https://github.com/GeoPressure/GeoPressureR/issues>

Depends R ($\geq 4.1.0$)

Imports assertthat, cli, config, EBImage, future, future.apply, ggplot2, glue, httr2 ($\geq 1.0.0$), jsonlite, keyring, leaflet ($\geq 2.2.0$), lifecycle, Matrix, methods, plotly, pracma, progressr, RColorBrewer, stats, suntools, terra, utils

Suggests callr, BiocManager, covr, ecmwfr ($\geq 2.0.0$), knitr, leaflet.providers, maps, rnatualearth, rnatualearthdata, ncd4, pkgload, Rarr, readxl, shiny, shinyjs, shinyWidgets, testthat ($\geq 3.0.0$), withr

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

Language en-GB

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libabsl-dev cmake libfftw3-dev libgdal-dev gdal-bin libgeos-dev make libicu-dev libjpeg-dev libpng-dev libsecret-1-dev libtiff-dev libuv1-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev

Repository <https://ropensci-staging.r-universe.dev>

Date/Publication 2026-10-08 11:46:40 UTC

RemoteUrl <https://github.com/Rafnuss/GeoPressureR>

RemoteRef HEAD

RemoteSha baa54ec5feedc91e5e66e03cd4f0143855178b6f

Contents

bird_create	3
edge_add_wind	4
geolight_fit_location	6
geolight_map	8
geolight_solar	11
geolightviz	13
geopressure_map	15
geopressure_map_preprocess	19
geopressure_timeseries	20
GeoPressureR	24
geopressuretemplate	25
geopressureviz	28
graph_add_wind	30
graph_assert	31
graph_create	32
graph_marginal	34
graph_most_likely	35
graph_set_movement	36
graph_simulation	39
load_interim	40
map_add_mask_water	41
map_create	42
map_expand	44
param_create	45
path2edge	46
path2elevation	47
path2twilight	49
plot.map	50
plot.tag	52
plot_graph_movement	54
plot_path	55
plot_pressurepath	57
plot_tag_acceleration	58
plot_tag_actogram	59
plot_tag_light	60
plot_tag_pressure	61
plot_tag_temperature	62
plot_tag_twilight	63
plot_twl_calib	64
pressurepath_create	65
pressurepath_variable_available	69
print.bird	70
print.graph	71
print.map	72
print.param	73
print.tag	73
rast.map	74

read_stap	75
speed2bearing	76
speed2prob	77
stap2duration	78
stap2flight	79
tag_assert	80
tag_create	81
tag_download_wind	85
tag_label	87
tag_label_auto	88
tag_label_read	90
tag_label_stap	91
tag_label_write	92
tag_set_map	93
tag_stap_daily	95
tag2path	96
trainset	98
ts2mat	99
twilight_create	100
twilight_label_read	102
twilight_label_write	103
windsupport	104
Index	105

bird_create	<i>Create bird flight traits</i>
--------------------	----------------------------------

Description

This function returns a list with the four morphological information necessary to construct the power curve: mass, wing span, wing aspect ratio, and body frontal area.

When any of these variables are missing, we query the AVONET database ([doi:10.6084/m9.figshare.16586228.v5](https://doi.org/10.6084/m9.figshare.16586228.v5)) using the scientific name from [the Clements Checklist](#).

Usage

```
bird_create(
  scientific_name,
  mass = NULL,
  wing_span = NULL,
  wing_aspect = NULL,
  wing_area = NULL,
  body_frontal_area = NULL,
  species_name = lifecycle::deprecated()
)
```

Arguments

<code>scientific_name</code>	Scientific name of the species
<code>mass</code>	Mass of the bird in kilograms
<code>wing_span</code>	Wing span in meters
<code>wing_aspect</code>	Wing aspect ratio (no unit)
<code>wing_area</code>	Wing area in meter square
<code>body_frontal_area</code>	Body frontal area in meter square
<code>species_name</code>	[Deprecated] Use <code>scientific_name</code> instead

Value

List containing mass, wing span, wing aspect ratio, and body frontal area.

See Also

Other bird: `print.bird()`

Other movement: `graph_set_movement()`, `plot_graph_movement()`, `speed2prob()`, `tag_download_wind()`

Examples

```
# Using AVONET dataset
bird_create("Acrocephalus arundinaceus")

# Using AVONET dataset + custom values
bird_create("Acrocephalus arundinaceus", wing_aspect = 8)

# Import your own bird. You will need mass, and at least two of wing_span,
# wing_aspect or wing_area.
bird_create("Madynuss nutshell", mass = 8, wing_span = 0.2, wing_aspect = 4)
```

edge_add_wind

Retrieve ERA5 variable along edge

Description

Reads the NetCDF files and extracts the variable requested along each flight defined by the edges.

- Time: linear interpolation using the resolution requested with `rounding_interval`
- Space: nearest neighbour interpolation by default, or bi-linear with `pracma::interp2` if `interp_spatial_linear=TRUE`. Note: spatial interpolation is limited to 0.1° for computational reasons.
- Pressure/altitude: linear interpolation using the exact `pressure` values

Usage

```

edge_add_wind(
  graph,
  edge_s,
  edge_t,
  pressure = NULL,
  variable = c("u_component_of_wind", "v_component_of_wind"),
  rounding_interval = 60,
  interp_spatial_linear = FALSE,
  return_averaged_variable = lifecycle::deprecated(),
  file = function(stap_id, tag_id) {

    glue::glue("./data/wind/{tag_id}/{tag_id}_{stap_id}.nc")
  },
  quiet = FALSE
)

```

Arguments

graph	either a tag or a graph GeoPressureR object.
edge_s	a index of the source node of the edge. Either a vector with 3D index or a matrix of 3 columns, one for each dimension.
edge_t	a index of the target node of the edge. Either a vector with 3D index or a matrix of 3 columns, one for each dimension.
pressure	pressure measurement of the associated tag data used to estimate the pressure level (i.e., altitude) of the bird during the flights. This data.frame needs to contain date as POSIXt and value in hPa. If not provided, uses graph\$pressure , assuming that argument graph is a GeoPressureR tag object.
variable	variables to extract, named exactly as in tag_download_wind() — the files must of course contain them. The NetCDF stores its arrays under GRIB short names ("u" for "u_component_of_wind") and this function translates; passing those short names still works but is deprecated.
rounding_interval	temporal resolution on which to query the variable (min). Default is to match ERA5 native resolution (1hr).
interp_spatial_linear	logical to interpolate the variable linearly over space, if FALSE takes the nearest neighbour. ERA5 native resolution is 0.25°
return_averaged_variable	[Deprecated] No longer used.
file	absolute or relative path of the ERA5 wind data file to be downloaded. Function taking as arguments (1) the stationary period identifier and (2) the tag_id.
quiet	logical to hide messages about the progress

Value

A data.frame with one row per time step, edge, and variable:

- `edge_id` edge index
- `val` value of the variable at each time step
- `pressure` pressure at each time step
- `date` datetime of each time step
- `w` weight for averaging
- `var` variable name, as passed to `variable`
- `lat` latitude at each time step
- `lon` longitude at each time step

See Also

[GeoPressureManual](#)

Examples

```
## Not run:
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    tag_set_map(extent = c(-16, 23, 0, 50), scale = 1) |>
    geopressure_map(quiet = TRUE)
})
graph <- graph_create(tag, quiet = TRUE)
wind <- edge_add_wind(graph, edge_s = graph$s, edge_t = graph$t, quiet = TRUE)

## End(Not run)
```

`geolight_fit_location`

Estimate one location per stationary period from twilight times

Description

For each stationary period (`stap_id`) with usable twilight events, estimate a single longitude, latitude, and effective zenith angle by minimizing the squared spherical distance to the corresponding iso-zenith small circles.

Each twilight defines a small circle on the Earth: the set of locations at a fixed angular distance (zenith angle) from the subsolar point at that time. The fitted location is the point that is, in a least-squares sense, closest to all those circles simultaneously.

For a stationary period with twilight-specific subsolar points `s_i`, the fitted longitude `lon`, latitude `lat`, and zenith angle `z` minimize

$$\text{mean}_i (d((\text{lon}, \text{lat}), s_i) - z)^2 + P_{\text{prior}}(z)$$

where $d(\text{lon}, \text{lat}), \text{s}_i$) is the great-circle angular distance from the candidate location to the subsolar point at twilight i . The first term is the mean squared residual to the iso-zenith small circles, normalized by the number of twilight events so its influence does not depend on sample size.

$P_{\text{prior}}(z)$ is an optional Gaussian penalty on the fitted zenith angle:

$$1/2 * ((z - \mu) / \sigma)^2$$

where $\mu = \text{zenith_prior_mean}$ and $\sigma = \text{zenith_prior_sd}$. This acts as a prior on the effective zenith angle while leaving longitude and latitude driven by the twilight geometry.

Usage

```
geolight_fit_location(
  tag,
  fitted_location_duration = Inf,
  extent = NULL,
  zenith_bounds = c(70, 110),
  zenith_prior_mean = 93,
  zenith_prior_sd = 1.3,
  zenith_prior_penalty_weight = 1e-05,
  compute_known = FALSE,
  quiet = FALSE
)
```

Arguments

tag	A GeoPressureR tag object containing <code>tag\$twilight</code> and <code>tag\$stap</code> .
fitted_location_duration	Minimum duration (in days) of stationary period(s) eligible to be used as a fitted location from twilight times. Default is <code>Inf</code> (disabled).
extent	Numeric vector <code>c(W, E, S, N)</code> in degrees (longitude west/east, latitude south/north).
zenith_bounds	Numeric vector of length 2 giving zenith bounds (degrees).
zenith_prior_mean	Gaussian prior mean for fitted zenith angle (degrees).
zenith_prior_sd	Gaussian prior standard deviation for fitted zenith angle (degrees).
zenith_prior_penalty_weight	Weight controlling how strongly automatically fitted locations are pulled toward <code>zenith_prior_mean</code> . This only affects fitted locations; known locations are still used directly when <code>compute_known = FALSE</code> .
compute_known	Logical; if <code>FALSE</code> , known stationary periods are copied from <code>tag\$stap</code> (rather than being estimated).
quiet	Logical; if <code>TRUE</code> , suppress informative messages.

Value

A path data.frame derived from `tag$stap` with added columns: `lon`, `lat`, `zenith`. Rows where fitting was not possible remain NA.

See Also

Other geolight: [geolight_map\(\)](#), [geolight_solar\(\)](#), [plot_twl_calib\(\)](#), [twilight_create\(\)](#), [twilight_label_read\(\)](#), [twilight_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read()
})
tag <- tag_stap_daily(tag, quiet = TRUE)
path <- geolight_fit_location(tag, fitted_location_duration = 5, quiet = TRUE)
```

geolight_map	<i>Compute likelihood map from twilight</i>
--------------	---

Description

This function estimates a likelihood map for each stationary period based on twilight data. The function performs the following steps:

1. Calibrate twilight errors from true known locations and/or automatically fitted calibration locations. See below for details.
2. Compute a likelihood map for each twilight using the calibration.
3. Combine the likelihood maps of all twilights belonging to the same stationary periods with a log-linear pooling. See [GeoPressureManual | Probability aggregation](#) for more information on probability aggregation using log-linear pooling.

Usage

```
geolight_map(
  tag,
  twl_calib_adjust = 1.4,
  fitted_location_duration = Inf,
  zenith_prior_mean = 93,
  zenith_prior_sd = 1.3,
  zenith_prior_penalty_weight = 1e-05,
  refine_fitted_location_scale = 20,
  refine_fitted_location_max_iter = 2,
  twl_llp = function(n) log(n)/n,
  compute_known = FALSE,
  keep_twl = FALSE,
  quiet = FALSE
)
```

```

geolight_map_aggregate(
  tag,
  compute_known = FALSE,
  twl_llp = function(n) log(n)/n,
  keep_twl = FALSE,
  quiet = FALSE
)

geolight_map_calibrate(
  tag,
  twl_calib_adjust = 1.4,
  fitted_location_duration = Inf,
  zenith_prior_mean = 93,
  zenith_prior_sd = 1.3,
  zenith_prior_penalty_weight = 1e-05,
  refine_fitted_location_scale = 20,
  refine_fitted_location_max_iter = 2,
  quiet = FALSE
)

geolight_map_likelihood(tag, compute_known = FALSE, quiet = FALSE)

```

Arguments

tag a GeoPressureR tag object.

twl_calib_adjust smoothing parameter for the kernel density (see `stats::density()`).

fitted_location_duration Minimum duration (in days) of stationary period(s) eligible to be used as an automatic (fitted) calibration site estimated from twilight times. If enabled (finite duration), eligible stationary periods without known coordinates are fitted from twilight times, then used like known locations for calibration. Default is `Inf` (disabled).

zenith_prior_mean Gaussian prior mean for the fitted zenith angle (degrees).

zenith_prior_sd Gaussian prior standard deviation for the fitted zenith angle (degrees).

zenith_prior_penalty_weight Weight controlling how strongly automatically fitted calibration locations are pulled toward **zenith_prior_mean**.

refine_fitted_location_scale Target spatial resolution, in kilometers, for fitted calibration-anchor refinement. This also defines the movement threshold for convergence.

refine_fitted_location_max_iter Maximum number of fitted-location refinement iterations. Use 0 to disable refinement. In practice, 2 iterations is usually sufficient.

<code>twl_llp</code>	log-linear pooling aggregation weight.
<code>compute_known</code>	logical defining if the map(s) for known stationary period should be estimated based on twilight or hard defined by the known location <code>stap\$known_1**</code>
<code>keep_twl</code>	logical defining if the likelihood map of each twilight is retained.
<code>quiet</code>	logical to hide messages about the progress

Value

a tag with the likelihood of light as `tag$map_light`

Calibration

Calibration can be based on:

- a known location for at least one stationary period (set via `tag_set_map(known = ...)`), and/or
- an automatically fitted calibration location (enabled via `fitted_location_duration`).

When `fitted_location_duration` is finite, stationary periods at least as long as this threshold and without known coordinates can be fitted from twilight geometry and used as calibration locations. If known locations are also present, the calibration combines the known and fitted locations. The fitted-location step uses a weak Gaussian prior on the effective zenith angle, controlled by `zenith_prior_mean`, `zenith_prior_sd`, and `zenith_prior_penalty_weight`.

After the initial fitted-location estimate, `refine_fitted_location_max_iter > 0` optionally refines automatically fitted calibration anchors by iteratively recalibrating the twilight-error distribution for the same stationary period and moving the anchor to the local light-likelihood mode. This is a calibration-anchor refinement, not a separate movement estimator. It is only applied to fitted calibration locations; true known locations are kept fixed. Refined fitted anchors have `zenith` set to NA because the initial fitted zenith no longer corresponds to the refined coordinates. The default is 2 iterations.

Instead of calibrating the twilight errors in terms of duration, we directly model the zenith angle error. We use a kernel distribution to fit the zenith angle during the known stationary period(s). The `twl_calib_adjust` parameter allows to manually adjust how smooth you want the fit of the zenith angle to be. Because the zenith angle error model is fitted with data from the calibration site only, and we are using it for all locations of the bird's journey, it is safer to assume a broader/smoothier distribution.

See Also

Other geolight: `geolight_fit_location()`, `geolight_solar()`, `plot_twl_calib()`, `twilight_create()`, `twilight_label_read()`, `twilight_label_write()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  # Read geolocator data and set map parameters
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    tag_set_map(
```

```

    extent = c(-16, 23, 0, 50),
    scale = 10,
    known = data.frame(
      stap_id = 1,
      known_lon = 17.05,
      known_lat = 48.9
    )
  ) |>
  twilight_create() |>
  twilight_label_read()
})

# Visualize twilights
plot(tag, type = "twilight")

# Calibrate twilight to zenith angle and visualize
tag <- geolight_map_calibrate(tag)
plot_twl_calib(tag)

# Compute likelihood for each twilight and visualize
tag <- geolight_map_likelihood(tag)
plot(tag, type = "map_light_twl")

# Aggregate twilights to stationary periods and visualize
tag <- geolight_map_aggregate(tag, quiet = TRUE)
plot(tag, type = "map_light")

```

geolight_solar

Solar zenith angle using NOAA / Meeus equations

Description

Compute solar zenith angles from time and location using the formulation implemented in the NOAA Solar Calculator, itself based on the astronomical algorithms of Meeus.

Usage

```
geolight_solar(date, lat, lon)
```

Arguments

<code>date</code>	POSIXct vector of times.
<code>lat</code>	Numeric vector of latitudes (degrees north).
<code>lon</code>	Numeric vector of longitudes (degrees east).

Details

The computation is split internally into three steps:

1. `geolight_solar_constants()`: Time-only solar quantities (solar time and declination)
2. `geolight_solar_zenith()`: Geometric solar zenith angle
3. `geolight_solar_refracted()`: Approximate atmospheric refraction correction

Conventions:

- `lon` is expressed in **degrees east** (positive eastward).
- `lat` is expressed in **degrees north**.
- All angles are in **degrees** unless stated otherwise.
- `date` is provided as POSIXct and internally converted to Julian day.

When `lat` and `lon` are vectors, the result is a 3-D array with dimensions:

```
c(length(lat), length(lon), length(date))
```

When `lat` and `lon` are scalars, the result reduces to a vector of length `length(date)`.

Solar time and hour angle:

Internally, the solar time is computed as NOAA's *true solar time* expressed as an angle (degrees) at Greenwich, without longitude adjustment. Longitude enters later through the hour angle:

```
hour_angle = solar_time + lon - 180
```

Values are intentionally not wrapped to `[0, 360)` because all downstream trigonometric functions are periodic.

Atmospheric refraction:

Atmospheric refraction is applied using NOAA's standard piecewise approximation as a function of solar elevation. This correction is approximate and does not account for local pressure or temperature.

Value

Solar zenith angle(s) in degrees, corrected for atmospheric refraction.

References

NOAA Global Monitoring Laboratory. Solar Calculation Details. <https://gml.noaa.gov/grad/solcalc/calcdetails.html>

Meeus, J. (1998). *Astronomical Algorithms*. Willmann-Bell.

See Also

Other geolight: `geolight_fit_location()`, `geolight_map()`, `plot_twl_calib()`, `twilight_create()`, `twilight_label_read()`, `twilight_label_write()`

Examples

```
date <- as.POSIXct(
  c("2020-06-21 12:00:00", "2020-12-21 12:00:00"),
  tz = "UTC"
)
z <- geolight_solar(date, lat = 46, lon = 6)
z
```

geolightviz

Start the GeoLightViz Shiny app

Description

GeoLightViz is an interactive tool for reviewing twilight observations and preparing light-based geolocation analyses. It displays the light record and detected twilight events so that observations affected by shading or other light obstructions can be marked as discarded. It can also be used to define or edit stationary periods from twilight patterns, which is useful when pressure or acceleration data are unavailable. When a map has been defined with `tag_set_map()` and stationary periods are available, the app links the light plot to a map of twilight likelihoods. You can explore candidate locations and tune twilight calibration and likelihood aggregation settings with visual feedback. Twilight labels and stationary periods can be saved or exported from the app.

Usage

```
geolightviz(
  x,
  staphath = NULL,
  launch_browser = TRUE,
  run_bg = TRUE,
  quiet = FALSE,
  ...
)
```

Arguments

x	a GeoPressureR <code>tag</code> object, a <code>.Rdata</code> file or the unique identifier <code>id</code> with a <code>.Rdata</code> file located in <code>"./data/interim/{id}.RData"</code> .
staphath	optional stationary path data.frame (defaults to <code>tag\$stap</code> when available).
launch_browser	If true (by default), the app runs in your browser, otherwise it runs on Rstudio.
run_bg	If true (by default), the app runs in a background R process using <code>callr::r_bg()</code> , allowing you to continue using the R console. If false, the app blocks the console until closed.

<code>quiet</code>	logical, currently unused.
<code>...</code>	currently unused.

Details

See the [GeoLightViz chapter of the GeoPressureManual](#) for an overview of the app's controls and workflow.

Value

When `run_bg = TRUE`, an invisible callr `r_process` running the app. When `run_bg = FALSE`, the return value of `shiny::runApp()`.

Basemap API keys

Interactive maps always offer Esri basemaps (gray canvas, imagery and topographic). CARTO, Mapbox and MapTiler basemaps are added to the layer control only when their API key is stored in your system keyring. Get a key from each provider:

- CARTO: "Get your API key" on [CARTO Basemaps](#).
- Mapbox: [Access tokens](#) page of your account (see [Mapbox documentation](#)).
- MapTiler: [API keys](#) page of your MapTiler Cloud account (see [MapTiler documentation](#)).

Then store each key once:

```
keyring::key_set_with_value("CARTO_API_KEY", password = "<your CARTO API key>")
keyring::key_set_with_value("MAPBOX_ACCESS_TOKEN", password = "<your Mapbox token>")
keyring::key_set_with_value("MAPTILER_API_KEY", password = "<your MapTiler API key>")
```

Mapbox access tokens are visible in the browser when a map is rendered. Use a public, read-only token rather than a secret token.

See Also

[GeoPressureManual](#) | [GeoLightViz](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read()
})
geolightviz(tag, run_bg = FALSE, launch_browser = FALSE)
```

`geopressure_map`*Compute likelihood map from pressure data*

Description

The `geopressure_map()` function computes a likelihood map for each stationary period based on pressure measurements. It is a wrapper of the following two child functions:

1. `geopressure_map_mismatch()` computes the mismatch maps between the pressure sensor measurements and the ERA5 reanalysis database.
2. `geopressure_map_likelihood()` converts the mismatch maps into a likelihood map.

See below for details. For more background on the method behind these functions, please refer to Nussbaumer et al. (2023a; [doi:10.1111/2041210X.14043](https://doi.org/10.1111/2041210X.14043)).

Usage

```
geopressure_map(  
  tag,  
  max_sample = 250,  
  margin = 30,  
  sd = 1,  
  thr_mask = 0.9,  
  log_linear_pooling_weight = function(n) log(n)/n,  
  timeout = 300,  
  workers = "auto",  
  era5_dataset = "land",  
  keep_mask = FALSE,  
  keep_mse = FALSE,  
  compute_known = FALSE,  
  debug = FALSE,  
  quiet = FALSE  
)  
  
geopressure_map_likelihood(  
  tag,  
  sd = 1,  
  log_linear_pooling_weight = function(n) log(n)/n,  
  keep_mse = TRUE  
)  
  
geopressure_map_mismatch(  
  tag,  
  max_sample = 250,  
  margin = 30,  
  thr_mask = 0.9,  
  timeout = 300,
```

```

workers = "auto",
era5_dataset = "land",
compute_known = FALSE,
keep_mask = TRUE,
debug = FALSE,
quiet = FALSE
)

```

Arguments

tag	a GeoPressureR tag object
max_sample	the computation of the maps is only performed on max_sample datapoints of pressure to reduce computational time. The samples are randomly (uniformly) selected on the time series.
margin	the margin is used in the mask map to accept measurement errors, small-scale topography, and vertical movements of the bird (unit in meters, 1hPa~10m).
sd	standard deviation of the pressure error. Numeric of length 1 or number of stationary periods.
thr_mask	threshold of the percentage of data points outside the elevation range to be considered not possible.
log_linear_pooling_weight	weighting function of the log-linear pooling, taking the number of samples of the stationary periods used and returning the weight of the aggregation. See GeoPressureManual Probability aggregation for more details.
timeout	duration before the code is interrupted both for the request on GeoPressureAPI and on GEE (in seconds, see <code>httr2::req_timeout()</code>).
workers	number of parallel requests on GEE. Integer between 1 and 99. "auto" adjust the number of workers to the number of stapelev to query.
era5_dataset	ERA5 product: "land" (default) at 0.1 degree resolution, "single-levels" at 0.25 degree resolution, or "both" to use ERA5-Land over land and global ERA5 elsewhere. Unlike altitude in pressurepath_create() , "land" is right here: the pressure mismatch is differential (the mean error is removed at each stationary period), so the static ERA5-Land bias cancels and its finer resolution is what the map needs.
keep_mask	logical defining if the mask map is returned in tag .
keep_mse	logical defining if the MSE map is returned in tag .
compute_known	logical defining if the map(s) for known stationary period should be estimated based on twilight or hard defined by the known location <code>stap\$known_1**</code>
debug	logical to display additional information to debug a request
quiet	logical to hide messages about the progress

Details

A map will only be computed for the stationary periods included in `tag$stap$include` and without a known position `tag$stap$known_1**` as defined by `tag_set_map()`. At known stationary periods, the likelihood map is a map of 0s with a single 1 at the pixel closest to the known position.

Value

Returns the same GeoPressureR `tag` object including the GeoPressureR map object `tag$map_pressure` containing the likelihood map of each stationary period. See `map_create()` for details. If `keep_mask` and `keep_mse` are each true, `tag` also includes the `tag$map_pressure_mse` and `tag$map_pressure_mask` maps, as well as `tag$stap$nb_sample`, indicating the number of datapoints used to compute the MSE.

Mismatch map with GeoPressureAPI

`geopressure_map_mismatch()` computes the mismatch maps on Google Earth Engine via the map entry point of the [GeoPressure API](#). This consists of the following steps:

1. **Pre-process pressure:** the pressure measurements are first smoothed and downsampled to a 1-hour resolution in order to match ERA-5 resolution (see `geopressure_map_preprocess()`).
2. **Generate requests:** Send a single request to the GeoPressureAPI to generate the Google Earth Engine (GEE) URLs, one for each elevation/stationary period which can be used to compute the maps on the GEE server.
3. **Send the requests:** Call the URLs, which will start the computation on the GEE server. At this step, it does not wait for an answer, but sends the requests in parallel to be faster.
4. **Compute and download the maps:** When all requests are sent, we wait for the GEE server to return a geotiff file (map) for each elevation/ stationary period.
5. **Post-process maps:** Read these geotiff maps as matrix, create the corresponding Geo-PressureR map object with `create_map()`.

The following two maps are returned for each stationary period:

1. **map_pressure_mse:** The Mean Square Error (MSE) between the data logger pressure time series and the reanalysis. The mean error is removed because we assume no specific altitude of the geolocator, thus allowing an altitudinal shift of the pressure time series.
2. **map_pressure_mask** (optionally): The mask of the proportion of pressure measurements corresponding to altitude values found within the min and max ground elevation at each location. The altitude value of the geolocator pressure time series is computed with the barometric formula accounting for the temporal variation of pressure (surface-pressure) and temperature (2m-temperature) based on ERA5 data. The min and max ground elevation of each pixel is computed from SRTM-90. This map is only returned if `keep_mask` is TRUE.

For more details, read the [GeoPressure API documentation](#) .

Elevation levels

It is possible to indicate different elevation levels when the bird was spending time at locations with different elevations within a general area (~10km), and thus within the same stationary period. This can be done by using `tag$label="elev_x"` for all measurements of the same elevation level x . See more information on the labelling of elevation levels in [the corresponding section in the GeoPressureManual](#).

Behind the scene, each of these elevation levels produces a new request on the GeoPressureAPI. The mismatch maps of all elevation levels belonging to the same stationary periods are combined (as a weighted average) to result in a single mismatch map per stationary period.

Convert mismatch map into likelihood map

We convert the map of the mean square error MSE and altitude mask z_{mask} computed by `geopressure_map_mismatch()` into a likelihood map with,

$$L = \left(\frac{1}{2\pi\sigma^2} \right)^{nw/2} \exp \left(-wn \frac{\text{MSE}}{2\sigma^2} \right) [z_{\text{mask}} > T],$$

where σ is the standard deviation (`sd`) of pressure and T is the mask threshold (`thr_mask`).

Because the auto-correlation of the time series is not accounted for in this equation, we use a log-linear pooling weight of $w = \log(n)/n$ by default, where n is the number of samples in the time series (i.e., data points used to compute the MSE). See [GeoPressureManual | Probability aggregation](#) for details.

Important Note: Since GeoPressure v3.1.0, the threshold of the mask is happening directly on the GEE server (i.e., during `geopressure_map_mismatch()`). This allows to compute the MSE only for pixels which are within the threshold, thus reducing the computational cost significantly.

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, and Felix Liechti. 2023. Global Positioning with Animal-borne Pressure Sensors. *Methods in Ecology and Evolution*, 14, 1118–1129 [doi:10.1111/2041210X.14043](#).

See Also

[GeoPressureManual](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    scale = 4
  )
})
```

```

tag <- geopressure_map_mismatch(tag,
  max_sample = 50,
  margin = 20,
  thr_mask = 0.95,
  keep_mask = TRUE,
  quiet = TRUE
)

plot(tag, type = "map_pressure_mse", plot_leaflet = FALSE)

plot(tag, type = "map_pressure_mask", plot_leaflet = FALSE)

tag <- geopressure_map_likelihood(tag,
  sd = 1,
  log_linear_pooling_weight = function(n) log(n) / n
)

plot(tag, type = "map_pressure")

```

```
geopressure_map_preprocess
```

Prepare pressure data for `geopressure_map()`

Description

Performs the following pre-processing for the pressure data.frame:

1. Remove discarded and flight pressure measurements.
2. Group pressure measurements by stationary period and elevation level.
3. Smooth and downscale pressure to a 1 hour resolution to match ERA5 data.

This function is used within `geopressure_map_mismatch()` but can be useful to check the pressure data sent to the GeoPressureAPI.

Usage

```
geopressure_map_preprocess(tag, compute_known = FALSE)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object
<code>compute_known</code>	logical defining if the map(s) for known stationary period should be estimated based on twilight or hard defined by the known location <code>stap\$known_1**</code>

Value

Pressure data.frame without flight and discarded values, on a 1 hour resolution.

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

pressure_processed <- geopressure_map_preprocess(tag)
str(pressure_processed)

# Plotting pressure displays the raw data (in grey) with the processed data from
# `geopressure_map_preprocess` (as coloured line)
plot_tag_pressure(tag)
```

geopressure_timeseries

Retrieve an ERA5 pressure time series

Description

`geopressure_timeseries()` retrieves an hourly ERA5 surface-pressure time series at one location. Supply a tag pressure series to also normalise ERA5 pressure and estimate altitude. `geopressure_timeseries_arco()` and `geopressure_timeseries_api()` are shortcuts that force the corresponding source.

Usage

```
geopressure_timeseries(
  lat,
  lon,
  pressure = NULL,
  start_time = NULL,
  end_time = NULL,
  quiet = FALSE,
  debug = FALSE,
  source = c("auto", "arco", "api"),
  era5_dataset = c("land", "single-levels")
)

geopressure_timeseries_api(
  lat,
  lon,
  pressure = NULL,
  start_time = NULL,
  end_time = NULL,
  quiet = FALSE,
  debug = FALSE
)
```

```

geopressure_timeseries_arco(
  lat,
  lon,
  pressure = NULL,
  start_time = NULL,
  end_time = NULL,
  quiet = FALSE,
  debug = FALSE,
  era5_dataset = c("land", "single-levels")
)

```

Arguments

lat	Numeric scalar latitude, between -90 and 90 degrees.
lon	Numeric scalar longitude, between -180 and 180 degrees.
pressure	Optional data.frame with a date-time date column and numeric pressure value column in hPa. Additional columns are retained.
start_time, end_time	Start and end of the requested interval when pressure is NULL.
quiet	Logical to suppress progress messages.
debug	Logical to display request details.
source	Data source: "auto", "arco", or "api".
era5_dataset	ERA5 product used by ARCO: "land" (default) at 0.1 degree resolution or "single-levels" at 0.25 degree resolution. Prefer "single-levels" when pressure is supplied, because altitude is then computed; see the <i>Choosing era5_dataset</i> section. GeoPressureAPI uses its own configuration.

Value

A data.frame containing **date**, **surface_pressure**, **lat**, and **lon**. With **pressure**, it also retains the input columns, renames **value** to **pressure_tag**, and adds **surface_pressure_norm** and **altitude**.

Data sources

source = "arco" reads ECMWF's Analysis-Ready Cloud-Optimised (ARCO) archive directly. It is generally faster for a long time series at one location, supports ERA5-Land and global ERA5, and requires an ECMWF API key plus the optional **Rarr** and **ecmwf** packages.

source = "api" asks the hosted GeoPressureAPI to prepare the data. It needs no ECMWF key or **Rarr** installation, but depends on that service and supports its fixed ERA5 configuration.

The default, **source = "auto"**, uses ARCO when a key stored by `ecmwf::wf_set_key()` is available and GeoPressureAPI otherwise. Use `geopressure_timeseries_arco()` or `geopressure_timeseries_api()` to select a backend explicitly.

ERA5 datasets and matching

With ARCO, `era5_dataset = "single-levels"` uses global ERA5 on a 0.25 degree grid and retains locations over water. `era5_dataset = "land"` (the default) uses ERA5-Land on a 0.1 degree grid; it has finer spatial resolution but is masked over oceans, so ocean locations are moved to the closest land cell, and it must not be used when `pressure` is supplied because `altitude` is then computed from it (see the *Choosing era5_dataset* section). GeoPressureAPI chooses its ERA5 data internally and moves ocean locations onshore.

Without `pressure`, the requested interval is returned hourly. With ARCO and `pressure`, each tag time is matched to its closest ERA5 hour and restored after matching; no temporal interpolation is used. GeoPressureAPI receives the original tag timestamps and performs the matching remotely.

Pressure-derived altitude

When altitude is produced, GeoPressureR uses the same barometric relation as **GeoPressureAPI**:

$$z_{\text{tag}} = z_{\text{ERA5}} + \frac{T_{\text{ERA5}}}{L_b} \left[\left(\frac{P_{\text{tag}}}{P_{\text{ERA5}}} \right)^{-RL_b/(gM)} - 1 \right].$$

Here, P_{tag} is tag pressure, while P_{ERA5} , T_{ERA5} , and z_{ERA5} are ERA5 surface pressure, 2 m temperature, and model-surface elevation. Model-surface elevation is obtained from ERA5 surface geopotential divided by standard gravity. The constants are the standard temperature lapse rate $L_b = -0.0065$ K/m, universal gas constant $R = 8.31432$ J/(mol K), standard gravity $g = 9.80665$ m/s², and molar mass of dry air $M = 0.0289644$ kg/mol. Pressure is converted to Pa internally and altitude is returned in metres above mean sea level.

Choosing era5_dataset

Use `era5_dataset = "single-levels"` whenever `altitude` is computed.

ERA5-Land's `surface_pressure` is not the exact hydrostatic image of the orography ERA5-Land publishes as `geopotential` — the two disagree by up to ~10 hPa in steep terrain. The z_{ERA5} term therefore fails to cancel from the relation above and the whole discrepancy lands in the retrieved altitude. Measured against 41,653 hourly station-pressure observations from 271 NOAA ISD stations (2–3576 m, Alps, July 2020), compared with surveyed station elevation:

<code>era5_dataset</code>	bias	MAE	RMSE
"single-levels"	−0.6 m	9.0 m	27.4 m
"land" / "both"	+2.5 m	55.3 m	76.9 m

ERA5-Land replays the land component of ERA5 and assimilates no atmospheric observations of its own, so it carries no independent information about absolute altitude. Substituting the orography implied by its own surface pressure removes the error exactly and reproduces "single-levels" to 0.1 m.

"land" and "both" remain the defaults for backward compatibility. They are appropriate for retrieving other variables at 0.1 degree resolution, and for `geopressure_map()`, whose pressure mismatch is differential and unaffected.

Accuracy

With `era5_dataset = "single-levels"` the altitude error separates into two parts that behave very differently. A static per-site offset (median 3.7 m, p90 15 m) reflects station metadata and sub-grid terrain and does not vary in time, while the temporal scatter (median SD 3.1 m, p90 7.7 m) is the actual reanalysis error. Expect roughly **3 m for relative altitude changes at a fixed location** and **10 m mean absolute error for absolute altitude**, degrading to tens of metres in steep terrain — where it is a fixed offset rather than noise. Precision is nearly independent of flight altitude: de-biased RMSE stays 2–7 m up to 1000 m above the model surface and about 12 m at 1000–3000 m above it.

A note on accumulated variables

`total_precipitation`, `surface_solar_radiation_downwards` and `surface_thermal_radiation_downwards` do **not** mean the same thing across backends when `era5_dataset = "land"`. GeoPressureAPI serves Earth Engine's ERA5-Land collection, where these bands are overwritten by their `*_hourly` counterparts and are therefore hourly increments. ARCO serves ECMWF's own ERA5-Land fields, which accumulate from 00 UTC. For `era5_dataset = "single-levels"` both are hourly and agree. Take the difference between consecutive hours if you need increments from ARCO over land.

Pressure normalisation

ERA5 surface pressure is also shifted to the mean tag-pressure level within each elevation-label group. Flight observations (`stap_id == 0`) and observations labelled "discard" are excluded from the group means. The adjusted series is returned as `surface_pressure_norm`.

ECMWF API key

ARCO access and CDS downloads use the same ECMWF Personal Access Token. Create an ECMWF account, copy the token from your [Climate Data Store profile](#), accept the dataset licence terms, and store the token once in your system keyring:

```
ecmwfr::wf_set_key("your-personal-access-token")
```

Do not put the token in a script or commit it to version control. GeoPressureR retrieves the stored token with `ecmwfr::wf_get_key()` for all ECMWF-backed functions, including `geopressure_timeseries()`, `pressurepath_create()`, and `tag_download_wind()`.

See Also

Other pressurepath: `plot_pressurepath()`, `pressurepath_create()`, `pressurepath_variable_available()`

Examples

```
geopressure_timeseries(  
  lat = 46,  
  lon = 6,  
  start_time = "2020-01-01",  
  end_time = "2020-01-02"  
)
```

GeoPressureR

GeoPressureR: Global positioning by atmospheric pressure.

Description

GeoPressureR is a R package which help researchers determine the position of a bird based on the data retrieved from multi-sensor geolocators.

Details

The main novelty of this package is to match atmospheric pressure measurements to weather reanalysis database (ERA-5) and produce a probability maps of position.

In addition, we use a graphical model to combine pressure, light, acceleration and windspeed data into a trajectory model of the bird.

Author(s)

Maintainer: Raphaël Nussbaumer <rafnuss@gmail.com> ([ORCID](#)) [copyright holder]

Authors:

- Raphaël Nussbaumer <rafnuss@gmail.com> ([ORCID](#)) [copyright holder]
- Mathieu Gravey <mathieu@mgravey.com> ([ORCID](#))

See Also

Useful links:

- <https://github.com/GeoPressure/GeoPressureR>
- <https://geopressure.org/GeoPressureR/>
- Report bugs at <https://github.com/GeoPressure/GeoPressureR/issues>

geopressuretemplate *Workflow for GeoPressureR*

Description

The `geopressuretemplate()` function manages the complete workflow for modelling bird trajectories using geolocator data. It includes the creation `tag` object, the construction of the likelihood maps (pressure and/or light), the creation of the graph model construction, and, finally, the estimation of the trajectory products and the pressure path computation. Read the [Workflow chapter in the GeoPressureManual](#) for more information.

Usage

```
geopressuretemplate(  
  id,  
  config = config::get(config = id),  
  quiet = FALSE,  
  file = glue::glue("./data/interim/{id}.RData"),  
  ...  
)  
  
geopressuretemplate_config(  
  id,  
  config = config::get(config = id),  
  assert_tag = TRUE,  
  assert_graph = FALSE,  
  ...  
)  
  
geopressuretemplate_graph(  
  id,  
  config = config::get(config = id),  
  quiet = FALSE,  
  file = glue::glue("./data/interim/{id}.RData"),  
  ...  
)  
  
geopressuretemplate_pressurepath(  
  id,  
  config = config::get(config = id),  
  quiet = FALSE,  
  file = glue::glue("./data/interim/{id}.RData"),  
  ...  
)  
  
geopressuretemplate_tag(  
  id,
```

```

    config = config::get(config = id),
    quiet = FALSE,
    file = glue::glue("./data/interim/{id}.RData"),
    ...
)

```

Arguments

<code>id</code>	unique identifier of a tag.
<code>config</code>	configuration object specifying workflow parameters, which is loaded by default using <code>config::get(config = id)</code> .
<code>quiet</code>	Logical. If <code>TRUE</code> , suppresses informational messages during execution. The default value is <code>FALSE</code> .
<code>file</code>	A file path to save the intermediate results (e.g., tag, graph, and pressure paths). Default is <code>./data/interim/{id}.RData</code> .
<code>...</code>	Additional parameters to overwrite default or config values. Always prefer to modify <code>config.yml</code> if possible.
<code>assert_tag</code>	Logical. If <code>TRUE</code> , check that the config is compatible for the creation of a tag. The default value is <code>TRUE</code> .
<code>assert_graph</code>	Logical. If <code>TRUE</code> , check that the config is compatible for the creation of a graph. The default value is <code>TRUE</code> . Set to <code>FALSE</code> only if you don't want to create a graph model

Details

The `geopressuretemplate` function is a high-level entry point that coordinates multiple steps for processing the geolocator data and produce trajectories. It relies on underlying child functions for each step:

- **Tag Creation** `geopressuretemplate_tag()`: Initializes and labels the `tag` object. It also generates light and pressure likelihood maps:
 1. `tag_create()`: Initializes the tag object.
 2. `tag_label()`: Adds labels.
 3. `tag_set_map()`: Sets the spatial and temporal parameters.
 4. If "map_pressure" is in the `config$geopressuretemplate$likelihood`:
 - `geopressure_map()`: Computes the pressure likelihood.
 5. If "map_light" is in the `config$geopressuretemplate$likelihood`:
 - `twilight_create()` |> `twilight_read()` |> `geolight_map()`: Computes the light likelihood.
- **Graph Creation** `geopressuretemplate_graph()`: Builds a movement model graph based tag data, and can include wind effects if specified. Outputs such as marginal distributions, most likely paths, and simulation paths can be computed.
 1. `graph_create()`: Creates the graph based on tag.

2. If `config$graph_set_movement$type == "gs"` (i.e., no wind):
 - `graph_set_movement()`: Sets the movement model without wind.
 3. If `config$graph_set_movement$type == "as"` (i.e., with wind):
 - `graph_add_wind()`: Adds wind data to the graph.
 - `graph_set_movement()`: Sets the movement model with wind.
 4. If "marginal" is in `config$geopressuretemplate$outputs`:
 - `graph_marginal()`: Computes the marginal distribution map.
 5. If "most_likely" is in `config$geopressuretemplate$outputs`:
 - `graph_most_likely()`: Computes the most likely path based on the movement model.
 6. If "simulation" is in `config$geopressuretemplate$outputs`:
 - `graph_simulation()`: Runs simulations to model multiple possible paths.
 7. `save()`: Saves the computed graph and associated objects in `data/interim/{id}.Rdata`
- **Pressure Path Processing** `geopressuretemplate_pressurepath()`: Computes pressurepaths (`pressurepath_create`) using the content of the `Rdata` file and appending the `pressurepath` `data.frame` to the same file.
1. If "most_likely" is in `config$geopressuretemplate$pressurepath`, computes the pressure path for `path_most_likely`.
 2. If "geopressureviz" is in `config$geopressuretemplate$pressurepath` computes the pressure path for `path_geopressureviz`

Each of these child functions can be called individually or automatically as part of the `geopressuretemplate` workflow.

Value

The function returns nothing. Instead, it saves the processed outputs (tag, graph, pressure paths, etc.) to the specified `file`.

See Also

[GeoPressureManual](#)

Examples

```
# Run the complete geopressuretemplate workflow
geopressuretemplate("18LX")

# Overwrite parameters passed through ...
geopressuretemplate("18LX",
  tag_set_map = list(scale = 2),
  geopressure_map = list(max_sample = 100, margin = 20)
)
```

```
# Or run step-by-step
# you can check that all the parameters are correctly set in the config file
geopressuretemplate_config(id)
# 1. creation of the tag
tag <- geopressuretemplate_tag("18LX")
# 2. creation of the graph
geopressuretemplate_graph("18LX")
# 3. Computation of the pressurepath
geopressuretemplate_pressurepath("18LX")
```

geopressureviz

Start the GeoPressureViz shiny app

Description

GeoPressureViz is an interactive app to inspect and manually edit migration paths together with the underlying map likelihoods and pressure time series by stationary period. It is useful to compare pressure-, light- and distance-based information before finalizing a path.

The app can start from a `tag` object already in memory, or from an interim `.RData/.rda` file that contains at least `tag` (and optionally `marginal`, `path_most_likely`, `pressurepath`, `pressurepath_most_likely`).

When `run_bg = FALSE`, the app stores the current edited path in the Shiny option `path_geopressureviz` and returns it invisibly when the app closes. You can capture it from the return value of this function or retrieve it afterwards in the same R session with `shiny::getShinyOption("path_geopressureviz")`.

When `run_bg = TRUE`, the function returns the background process object instead of the edited path.

The app Save button writes the current path to `./data/interim/{id}-path-geopressureviz.csv`.

Learn more about GeoPressureViz in the [GeoPressureManual](#).

Usage

```
geopressureviz(
  x,
  path = NULL,
  marginal = NULL,
  launch_browser = TRUE,
  run_bg = TRUE
)
```

Arguments

- `x` One of:
- a GeoPressureR `tag` object;
 - a path to an existing `.RData/.rda` file;
 - a tag id (character scalar), interpreted as `"./data/interim/{id}.RData"`.

<code>path</code>	Optional GeoPressureR <code>path</code> or <code>pressurepath</code> data.frame. If NULL, a path is resolved from available inputs in this order: • for file/id input (x character): <code>path_most_likely</code> (if present), then <code>pressurepath</code> (if present); • otherwise fallback to <code>tag2path(tag, interp = 1)</code>.
<code>marginal</code>	map of the marginal probability computed with <code>graph_marginal()</code> . Overwrite the <code>path</code> or <code>pressurepath</code> contained in the <code>.Rdata</code> file.
<code>launch_browser</code>	If true (by default), the app runs in your browser, otherwise it runs on Rstudio.
<code>run_bg</code>	If true, the app runs in a background R session using the <code>callr</code> package. This allows you to continue using your R session while the app is running.

Value

When `run_bg = FALSE`: The edited path, returned invisibly when the app closes. During the app session, the current path is also stored in the Shiny option `shiny::getShinyOption("path_geopressureviz")` which can be read afterwards in the same R session. When `run_bg = TRUE`: The background process object. In this case the edited path is not returned to the calling console session.

Basemap API keys

Interactive maps always offer Esri basemaps (gray canvas, imagery and topographic). CARTO, Mapbox and MapTiler basemaps are added to the layer control only when their API key is stored in your system keyring. Get a key from each provider:

- CARTO: "Get your API key" on [CARTO Basemaps](#).
- Mapbox: [Access tokens](#) page of your account (see [Mapbox documentation](#)).
- MapTiler: [API keys](#) page of your MapTiler Cloud account (see [MapTiler documentation](#)).

Then store each key once:

```
keyring::key_set_with_value("CARTO_API_KEY", password = "<your CARTO API key>")
keyring::key_set_with_value("MAPBOX_ACCESS_TOKEN", password = "<your Mapbox token>")
keyring::key_set_with_value("MAPTILER_API_KEY", password = "<your MapTiler API key>")
```

Mapbox access tokens are visible in the browser when a map is rendered. Use a public, read-only token rather than a secret token.

See Also

[GeoPressureManual](#)

Examples

```
geopressureviz("18LX")
```

graph_add_wind	Compute windspeed and airspeed on a graph
----------------	---

Description

Reads the NetCDF files downloaded and interpolate the average windspeed experienced by the bird on each possible edge, as well as the corresponding airspeed.

In addition, the graph can be further pruned based on a threshold of airspeed `thr_as`.

See section 2.2.4 in Nussbaumer (2023b) for more technical details and the [GeoPressure-Manual](#) for an illustration on how to use it.

Usage

```
graph_add_wind(graph, thr_as = Inf, variable = lifecycle::deprecated(), ...)
```

Arguments

<code>graph</code>	a GeoPressureR graph object.
<code>thr_as</code>	threshold of airspeed (km/h).
<code>variable</code>	[Deprecated] No longer used. <code>graph_add_wind()</code> always uses <code>c("u", "v")</code> .
<code>...</code>	Arguments passed on to edge_add_wind
<code>pressure</code>	pressure measurement of the associated <code>tag</code> data used to estimate the pressure level (i.e., altitude) of the bird during the flights. This data.frame needs to contain <code>date</code> as POSIXt and <code>value</code> in hPa. If not provided, uses <code>graph\$pressure</code> , assuming that argument <code>graph</code> is a GeoPressureR <code>tag</code> object.
<code>rounding_interval</code>	temporal resolution on which to query the variable (min). Default is to match ERA5 native resolution (1hr).
<code>interp_spatial_linear</code>	logical to interpolate the variable linearly over space, if <code>FALSE</code> takes the nearest neighbour. ERA5 native resolution is 0.25°
<code>quiet</code>	logical to hide messages about the progress
<code>file</code>	absolute or relative path of the ERA5 wind data file to be downloaded. Function taking as arguments (1) the stationary period identifier and (2) the <code>tag_id</code> .

Value

A `graph` object with windspeed and airspeed as `ws` and `as` respectively.

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 doi:[10.1111/2041-210X.14082](#).

See Also[GeoPressureManual](#)

Other graph: [graph_create\(\)](#), [graph_marginal\(\)](#), [graph_most_likely\(\)](#), [graph_set_movement\(\)](#), [graph_simulation\(\)](#), [print.graph\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(extent = c(-16, 23, 0, 50), scale = 1) |>
  geopressure_map(quiet = TRUE)
})
graph <- graph_create(tag, quiet = TRUE)
graph <- graph_add_wind(graph, pressure = tag$pressure, quiet = TRUE)
```

graph_assert	<i>Assert the status of a graph</i>
--------------	-------------------------------------

Description

This function check the condition of a **graph** object.

Usage

```
graph_assert(graph, condition = "graph")
```

Arguments

graph	a GeoPressureR graph object
condition	condition to assert graph for. One of "graph" (default), "movement", or "full".

Value

logical indicating whether the **graph** object fulfil the condition.

Examples

```
graph_assert(graph, "movement")
```

graph_create	Create a graph object
--------------	------------------------------

Description

This function returns a trellis graph representing the trajectory of a bird based on filtering and pruning the likelihood maps provided.

In the final graph, we only keep the likely nodes (i.e., position of the bird at each stationary periods) defined as (1) those whose likelihood value are within the threshold of percentile `thr_likelihood` of the total likelihood map and (2) those which are connected to at least one edge of the previous and next stationary periods requiring an average ground speed lower than `thr_gs` (in km/h).

For more details and illustration, see [section 2.2 of Nussbaumer et al. \(2023b\)](#) and the [GeoPressureManual](#)

Usage

```
graph_create(
  tag,
  thr_likelihood = 0.99,
  thr_gs = 150,
  likelihood = NULL,
  quiet = FALSE,
  geosphere_dist = lifecycle::deprecated(),
  geosphere_bearing = lifecycle::deprecated(),
  workers = lifecycle::deprecated()
)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object.
<code>thr_likelihood</code>	threshold of percentile (see details).
<code>thr_gs</code>	threshold of groundspeed (km/h) (see details).
<code>likelihood</code>	Field of the <code>tag</code> list containing the likelihood map (character). By default, uses the product of <code>map_pressure</code> and <code>map_light</code> if available, otherwise the first available likelihood map in the tag.
<code>quiet</code>	logical to hide messages about the progress.
<code>geosphere_dist</code>	[Deprecated] This argument is no longer used. Distance calculations now use a custom memory-efficient Haversine implementation.
<code>geosphere_bearing</code>	[Deprecated] This argument is no longer used. Bearing calculations now use a custom memory-efficient implementation.
<code>workers</code>	[Deprecated] This argument is no longer used. Parallel processing has been removed to avoid memory issues.

Value

Graph as a list

- **s**: source node (index in the 3d grid lat-lon-stap)
- **t**: target node (index in the 3d grid lat-lon-stap)
- **gs**: average ground speed required to make that transition (km/h) as complex number representing the E-W as real and S-N as imaginary
- **obs**: observation model, corresponding to the normalized likelihood in a 3D matrix of size **sz**
- **sz**: size of the 3d grid lat-lon-stap
- **stap**: data.frame of all stationary periods (same as **tag\$stap**)
- **equipment**: node(s) of the first stap (index in the 3d grid lat-lon-stap)
- **retrieval**: node(s) of the last stap (index in the 3d grid lat-lon-stap)
- **mask_water**: logical matrix of water-land
- **param**: list of parameters including **thr_likelihoood** and **thr_gs** (same as **tag\$param**)

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 doi:10.1111/2041-210X.14082.

See Also

[GeoPressureManual](#)

Other graph: [graph_add_wind\(\)](#), [graph_marginal\(\)](#), [graph_most_likely\(\)](#), [graph_set_movement\(\)](#), [graph_simulation\(\)](#), [print.graph\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read() |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
  ) |>
  geopressure_map(quiet = TRUE) |>
  geolight_map(quiet = TRUE)
})

# Create graph
graph <- graph_create(tag, thr_likelihoood = 0.95, thr_gs = 100, quiet = TRUE)

print(graph)
```

graph_marginal	<i>Compute marginal probability map</i>
----------------	---

Description

Compute the marginal probability map from a graph. The computation uses the [forward backward algorithm](#). For more details, see [section 2.3.2 of Nussbaumer et al. \(2023b\)](#) and the [GeoPressureManual](#).

Usage

```
graph_marginal(graph, quiet = FALSE)
```

Arguments

graph	a GeoPressureR graph with defined movement model graph_set_movement() .
quiet	logical to hide messages about the progress.

Value

A list of the marginal maps for each stationary period (even those not modelled). Best to include within [tag](#).

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 [doi:10.1111/2041-210X.14082](#).

See Also

[GeoPressureManual](#)

Other graph: [graph_add_wind\(\)](#), [graph_create\(\)](#), [graph_most_likely\(\)](#), [graph_set_movement\(\)](#), [graph_simulation\(\)](#), [print.graph\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read() |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
  ) |>
  geopressure_map(quiet = TRUE) |>
```

```

    geolight_map(quiet = TRUE)
  })

  # Create graph
  graph <- graph_create(tag, quiet = TRUE)

  # Define movement model
  graph <- graph_set_movement(graph)

  # Compute marginal
  marginal <- graph_marginal(graph)

  plot(marginal)

```

graph_most_likely	<i>Compute the most likely trajectory</i>
-------------------	---

Description

Compute the trajectory which maximizes the joint probability using the [Viterbi algorithm](#) on the graph structure. For more details, see [section 2.3.1 of Nussbaumer et al. \(2023b\)](#) and the [GeoPressureManual](#).

Usage

```
graph_most_likely(graph, quiet = FALSE)
```

Arguments

graph	a graph object.
quiet	logical to hide messages about the progress.

Value

Path data.frame containing the columns -**stap_id** stationary period

- **j** unique ID for each path, here always 1 as there is a single path.
- **ind** indices of the coordinate in the 2D grid. Useful to retrieve map or graph information
- **lat** latitude,
- **lon** longitude
- **start** datetime of the start of the stationary period (same as in **stap**)
- **end** datetime of the end of the stationary period (same as in **stap**)
- **include** logical if stationary period was modelled (same as in **stap**)

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 doi:10.1111/2041-210X.14082.

See Also

[GeoPressureManual](#)

Other graph: [graph_add_wind\(\)](#), [graph_create\(\)](#), [graph_marginal\(\)](#), [graph_set_movement\(\)](#), [graph_simulation\(\)](#), [print.graph\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read() |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
  ) |>
  geopressure_map(quiet = TRUE) |>
  geolight_map(quiet = TRUE)
})

# Create graph
graph <- graph_create(tag, quiet = TRUE)

# Define movement model
graph <- graph_set_movement(graph)

# Compute most likely path
path_most_likely <- graph_most_likely(graph, quiet = TRUE)

plot_path(path_most_likely)
```

graph_set_movement	Define the movement model of a graph
--------------------	--------------------------------------

Description

Configure the movement model of a **graph** by defining the value of the parameters needed to build the transition [graph_transition\(\)](#) through [speed2prob\(\)](#).

Three methods are currently implemented with two parametric function "gamma" and "logis" suitable when wind data is not available and are thus defining the probability of a groundspeed.

If wind data is available, it is recommended to use the "power" method which rely on the power curve equation (energy vs airspeed) to estimate the probability of an airspeed. Read more about this approach in [section 2.2.5. of Nussbaumer et al. \(2023b\)](#)

Usage

```
graph_set_movement(
  graph,
  type = ifelse("ws" %in% names(graph), "as", "gs"),
  method = ifelse("ws" %in% names(graph), "power", "gamma"),
  shape = 7,
  scale = 7,
  location = 40,
  bird = NULL,
  power2prob = function(power) (1/power)^3,
  low_speed_fix = 15,
  zero_speed_ratio = 1
)
```

Arguments

graph	a GeoPressureR graph object.
type	Ground speed "gs" or airspeed "as"
method	method used to convert the speed to probability ("gamma", "logis" or "power")
shape	parameter of the gamma distribution (km/h)
scale	parameter of the gamma and logistic distribution (km/h)
location	parameter for the logistic distribution (km/h)
bird	A GeoPressureR bird object containing the basic morphological traits necessary: mass, wing span, wing aspect ratio, and body frontal area. See bird_create() .
power2prob	function taking power as a single argument and returning a probability
low_speed_fix	speed below which the probability remains the same, i.e. we assign the same probability at low_speed_fix for any lower speed. This parameter is used to allow short flights covering small distances. (unit of km/h)
zero_speed_ratio	multiplicative ratio of the probability for speed zero. This ratio apply only when the bird is staying at the same location (fly and come back or stay within pixel size). This parameter (when greater than 1) is used to favour a bird to stay at the same location rather than perform short fly.

Value

Graph list with a new list **graph\$movement** storing all the parameters needed to compute the transition probability

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 doi:10.1111/2041-210X.14082.

See Also

Other graph: [graph_add_wind\(\)](#), [graph_create\(\)](#), [graph_marginal\(\)](#), [graph_most_likely\(\)](#), [graph_simulation\(\)](#), [print.graph\(\)](#)

Other movement: [bird_create\(\)](#), [plot_graph_movement\(\)](#), [speed2prob\(\)](#), [tag_download_wind\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    twilight_create() |>
    twilight_label_read() |>
    tag_set_map(
      extent = c(-16, 23, 0, 50),
      known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
    ) |>
    geopressure_map(quiet = TRUE) |>
    geolight_map(quiet = TRUE)
})

graph <- graph_create(tag, quiet = TRUE)

graph <- graph_set_movement(graph,
  method = "gamma",
  shape = 4,
  scale = 6,
  low_speed_fix = 10
)
plot_graph_movement(graph)

graph <- graph_set_movement(graph,
  method = "logis",
  shape = 4,
  location = 60,
  low_speed_fix = 10
)
plot_graph_movement(graph)
```

graph_simulation	<i>Simulate randomly multiple trajectories</i>
------------------	--

Description

This function randomly simulates multiple trajectories from a graph using the forward filtering backward sampling algorithm. For more details, see [section 2.3.3 of Nussbaumer et al. \(2023b\)](#) and the [GeoPressureManual](#).

Usage

```
graph_simulation(graph, nj = 10, quiet = FALSE)
```

Arguments

graph	a graph object.
nj	Number of simulations.
quiet	logical to hide messages about the progress.

Value

Path data.frame containing the columns -**stap_id** stationary period

- j unique ID for each simulation.
- ind indices of the coordinate in the 2D grid. Useful to retrieve map or graph information
- lat latitude,
- lon longitude
- start datetime of the start of the stationary period (same as in **stap**)
- end datetime of the end of the stationary period (same as in **stap**)
- include logical if stationary period was modelled (same as in **stap**)

References

Nussbaumer, Raphaël, Mathieu Gravey, Martins Briedis, Felix Liechti, and Daniel Sheldon. 2023. Reconstructing bird trajectories from pressure and wind data using a highly optimized hidden Markov model. *Methods in Ecology and Evolution*, 14, 1118–1129 [doi:10.1111/2041-210X.14082](#).

See Also

[GeoPressureManual](#)

Other graph: [graph_add_wind\(\)](#), [graph_create\(\)](#), [graph_marginal\(\)](#), [graph_most_likely\(\)](#), [graph_set_movement\(\)](#), [print.graph\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read() |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
  ) |>
  geopressure_map(quiet = TRUE) |>
  geolight_map(quiet = TRUE)
})

# Create graph
graph <- graph_create(tag, quiet = TRUE)

# Define movement model
graph <- graph_set_movement(graph)

# Compute simulations
path_simulation <- graph_simulation(graph, quiet = TRUE)

plot_path(path_simulation, plot_leaflet = FALSE)
```

load_interim

Load interim objects from an RData file

Description

Loads one or more objects from an interim `.RData` file created during the GeoPressure analysis. This is a convenience function to quickly restore saved objects (e.g., `tag`, `graph`, `path_most_likely`) for a given tag id or directly from a file path.

Usage

```
load_interim(
  x,
  var = NULL,
  var_optional = NULL,
  envir = parent.frame(),
  verbose = FALSE
)
```

Arguments

x A character string of length 1. If `x` is the path to an existing `.RData/.rda` file, that file is used. Otherwise, `x` is interpreted as a tag identifier and the function looks for `./data/interim/{x}.RData`.

var	Optional name (or names) of objects that must exist in the file. If NULL and var_optional is also NULL, all objects from the file are loaded into envir (the behaviour of <code>base::load()</code>).
var_optional	Character vector of object names that may or may not exist in the file. Any of these that are present will be loaded; missing ones are silently ignored.
envir	the environment where the data should be loaded.
verbose	should item names be printed during loading?

Value

If no selection is requested (**var** and **var_optional** both NULL), invisibly returns the character vector of object names loaded (as in `base::load()`). Otherwise, if a single object name is requested, invisibly returns that object; if multiple names are requested, invisibly returns a named list of those objects.

Examples

```
load_interim("18LX", var = c("tag", "graph"))
```

map_add_mask_water	<i>Create water mask for geolight map</i>
---------------------------	---

Description

This function creates a water mask for maps with **geolight_map** already run. The mask is based on coastline data from Natural Earth and is stored as **map\$mask_water** as a boolean matrix where TRUE indicates water and FALSE indicates land.

A pixel is marked as land if any land exists within it (using the conservative **touches=TRUE** approach in rasterization), which is important for detecting small islands; the mask then flags the remaining pixels as water.

Usage

```
map_add_mask_water(map, ne_scale = "medium", land_threshold = 0)
```

Arguments

map	a GeoPressureR map object from geolight_map .
ne_scale	Natural Earth resolution passed to <code>rnaturalearth::ne_countries(scale = ...)</code> . Valid values are "small", "medium", "large" or <code>c(110, 50, 10)</code> . Default is "medium".
land_threshold	Numeric threshold used to classify pixels as water from the rasterized land fraction. Pixels with land fraction at or below land_threshold are marked as water. Use 0 for a conservative mask where any land in a pixel makes it land.

Details

The coastline data is rasterized to match the map's grid (defined by `extent` and `scale` from the map). `land_threshold` controls how much land can be present in a pixel before it is treated as land in `map$mask_water`. For "large"/10 resolution, `rnaturalearthhires` is used by `rnaturalearth` when available.

Value

A map object with `mask_water` added as a logical matrix.

See Also

Other map: `map_create()`, `print.map()`, `rast.map()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    scale = 4,
    known = data.frame(
      stap_id = 1,
      known_lon = 17.05,
      known_lat = 48.9
    )
  ) |>
  twilight_create() |>
  twilight_label_read() |>
  geolight_map(quiet = TRUE)

  # Create water mask
  tag$map_light <- map_add_mask_water(tag$map_light)

  # View the mask
  image(tag$map_light$mask_water)
})
```

map_create

Create a map object

Description

This function creates a GeoPressureR map object from a spatio-temporal dataset. The data needs to be discretized according to `scale`, `extent` (space) and `stap` (time).

This function is used by `geopressure_map` and `graph_marginal`.

Usage

```
map_create(data, extent, scale, stap, id = NA, type = "unknown")
```

Arguments

data	list of matrices of the same size, one for each stationary period.
extent	geographical extent of the map on which the likelihood and graph model will be computed. Vector of length 4 <code>c(xmin, xmax, ymin, ymax)</code> or <code>c(W, E, S, N)</code> .
scale	number of pixels per 1° latitude-longitude. For instance, <code>scale = 10</code> for a resolution of 0.1° (~10km) and <code>scale=4</code> for a resolution of 0.25° (~30km). To avoid interpolating the ERA5 data, the scale should be equal to or smaller than 10. Read more about scale on the Google earth Engine documentation .
stap	a data.frame of stationary periods.
id	tag identifier.
type	map type used for default palettes in plots. One of: "unknown", "pressure", "light", "magnetic", "pressure_mse", "pressure_mask", "mask_water", "magnetic_intensity", "magnetic_inclination", "marginal", "twilight".

Value

A GeoPressure map object

See Also

Other map: [map_add_mask_water\(\)](#), [print.map\(\)](#), [rast.map\(\)](#)

Examples

```
data <- lapply(1:10, \(x) matrix(runif(5000), nrow = 50, ncol = 100))
scale <- 10
extent <- c(0, 10, 0, 5)
seq(as.Date("2023-01-01"), as.Date("2023-01-10"), by = "day")
stap <- data.frame(
  stap_id = 1:10,
  start = seq(as.POSIXct("2023-01-01", tz = "UTC"),
    as.POSIXct("2023-01-10 UTC", tz = "UTC"),
    by = "day"
  ),
  include = TRUE
)
stap$end <- stap$start + sample(1:10) * 10000

# Create the map
map <- map_create(
  data = data,
  extent = extent,
  scale = scale,
```

```

    stap = stap,
    id = "18LX",
    type = "pressure"
)

print(map)

plot(map)

```

map_expand

Construct grid from extent and scale

Description

Take the two parameters defining the spatial grid of a map in GeoPressureR (**extent** and **scale**) and constructs various possible spatial variable of interest for the spatial grid (e.g., grid dimension, latitude and longitude...)

This function is used in multiple functions of GeoPressureR to allow to only store **extent** and **scale** in **tag** and **graph** while having access to all spatial variables anywhere anytime.

Usage

```
map_expand(extent, scale)
```

Arguments

extent	geographical extent of the map on which the likelihood and graph model will be computed. Vector of length 4 c(xmin , xmax , ymin , ymax) or c(W , E , S , N).
scale	number of pixels per 1° latitude-longitude. For instance, scale = 10 for a resolution of 0.1° (~10km) and scale =4 for a resolution of 0.25° (~30km). To avoid interpolating the ERA5 data, the scale should be equal to or smaller than 10. Read more about scale on the Google earth Engine documentation .

Value

A list containing:

- **extent** same as input
- **scale** same as input
- **lat** vector of latitude
- **lon** vector of longitude
- **dim** vector of length 2 of the dimension of the map (number of pixel in lat and lon)

Examples

```
str(map_expand(extent = c(0, 10, 0, 5), scale = 1))

str(map_expand(extent = c(-16, 23, 0, 50), scale = 10))
```

param_create	Create a param list
--------------	---------------------

Description

Create the list of parameter for GeoPressureR **tag** and **graph** objects.

param list are mostly used to archived the actual value of parameters used to create a **tag** and/or a **graph**, thus allowing for examination of parameters post-creation. This function should therefore not be used to set/define parameters ahead of computation. In reality, there are very few external case of use for this function.

Usage

```
param_create(id, default = FALSE, ...)
```

Arguments

id	Unique identifier of a tag.
default	logical to initiate param with default value of the package.
...	arguments passed to other methods.

Value

A GeoPressureR **param** list

See Also

Other param: [print.param\(\)](#)

Examples

```
param <- param_create("18LX", default = TRUE)
print(param)
```

path2edge

Extract the edges of a path from a graph

Description

Retrieve the edges in a **graph** corresponding to the flight transition defined by a **path**. These edges can be useful to extract flight information specific to a path.

Usage

```
path2edge(path, tag_graph)
```

Arguments

path a GeoPressureR **path** data.frame
tag_graph either a **tag** or a **graph** GeoPressureR object.

Value

Data.frame of the edge containing:

- **s**: index in 3D (lat-lon-stap) of the origin (source).
- **t**: index in 3D (lat-lon-stap) of the destination (target).
- **lat_s**: latitude of the origin (source).
- **lat_t**: latitude of the destination (target).
- **lon_s**: longitude of the origin (source).
- **lon_t**: longitude of the destination (target).
- **stap_s**: stationary period of the origin (source).
- **stap_t**: stationary period of the destination (target).
- **distance**: Distance (in km) of the flight.
- **start**: end of the flight.
- **end**: start of the flight.
- **duration**: duration of the flight.
- **n**: number of flight.
- **bearing**: Angle of the flight in degree (0° = North, 90° = East).
- **gs**: groundspeed vector expressed as a complex number. You can compute the ground-speed value (km/h) with **abs(gs)**, the W-E and S-N component of the flight with **Re(gs)** and **Im(gs)**, and the angle/direction with **Arg(gs)**. If graph provided.
- **ws**: if computed with **graph_add_wind()**, same value as **gs**. Airspeed is computed with **as = gs - ws** in complex number to keep the vectorial additive properties. If graph provided.

See Also[GeoPressureManual](#)Other path: [path2elevation\(\)](#), [path2twilight\(\)](#), [plot_path\(\)](#), [tag2path\(\)](#)**Examples**

```
edge <- path2edge(path, tag)
```

path2elevation	<i>Download ground elevation along a path</i>
----------------	---

Description

This function is a wrapper of the GeoPressureAPI `elevationPath` entry point. It queries the ground elevation from SRTM ([SRTM90_V4](#)) along a polyline specify by `path`.

Because the position are often defined on a relative coarse scale (e.g., $0.25^\circ \sim 29\text{km}$), you can request the elevation at a specify resolution defined by `scale` and return pre-defined `percentile` of the elevation at this resolution.

The returned data.frame provide the ground elevation along the path with a resolution defined by `sampling_scale`.

Usage

```
path2elevation(
  path,
  scale = 4,
  sampling_scale = scale,
  percentile = c(10, 50, 90),
  debug = FALSE
)
```

Arguments

<code>path</code>	a GeoPressureR <code>path</code> data.frame
<code>scale</code>	spatial resolution of the SRTM to use to query the elevation. <code>scale</code> is defined as the number of pixels per 1° latitude-longitude (see <code>tag_set_map()</code> for details). Native resolution of SRTM is 30m.
<code>sampling_scale</code>	spatial resolution of the point along the polyline on which the SRTM is estimated. Same unit as <code>scale</code> .
<code>percentile</code>	percentile of the ground elevation distribution found within each grid cell of the SRTM at the resolution defined by <code>scale</code> . 50 corresponds to the median.
<code>debug</code>	logical to display additional information to debug a request

Value

A data.frame containing

- `stap_id` numeric value corresponding to the ratio of distance between position of known stap
- `lon`
- `lat`
- `distance` distance in km along the path starting at the first `stap_id`

See Also

Other path: [path2edge\(\)](#), [path2twilight\(\)](#), [plot_path\(\)](#), [tag2path\(\)](#)

Examples

```
# Create a path
path <- data.frame(
  lon = c(8.47, 9.41, 9.01, -0.91, 14.24, 27.30, 34.39, 30.00),
  lat = c(48.89, 44.78, 40.07, 37.68, 17.33, 7.32, 8.09, -23.13),
  start = as.POSIXct(
    c(
      "2017-05-01 00:42", "2017-05-03 01:22", "2017-05-03 22:47", "2017-05-06 03:32",
      "2017-05-07 01:12", "2017-05-07 22:32", "2017-05-09 21:52", "2017-05-10 21:12"
    ),
    tz = "UTC"
  ),
  end = as.POSIXct(
    c(
      "2017-05-02 22:12", "2017-05-03 20:12", "2017-05-06 02:47", "2017-05-06 23:22",
      "2017-05-07 17:42", "2017-05-09 20:27", "2017-05-10 19:57", "2017-05-11 21:17"
    ),
    tz = "UTC"
  ),
  stap_id = seq(1, 8)
)

plot_path(path)

elevation <- path2elevation(path)

plot(elevation$distance, elevation$X50,
     type = "l",
     ylab = "Elevation (m)", xlab = "Distance from start (km)"
)
lines(elevation$distance, elevation$X10, lty = 5)
lines(elevation$distance, elevation$X90, lty = 5)
id <- elevation$stap_id %% 1 == 0
points(elevation$distance[id], elevation$X90[id], col = "red")
```

<code>path2twilight</code>	<i>Compute exact astronomical twilights from a <code>path</code> (positions and dates)</i>
----------------------------	--

Description

This function computes the theoretical twilight (i.e., datetime of sunrise and sunset) at given locations and for specific date.

We use the `suntools` package for this.

By default (`solar_dep=0`), the computation returns sunrise and sunset. But, it is also possible to compute different twilights by setting depression angle value greater than 0 (6° for civil, 12° for nautical and 18° for astronomical). The corresponding solar elevation threshold is `-solar_dep`, and the equivalent solar zenith angle is `90 + solar_dep`.

Usage

```
path2twilight(path, date = NULL, solar_dep = 0, return_long = TRUE)
```

Arguments

<code>path</code>	a GeoPressureR <code>path</code> or <code>pressurepath</code> data.frame
<code>date</code>	a vector of POSIXt datetime for which sunrise and sunset are computed. Be default, uses the range of <code>path\$date</code> provided.
<code>solar_dep</code>	a numerical value representing the solar depression angle used to compute sunrise and sunset. The corresponding solar elevation threshold is <code>-solar_dep</code> , and the equivalent solar zenith angle is <code>90 + solar_dep</code> (e.g. <code>solar_dep = 6</code> corresponds to zenith angle 96°).
<code>return_long</code>	logical defining the format of the data.frame returned. If <code>TRUE</code> , returns the long format identical to <code>twilight_create()</code> . If <code>FALSE</code> , return the sunrise and sunset as different column, making the data.frame the same size as <code>date</code> .

Value

if `return_long == TRUE`, a `twilight` data.frame (same as `twilight_create()`) with columns:

- `date` same as input `date`
- `twilight` date-time of twilight
- `rise` logical indicating sunrise (`TRUE`) or sunset (`FALSE`).
- `stap_id` same as `path$stap_id`
- `lat` same as `path$lat`
- `lon` same as `path$lon`

if `return_long == FALSE`, a data.frame with the same size of `date` with columns:

- `date` same as input `date`
- `sunrise` date-time of sunrise
- `sunset` date-time of sunset
- `stap_id` same as `path$stap_id`
- `lat` same as `path$lat`
- `lon` same as `path$lon`

See Also

[GeoPressureManual](#) , [suntools](#)

Other path: [path2edge\(\)](#), [path2elevation\(\)](#), [plot_path\(\)](#), [tag2path\(\)](#)

Examples

```
path <- data.frame(
  stap_id = c(1, 2, 3, 4, 5),
  j = c(1, 1, 1, 1, 1),
  ind = c(1652L, 1603L, 1505L, 1609L, 1463L),
  lat = c(48.9, 47.5, 45.5, 41.5, 37.5),
  lon = c(17.05, 16.5, 14.5, 16.5, 13.5),
  start = as.POSIXct(
    c(1501113450, 1501888650, 1501987950, 1502075550, 1502151150),
    tzzone = "UTC"
  ),
  end = as.POSIXct(c(1501876050, 1501961250, 1502046750, 1502133150, 1502323050), tzzone = "UTC"),
  include = c(TRUE, TRUE, TRUE, TRUE, TRUE),
  known = c(TRUE, FALSE, FALSE, FALSE, FALSE),
  interp = c(FALSE, FALSE, FALSE, FALSE, FALSE)
)

twl <- path2twilight(path)
```

`plot.map`

Plot a map object

Description

This function plots a `GeoPressureR` `map` object.

You can plot on top of the `map` a `path`, this uses the `plot_path()` function.

Our maps are defined in lat-lon (i.e., EPSG:4326), but the display of maps on web map are nearly always in **web mercator** (i.e., EPSG:3857). We therefore need to reproject our map for display. However, we don't really want to interpolate the map as each pixel might be important to visualize. We therefore re-project with a near-neighbour interpolation (`method = "near"` in [terra::project\(\)](#)). Yet to avoid having pixel misplaced, we generally need to use a projection with a finer resolution. The argument `fac_res_proj` controls the relative change of resolution between the original map to the projected map.

Usage

```
## S3 method for class 'map'
plot(
  x,
  path = NULL,
  thr_likelihood = 1,
  plot_leaflet = TRUE,
  provider = NULL,
  provider_options = leaflet::providerTileOptions(),
  palette = NULL,
  opacity = 0.8,
  legend = FALSE,
  fac_res_proj = 4,
  ...
)
```

Arguments

x	a GeoPressureR map object
path	a GeoPressureR path data.frame
thr_likelihood	Threshold to display likelihood values.
plot_leaflet	logical to use an interactive leaflet map instead of terra::plot
provider	optional tile provider name (see leaflet::providers). When NULL , uses all available GeoPressureR basemap layers; see the <i>Basemap API keys</i> section.
provider_options	tile options. See leaflet::addProviderTiles() and leaflet::providerTileOptions() .
palette	color palette name or vector of colors.
opacity	opacity of the raster layer in leaflet .
legend	logical to display the legend.
fac_res_proj	Factor of the resolution of the reprojection (see details above). A value of 1 will roughly reproject on a map with similar size resulting in relatively high inaccuracy of the pixel displayed. Increasing this factor will reduce the uncertainty but might also increase the computational cost of the reprojection.
...	additional parameters passed to leaflet::addRasterImage() or terra::plot() .

Value

a plot or leaflet object.

Basemap API keys

Interactive maps always offer Esri basemaps (gray canvas, imagery and topographic). CARTO, Mapbox and MapTiler basemaps are added to the layer control only when their API key is stored in your system keyring. Get a key from each provider:

- CARTO: "Get your API key" on [CARTO Basemaps](#).
- Mapbox: [Access tokens](#) page of your account (see [Mapbox documentation](#)).
- MapTiler: [API keys](#) page of your MapTiler Cloud account (see [MapTiler documentation](#)).

Then store each key once:

```
keyring::key_set_with_value("CARTO_API_KEY", password = "<your CARTO API key>")
keyring::key_set_with_value("MAPBOX_ACCESS_TOKEN", password = "<your Mapbox token>")
keyring::key_set_with_value("MAPTILER_API_KEY", password = "<your MapTiler API key>")
```

Mapbox access tokens are visible in the browser when a map is rendered. Use a public, read-only token rather than a secret token.

See Also

[plot_path\(\)](#) [plot.tag\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    scale = 4
  ) |>
  geopressure_map(quiet = TRUE)
})

plot(tag$map_pressure)

plot(tag$map_pressure, plot_leaflet = FALSE)

# `thr_likelihood` can be used to visualize its effect in `graph_create`
plot(tag$map_pressure,
  thr_likelihood = 0.9,
  palette = "viridis",
  opacity = 1,
  provider = "Esri.WorldGrayCanvas"
)
```

Description

This function plots a GeoPressureR `tag` object as a time series or a map.

By default, `type` is determined in the following order of preference according to availability:

`c("map_pressure", "map_light"), "map_pressure", "map_light", "pressure".`

`plot.tag()` calls different plotting functions depending on `type`.

- "pressure": `plot_tag_pressure()`
- "light": `plot_tag_light()`
- "acceleration": `plot_tag_acceleration()`
- "temperature": `plot_tag_temperature()`
- "twilight": `plot_tag_twilight()`
- "actogram": `plot_tag_actogram()`
- "map_*" : `plot.map()` with `tag$map_*` as first argument.

Refers to these functions for additional parameters and more flexibility in the plotting.

Usage

```
## S3 method for class 'tag'
plot(x, type = NULL, ...)
```

Arguments

<code>x</code>	a GeoPressureR <code>tag</code> object.
<code>type</code>	type of the plot to display. One of "pressure", "acceleration", "light", "temperature", "twilight", "actogram", "map", "map_pressure", "map_light", "map_pressure_mse", "map_pressure_mask", "mask_water", "temperature_external", "temperature_internal". Map can be combined by providing a vector of type.
<code>...</code>	additional parameters for <code>plot_tag_pressure()</code> , <code>plot_tag_acceleration()</code> , <code>plot_tag_light()</code> , <code>plot_tag_twilight()</code> or <code>plot.map()</code> . For twilight and actogram plots, use <code>flip_axes = TRUE</code> to place time horizontally and <code>double_plot = TRUE</code> for a 48-hour double plot.

Value

a plot, ggplotly or leaflet object.

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read()
})
```

```

# By default, plot will display the time series of pressure
plot(tag)
# Change the `type` to display other sensor
plot(tag, type = "acceleration")
plot(tag, type = "light")
plot(tag, type = "temperature")
plot(tag, type = "twilight")
plot(tag, type = "actogram")

# After you compute any likelihood map, the default will
# become this map (i.e., `type = "map"`)
tag <- tag_set_map(tag,
  extent = c(-16, 23, 0, 50),
  scale = 4,
  known = data.frame(
    stap_id = 1,
    known_lon = 17.05,
    known_lat = 48.9
  )
) |>
  geopressure_map(quiet = TRUE)
plot(tag)
# The likelihood map of light can be display with
tag <- geolight_map(tag, quiet = TRUE)
plot(tag, type = "map_light")
# When both pressure and light likelihood are present,
# the default is to display their products, equivalent
# to choose `type = c("map_pressure", "map_light")`
plot(tag)

```

plot_graph_movement *Plot movement model of a graph*

Description

This function display a plot of pressure time series recorded by a tag

Usage

```
plot_graph_movement(graph, speed = seq(0, 120), plot_plotly = FALSE)
```

Arguments

<code>graph</code>	a GeoPressureR <code>graph</code> object or a movement list (from <code>graph_set_movement</code>).
<code>speed</code>	Vector of speed value (km/h) used on the x-axis.
<code>plot_plotly</code>	logical to use <code>plotly</code>

Value

a plot or `ggplotly` object.

See Also

Other movement: [bird_create\(\)](#), [graph_set_movement\(\)](#), [speed2prob\(\)](#), [tag_download_wind\(\)](#)

Examples

```

movement_gamma <- list(
  type = "gs",
  method = "gamma",
  shape = 7,
  scale = 7,
  low_speed_fix = 15,
  zero_speed_ratio = 1
)
plot_graph_movement(movement_gamma)

movement_logis <- list(
  type = "gs",
  method = "logis",
  scale = 7,
  location = 40,
  low_speed_fix = 15,
  zero_speed_ratio = 1
)
plot_graph_movement(movement_logis)

bird <- bird_create("Example bird", mass = 0.1, wing_span = 0.4, wing_aspect = 7)
movement_power <- list(
  type = "as",
  method = "power",
  bird = bird,
  power2prob = \ (power) (1 / power)^3,
  low_speed_fix = 15,
  zero_speed_ratio = 1
)
plot_graph_movement(movement_power)

```

plot_path

Plot a path

Description

This function plots a `path` data.frame. This function is used in [plot.map\(\)](#).

Usage

```

plot_path(
  path,
  plot_leaflet = TRUE,
  map = NULL,

```

```

    provider = NULL,
    provider_options = leaflet::providerTileOptions(),
    pad = 3,
    polyline = NULL,
    circle = NULL,
    group = NULL
  )

```

Arguments

<code>path</code>	a GeoPressureR <code>path</code> data.frame.
<code>plot_leaflet</code>	logical defining if the plot is an interactive <code>leaflet</code> map or a static basic plot.
<code>map</code>	optional <code>map</code> object to plot the path on top of.
<code>provider</code>	optional tile provider name (see <code>leaflet::providers</code>). When <code>NULL</code> , uses all available GeoPressureR basemap layers; see the <i>Basemap API keys</i> section.
<code>provider_options</code>	tile options. See <code>leaflet::addProviderTiles()</code> and <code>leaflet::providerTileOptions()</code> .
<code>pad</code>	padding of the map in degree lat-lon (only for <code>plot_leaflet = FALSE</code>).
<code>polyline</code>	list of parameters passed to <code>leaflet::addPolylines()</code>
<code>circle</code>	list of parameters passed to <code>leaflet::addCircleMarkers()</code>
<code>group</code>	optional Leaflet overlay group name. When supplied, all trajectories in <code>path</code> can be shown or hidden together while retaining separate line geometries for each <code>j</code> .

Value

A `leaflet` map when `plot_leaflet = TRUE`, otherwise a `ggplot2` object.

Basemap API keys

Interactive maps always offer Esri basemaps (gray canvas, imagery and topographic). CARTO, Mapbox and MapTiler basemaps are added to the layer control only when their API key is stored in your system keyring. Get a key from each provider:

- CARTO: "Get your API key" on [CARTO Basemaps](#).
- Mapbox: [Access tokens](#) page of your account (see [Mapbox documentation](#)).
- MapTiler: [API keys](#) page of your MapTiler Cloud account (see [MapTiler documentation](#)).

Then store each key once:

```

keyring::key_set_with_value("CARTO_API_KEY", password = "<your CARTO API key>")
keyring::key_set_with_value("MAPBOX_ACCESS_TOKEN", password = "<your Mapbox token>")
keyring::key_set_with_value("MAPTILER_API_KEY", password = "<your MapTiler API key>")

```

Mapbox access tokens are visible in the browser when a map is rendered. Use a public, read-only token rather than a secret token.

See Also[plot.map\(\)](#)Other path: [path2edge\(\)](#), [path2elevation\(\)](#), [path2twilight\(\)](#), [tag2path\(\)](#)**Examples**

```

withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE)
})

path <- data.frame(
  stap_id = 1:4,
  lon = c(17.05, 16.2, NA, 15.4),
  lat = c(48.9, 47.8, NA, 46.5),
  known_lon = c(17.05, NA, NA, NA),
  known_lat = c(48.9, NA, NA, NA),
  start = as.POSIXct(
    c("2018-01-01", "2018-01-03", "2018-01-05", "2018-01-07"),
    tz = "UTC"
  ),
  end = as.POSIXct(
    c("2018-01-02", "2018-01-04", "2018-01-06", "2018-01-08"),
    tz = "UTC"
  )
)

plot_path(path)

plot_path(path, plot_leaflet = FALSE)

```

plot_pressurepath	<i>Plot a pressurepath</i>
-------------------	----------------------------

Description

Display a `pressurepath` `data.frame` as a time series or histogram

Usage

```

plot_pressurepath(
  pressurepath,
  type = "timeseries",
  sd = attr(pressurepath, "sd"),
  warning_std_thr = 3,
  plot_plotly = TRUE
)

```

Arguments

pressurepath a GeoPressureR `pressurepath` data.frame.
type "timeseries" (default), "histogram", or "altitude"
sd standard deviation of the pressure error. Numeric of length 1 or number of stationary periods.
warning_std_thr Threshold of outlier, coefficient of the *z-score*
plot_plotly logical to use `plotly`

Value

a plot or `ggplotly` object.

See Also

Other `pressurepath`: [geopressure_timeseries\(\)](#), [pressurepath_create\(\)](#), [pressurepath_variable_available\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

path <- data.frame(
  stap_id = tag$stap$stap_id,
  lat = c(48.5, 32.5, 30.5, 49.5, 41.6),
  lon = c(17.5, 13.5, 16.5, 21.5, 12.7)
)

pressurepath <- pressurepath_create(tag, path = path, quiet = TRUE)

plot_pressurepath(pressurepath)

plot_pressurepath(pressurepath, type = "histogram")

plot_pressurepath(pressurepath, type = "altitude")
```

`plot_tag_acceleration`

Plot acceleration data of a tag

Description

This function display a plot of acceleration time series recorded by a tag

Usage

```
plot_tag_acceleration(
  tag,
  variable = "activity",
  plot_plotly = TRUE,
  label_auto = TRUE,
  min_duration = 30
)
```

Arguments

tag	a GeoPressureR tag object
variable	type of acceleration variable to plot "activity" (or "value") or "mean_acceleration_z"
plot_plotly	logical to use plotly
label_auto	logical to compute and plot the flight label using <code>tag_label_auto()</code> . Only if labels are not already present on <code>tag\$acceleration\$label</code>
min_duration	Minimal duration (in minutes) to consider a high activity as migratory flight.

Value

a plot or ggplotly object.

See Also

Other `plot_tag`: [plot_tag_actogram\(\)](#), [plot_tag_light\(\)](#), [plot_tag_pressure\(\)](#), [plot_tag_temperature\(\)](#), [plot_tag_twilight\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)
})

plot_tag_acceleration(tag)
```

<code>plot_tag_actogram</code>	<i>Plot Actogram data of a tag</i>
--------------------------------	------------------------------------

Description

This function display a plot of the acceleration time series recorded by a tag

Usage

```
plot_tag_actogram(  
  tag,  
  twl_offset = NULL,  
  plot_plotly = TRUE,  
  flip_axes = FALSE,  
  double_plot = FALSE  
)
```

Arguments

- tag** a GeoPressureR **tag** object
- twl_offset** Shift of the middle of the night compared to 00:00 UTC (in hours). If not provided, it uses the middle of all nights.
- plot_plotly** logical to use **plotly**
- flip_axes** logical to place time on the x-axis and dates on the y-axis. Dates increase downward in the flipped display.
- double_plot** logical to display a 48-hour window. The second 24-hour half contains the following day's observations.

Value

a ggplot or ggplotly object.

See Also

Other `plot_tag`: [plot_tag_acceleration\(\)](#), [plot_tag_light\(\)](#), [plot_tag_pressure\(\)](#), [plot_tag_temperature\(\)](#), [plot_tag_twilight\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {  
  tag <- tag_create("18LX", quiet = TRUE)  
  
  plot_tag_actogram(tag, plot_plotly = TRUE)  
})
```

plot_tag_light	<i>Plot light data of a tag</i>
----------------	---------------------------------

Description

This function display a plot of light time series recorded by a tag

Usage

```
plot_tag_light(tag, transform_light = TRUE, plot_plotly = TRUE)
```

Arguments

`tag` a GeoPressureR `tag` object

`transform_light` logical to display a log transformation of light

`plot_plotly` logical to use `plotly`

Value

a plot or ggplotly object.

See Also

Other `plot_tag`: [plot_tag_acceleration\(\)](#), [plot_tag_actogram\(\)](#), [plot_tag_pressure\(\)](#), [plot_tag_temperature\(\)](#), [plot_tag_twilight\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)
})

plot_tag_light(tag)
```

<code>plot_tag_pressure</code>	<i>Plot pressure data of a tag</i>
--------------------------------	------------------------------------

Description

This function display a plot of pressure time series recorded by a tag

Usage

```
plot_tag_pressure(tag, plot_plotly = TRUE, warning_pressure_diff = 3)
```

Arguments

`tag` a GeoPressureR `tag` object.

`plot_plotly` logical to use `plotly`.

`warning_pressure_diff` Threshold of pressure hourly difference marking as warning (hPa).

Value

a plot or ggplotly object.

See Also

Other plot_tag: `plot_tag_acceleration()`, `plot_tag_actogram()`, `plot_tag_light()`, `plot_tag_temperature()`, `plot_tag_twilight()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)
})

plot_tag_pressure(tag, plot_plotly = FALSE)

withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_label(tag, quiet = TRUE)
})

plot_tag_pressure(tag)
```

`plot_tag_temperature` *Plot temperature data of a tag*

Description

This function display a plot of temperature time series recorded by a tag

Usage

```
plot_tag_temperature(tag, variable = "external", plot_plotly = TRUE)
```

Arguments

<code>tag</code>	a GeoPressureR tag object
<code>variable</code>	temperature variable to plot "external" or "internal"
<code>plot_plotly</code>	logical to use plotly

Value

a plot or ggplotly object.

See Also

Other plot_tag: `plot_tag_acceleration()`, `plot_tag_actogram()`, `plot_tag_light()`, `plot_tag_pressure()`, `plot_tag_twilight()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)
})

plot_tag_temperature(tag)
```

plot_tag_twilight	<i>Plot twilight data of a tag</i>
-------------------	------------------------------------

Description

This function display a plot of twilight time series recorded by a tag

Usage

```
plot_tag_twilight(
  tag,
  twilight_line = NULL,
  transform_light = TRUE,
  twl_offset = NULL,
  plot_plotly = TRUE,
  flip_axes = FALSE,
  double_plot = FALSE
)
```

Arguments

tag	a GeoPressureR tag object
twilight_line	a twilight data.frame typically created with <code>path2twilight()</code> which is displayed as a line
transform_light	logical to use a log transformation of light
twl_offset	Shift of the middle of the night compared to 00:00 UTC (in hours). If not provided, it uses the middle of all nights.
plot_plotly	logical to use plotly
flip_axes	logical to place time on the x-axis and dates on the y-axis. Dates increase downward in the flipped display.
double_plot	logical to display a 48-hour window. The second 24-hour half contains the following day's observations.

Value

a ggplot or ggplotly object.

See Also

Other plot_tag: `plot_tag_acceleration()`, `plot_tag_actogram()`, `plot_tag_light()`, `plot_tag_pressure()`, `plot_tag_temperature()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)

  plot_tag_twilight(tag, plot_plotly = TRUE)

  tag <- tag_label(tag, quiet = TRUE)

  plot_tag_twilight(tag)
})
```

plot_twl_calib	<i>Plot twilight calibration diagnostics</i>
----------------	--

Description

This function plots the stacked calibration histogram and fitted density, with calibration stationary periods labeled as known or fitted anchors. Refined fitted anchors remain labeled as fitted. If a `path` is provided, it instead plots the solar zenith angle histogram derived from `tag$twilight` and the path, with the calibration density overlaid.

Usage

```
plot_twl_calib(x, path = NULL, warning_ci = 0.95, plot_plotly = TRUE, ...)

plot_twl_calib_path(tag, path, warning_ci = 0.9973, plot_plotly = TRUE)
```

Arguments

<code>x</code>	a GeoPressureR <code>twl_calib</code> object or a <code>tag</code> object with calibration.
<code>path</code>	a GeoPressureR <code>path</code> or <code>pressurepath</code> data.frame. If provided, plot the path-based calibration diagnostic instead of the stacked calibration histogram.
<code>warning_ci</code>	Width of the symmetric confidence interval used to flag staps with twilights outside the calibrated range (e.g., 0.95 for 2.5%/97.5%).
<code>plot_plotly</code>	logical to use <code>plotly</code> .
<code>...</code>	unused
<code>tag</code>	a GeoPressureR <code>tag</code> object with twilight calibration (used by <code>plot_twl_calib_path()</code>).

Value

a ggplot or ggplotly object

See Also

Other geolight: [geolight_fit_location\(\)](#), [geolight_map\(\)](#), [geolight_solar\(\)](#), [twilight_create\(\)](#), [twilight_label_read\(\)](#), [twilight_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    scale = 5,
    known = data.frame(
      stap_id = 1,
      known_lon = 17.05,
      known_lat = 48.9
    )
  ) |>
  twilight_create() |>
  twilight_label_read() |>
  geolight_map_calibrate()
})

# Histogram + kernel density
plot_twl_calib(tag)

# With fitted location duration of 5 days minimum
tag <- geolight_map_calibrate(tag, fitted_location_duration = 5)

# Path-based diagnostic (uses tag$twilight)
path <- tag$stap[, c("stap_id", "start", "end")]
path$lon <- ifelse(is.na(tag$stap$known_lon), 0, tag$stap$known_lon)
path$lat <- ifelse(is.na(tag$stap$known_lat), 0, tag$stap$known_lat)
plot_twl_calib(tag, path = path, plot_plotly = FALSE)
```

pressurepath_create *Create a pressure path*

Description

pressurepath_create() combines a tag pressure series with a path and retrieves matching ERA5 variables. Positions during flights are linearly interpolated between stationary periods. **pressurepath_create_arco()** and **pressurepath_create_api()** are shortcuts that force the corresponding source.

Usage

```
pressurepath_create(
  tag,
  path = tag2path(tag),
  variable = c("altitude", "surface_pressure"),
  solar_dep = 0,
  era5_dataset = "both",
  preprocess = FALSE,
  workers = "auto",
  quiet = FALSE,
  debug = FALSE,
  source = c("auto", "arco", "api")
)
```

```
pressurepath_create_api(
  tag,
  path = tag2path(tag),
  variable = c("altitude", "surface_pressure"),
  solar_dep = 0,
  era5_dataset = "both",
  preprocess = FALSE,
  workers = "auto",
  quiet = FALSE,
  debug = FALSE
)
```

```
pressurepath_create_arco(
  tag,
  path = tag2path(tag),
  variable = c("altitude", "surface_pressure"),
  solar_dep = 0,
  era5_dataset = "both",
  preprocess = FALSE,
  quiet = FALSE,
  debug = FALSE
)
```

Arguments

tag	A GeoPressureR tag object.
path	A GeoPressureR path data.frame.
variable	ERA5 variables to retrieve, named as in the ERA5 catalogue , plus the derived "altitude". The same names work for every source , but the available set depends on both source and era5_dataset — see pressurepath_variable_available() .
solar_dep	Solar depression angle used to compute sunrise and sunset, or NULL to skip this computation.
era5_dataset	ERA5 product: "both" (default) to use ERA5-Land over land and global ERA5 elsewhere, "land", or "single-levels". Prefer "single-levels"

	whenever <code>variable</code> includes "altitude"; see the <i>Choosing era5_dataset</i> section.
<code>preprocess</code>	Whether to preprocess pressure with <code>geopressure_map_preprocess()</code> .
<code>workers</code>	Number of parallel GeoPressureAPI requests, or "auto".
<code>quiet</code>	Logical to suppress progress messages.
<code>debug</code>	Logical to display request details.
<code>source</code>	Data source: "auto", "arco", or "api".

Value

A `pressurepath` data.frame containing tag pressure, coordinates, requested ERA5 variables, normalised surface pressure, and optional sunrise and sunset times.

Data sources

`source = "arco"` reads ECMWF's Analysis-Ready Cloud-Optimised (ARCO) archive directly. It requires an ECMWF API key and the optional `Rarr` and `ecmwfR` packages. `variable` uses the same names as the hosted API, but the archive carries fewer of them: 20 for `era5_dataset = "single-levels"`, 16 for "land", and the 8 they share for "both", plus the derived "altitude". Use `pressurepath_variable_available()` to list them.

`source = "api"` uses the hosted GeoPressureAPI. It needs no ECMWF key or `Rarr` installation and reaches the full Earth Engine band set: 292 variables for `era5_dataset = "single-levels"` and "both", 70 for "land".

The default, `source = "auto"`, uses ARCO when a key stored by `ecmwfR::wf_set_key()` is available and GeoPressureAPI otherwise. Use `pressurepath_create_arco()` or `pressurepath_create_api()` to select a backend explicitly.

Path and ERA5 processing

Measurements are retained only when `path` contains the surrounding stationary periods. Coordinates during flights are interpolated linearly, while ERA5 values are sampled at the nearest grid cell and hour. `era5_dataset = "single-levels"` uses 0.25 degree global ERA5, "land" uses 0.1 degree ERA5-Land, and "both" selects ERA5-Land over land and global ERA5 over water.

Surface pressure is returned in hPa and normalised to the mean tag pressure within each stationary-period and elevation-label group. Observations labelled "discard" are excluded from those means. If `solar_dep` is not NULL, local sunrise and sunset are added.

Pressure-derived altitude

When altitude is produced, GeoPressureR uses the same barometric relation as **GeoPressureAPI**:

$$z_{\text{tag}} = z_{\text{ERA5}} + \frac{T_{\text{ERA5}}}{L_b} \left[\left(\frac{P_{\text{tag}}}{P_{\text{ERA5}}} \right)^{-RL_b/(gM)} - 1 \right].$$

Here, P_{tag} is tag pressure, while P_{ERA5} , T_{ERA5} , and z_{ERA5} are ERA5 surface pressure, 2 m temperature, and model-surface elevation. Model-surface elevation is obtained from

ERA5 surface geopotential divided by standard gravity. The constants are the standard temperature lapse rate $L_b = -0.0065$ K/m, universal gas constant $R = 8.31432$ J/(mol K), standard gravity $g = 9.80665$ m/s², and molar mass of dry air $M = 0.0289644$ kg/mol. Pressure is converted to Pa internally and altitude is returned in metres above mean sea level.

Choosing era5_dataset

Use `era5_dataset = "single-levels"` whenever altitude is computed.

ERA5-Land's `surface_pressure` is not the exact hydrostatic image of the orography ERA5-Land publishes as `geopotential` — the two disagree by up to ~10 hPa in steep terrain. The z_{ERA5} term therefore fails to cancel from the relation above and the whole discrepancy lands in the retrieved altitude. Measured against 41,653 hourly station-pressure observations from 271 NOAA ISD stations (2–3576 m, Alps, July 2020), compared with surveyed station elevation:

era5_dataset	bias	MAE	RMSE
"single-levels"	−0.6 m	9.0 m	27.4 m
"land" / "both"	+2.5 m	55.3 m	76.9 m

ERA5-Land replays the land component of ERA5 and assimilates no atmospheric observations of its own, so it carries no independent information about absolute altitude. Substituting the orography implied by its own surface pressure removes the error exactly and reproduces `"single-levels"` to 0.1 m.

`"land"` and `"both"` remain the defaults for backward compatibility. They are appropriate for retrieving other variables at 0.1 degree resolution, and for `geopressure_map()`, whose pressure mismatch is differential and unaffected.

Accuracy

With `era5_dataset = "single-levels"` the altitude error separates into two parts that behave very differently. A static per-site offset (median 3.7 m, p90 15 m) reflects station metadata and sub-grid terrain and does not vary in time, while the temporal scatter (median SD 3.1 m, p90 7.7 m) is the actual reanalysis error. Expect roughly **3 m for relative altitude changes at a fixed location** and **10 m mean absolute error for absolute altitude**, degrading to tens of metres in steep terrain — where it is a fixed offset rather than noise. Precision is nearly independent of flight altitude: de-biased RMSE stays 2–7 m up to 1000 m above the model surface and about 12 m at 1000–3000 m above it.

A note on accumulated variables

`total_precipitation`, `surface_solar_radiation_downwards` and `surface_thermal_radiation_downwards` do **not** mean the same thing across backends when `era5_dataset = "land"`. GeoPressureAPI serves Earth Engine's ERA5-Land collection, where these bands are overwritten by their `*_hourly` counterparts and are therefore hourly increments. ARCO serves ECMWF's own ERA5-Land fields, which accumulate from 00 UTC. For `era5_dataset = "single-levels"` both are hourly and agree. Take the difference between consecutive hours if you need increments from ARCO over land.

ECMWF API key

ARCO access and CDS downloads use the same ECMWF Personal Access Token. Create an ECMWF account, copy the token from your [Climate Data Store profile](#), accept the dataset licence terms, and store the token once in your system keyring:

```
ecmwf::wf_set_key("your-personal-access-token")
```

Do not put the token in a script or commit it to version control. GeoPressureR retrieves the stored token with `ecmwf::wf_get_key()` for all ECMWF-backed functions, including `geopressure_timeseries()`, `pressurepath_create()`, and `tag_download_wind()`.

See Also

Other pressurepath: `geopressure_timeseries()`, `plot_pressurepath()`, `pressurepath_variable_available()`

Examples

```
pressurepath <- pressurepath_create(tag, path)
```

```
pressurepath_variable_available
```

List the ERA5 variables a backend and product can return

Description

The variables available to `pressurepath_create()` depend on **both** the backend and the ERA5 product, and the sets genuinely differ. GeoPressureAPI serves whatever bands the underlying Earth Engine collection carries: 292 for ERA5 single levels against 70 for ERA5-Land, with only 44 in common. `era5_dataset = "both"` mosaics ERA5-Land on top of ERA5 but returns ERA5's band namespace, so its set equals `"single-levels"` and ERA5-Land-only bands are dropped. The ARCO backend reads ECMWF's zarr stores directly and currently maps only surface pressure and 2 m temperature, so it can return `"surface_pressure"` and the derived `"altitude"`.

Usage

```
pressurepath_variable_available(
  source = c("api", "arco"),
  era5_dataset = c("single-levels", "land", "both")
)
```

Arguments

```
source      Backend: "api" or "arco".
era5_dataset ERA5 product: "single-levels", "land", or "both".
```

Details

Requesting a variable the chosen product does not have used to fail silently: GeoPressureAPI returned 200 `success` with every array empty — not just the missing one — which arrived as a `pressurepath` whose ERA5 columns were all NA. `pressurepath_create()` now validates the request up front with this list.

Value

A character vector of variable names accepted by `variable`, including "altitude".

See Also

Other pressurepath: `geopressure_timeseries()`, `plot_pressurepath()`, `pressurepath_create()`

Examples

```
length(pressurepath_variable_available("api", "single-levels"))
length(pressurepath_variable_available("api", "land"))
pressurepath_variable_available("arco", "land")
```

<code>print.bird</code>	<i>Print a bird object</i>
-------------------------	----------------------------

Description

This function displays the information of a `bird` object.

Usage

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

Arguments

<code>x</code>	a GeoPressureR <code>bird</code> object.
<code>...</code>	arguments passed to other methods

Value

`bird` is returned invisibly and unchanged

See Also

Other bird: `bird_create()`

Examples

```
# Using AVONET dataset
bird_create("Acrocephalus arundinaceus")
```

print.graph	<i>Print a graph object</i>
-------------	-----------------------------

Description

This function displays the information of a **graph** object.

Usage

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

Arguments

x	a GeoPressureR graph object.
...	arguments passed to other methods

Value

graph is returned invisibly and unchanged

See Also

Other graph: [graph_add_wind\(\)](#), [graph_create\(\)](#), [graph_marginal\(\)](#), [graph_most_likely\(\)](#), [graph_set_movement\(\)](#), [graph_simulation\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  twilight_create() |>
  twilight_label_read() |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    known = data.frame(stap_id = 1, known_lon = 17.05, known_lat = 48.9)
  ) |>
  geopressure_map(quiet = TRUE) |>
  geolight_map(quiet = TRUE)
})

graph <- graph_create(tag, quiet = TRUE)

print(graph)
```

print.map	<i>Print a map object</i>
-----------	---------------------------

Description

This function displays the information of a `map` object.

Usage

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

Arguments

<code>x</code>	a GeoPressureR <code>map</code> object
<code>...</code>	arguments passed to other methods

Value

`map` is returned invisibly and unchanged

See Also

Other map: [map_add_mask_water\(\)](#), [map_create\(\)](#), [rast.map\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    tag_set_map(
      extent = c(-16, 23, 0, 50),
      scale = 4
    ) |>
    geopressure_map(quiet = TRUE)
})

print(tag$map_pressure)
```

print.param	<i>Print a param list</i>
-------------	---------------------------

Description

This function displays the information of a `param` list.

Usage

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

Arguments

<code>x</code>	a <code>GeoPressureR</code> <code>param</code> list.
<code>...</code>	arguments passed to other methods

Value

`param` is returned invisibly and unchanged

See Also

Other param: [param_create\(\)](#)

Examples

```
# Display the default parameters used in the package.  
param <- param_create("18LX", default = TRUE)  
  
print(param)
```

print.tag	<i>Print a tag object</i>
-----------	---------------------------

Description

This function displays the information of a `tag` object.

Usage

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

Arguments

x a GeoPressureR **tag** object
... further arguments passed to or from other methods.

Value

tag is returned invisibly and unchanged

See Also

Other tag: [read_stap\(\)](#), [tag_create\(\)](#), [tag_set_map\(\)](#), [tag_stap_daily\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)
})

print(tag)
```

rast.map

Construct a SpatRaster from a map

Description

This function convert a GeoPressureR **map** object into a [terra::SpatRaster](#) with the data of each stationary periods stored in a different layer.

Usage

```
rast.map(x, names = glue::glue("#{map$stap$stap_id}"), crs = "epsg:4326", ...)
```

Arguments

x a GeoPressureR **map** object
names names of the SpatRaster layers created. See [terra::names](#).
crs character. Description of the Coordinate Reference System (map projection) in PROJ.4, WKT or **authority:code** notation. See [crs](#). If this argument is missing, and the x coordinates are within -360 .. 360 and the y coordinates are within -90 .. 90, longitude/latitude is assigned
... additional parameters for [terra::rast](#)

Value

A [terra::SpatRaster](#) object.

See Also

Other map: [map_add_mask_water\(\)](#), [map_create\(\)](#), [print.map\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
    extent = c(-16, 23, 0, 50),
    scale = 4
  ) |>
  geopressure_map(quiet = TRUE)
})

rast.map(tag$map_pressure)
```

read_stap	<i>Read or validate stap data</i>
-----------	-----------------------------------

Description

Reads a CSV file defining coarse stationary periods (**stap0**) or validates a data.frame, including date parsing, ordering, and overlap checks.

Usage

```
read_stap(x, required_cols = c("start", "end"))
```

Arguments

x a **stap** data.frame, a CSV path, or a tag id (character scalar). If **x** is a tag id, the default file path is `./data/stap-label/{id}.csv`.

required_cols character vector of required columns (default: `c("start", "end")`).

Value

A data.frame containing at least the required columns, sorted by **start**.

See Also

Other tag: [print.tag\(\)](#), [tag_create\(\)](#), [tag_set_map\(\)](#), [tag_stap_daily\(\)](#)

Examples

```
# From a data.frame
stap_df <- data.frame(
  start = as.POSIXct(c("2020-01-01", "2020-01-03"), tz = "UTC"),
  end = as.POSIXct(c("2020-01-02", "2020-01-04"), tz = "UTC")
)
read_stap(stap_df)

# From a CSV path
# read_stap("./data/stap-label/18LX.csv")
```

speed2bearing

Compute the bearing of a speed vector

Description

This function convert a speed vector represented in complex number into a bearing angle ($0^\circ=N$; $90^\circ=E\dots$). This conversion is needed because trigonometric angle are different than bearing.

Usage

```
speed2bearing(
  speed,
  speed_ref = complex(real = 0, imaginary = 1),
  positive = TRUE
)
```

Arguments

speed	speed as complex value
speed_ref	reference vector of the angle. Default is the North in order to return bearing. Use speed_ref = 1 for trigonometric orientation (clockwise from East).
positive	logical to ensure the bearing is positive between 0-360 degree.

Value

bearing angle in degree

Examples

```
speed2bearing(complex(real = 10, imaginary = 5))
```

speed2prob	<i>Compute probability of a bird speed</i>
------------	--

Description

Converts a speed (airspeed or ground speed) to a probability based on the movement model.

Usage

```
speed2prob(speed, movement)
```

Arguments

speed	airspeed or ground speed in km/h
movement	a list of the movement model parameter defined with <code>graph_set_movement</code>

Value

Probability values corresponding to the speed provided.

See Also

Other movement: [bird_create\(\)](#), [graph_set_movement\(\)](#), [plot_graph_movement\(\)](#), [tag_download_wind\(\)](#)

Examples

```
speed <- seq(1, 120)
low_speed_fix <- 20 # minimum speed allowed
prob <- speed2prob(
  speed,
  list(
    method = "gamma",
    shape = 7,
    scale = 7,
    low_speed_fix = low_speed_fix,
    zero_speed_ratio = 0
  )
)
plot(speed, prob,
  type = "l",
  xlab = "Groundspeed [km/h]",
  ylab = "Probability"
)
abline(v = low_speed_fix)

# Using airspeed
bird <- bird_create("Acrocephalus arundinaceus")
prob <- speed2prob(
  speed,
  list(
```

```

    method = "power",
    bird = bird,
    power2prob = \ (power) (1 / power)^3,
    low_speed_fix = low_speed_fix,
    zero_speed_ratio = 0
  )
)
plot(speed, prob, type = "l", xlab = "Airspeed [km/h]", ylab = "Probability")
abline(v = low_speed_fix)

```

stap2duration	<i>Compute duration of stationary periods</i>
---------------	---

Description

This function returns the duration between `stap$start` and `stap$end`.

The function can be used with any `data.frame` containing `start` and `end` as `POSIXct` (e.g., `flight` or `path`).

Usage

```
stap2duration(stap_path, units = "days", return_numeric = TRUE)
```

Arguments

<code>stap_path</code>	A <code>stap</code> <code>data.frame</code> .
<code>units</code>	character string. Units in which the results are desired. Can be abbreviated. See <code>difftime()</code>
<code>return_numeric</code>	logical to return the duration as a numeric rather than with a duration format.

Value

Vector of duration

Examples

```

# Create fake stap
stap <- data.frame(
  start = seq(as.POSIXct("1990-01-01"), as.POSIXct("1991-01-01"), length.out = 13)
)
stap$end <- stap$start + 60 * 60 * 24 * 30
stap$stap_id <- seq_len(nrow(stap))

stap2duration(stap)

stap2duration(stap, units = "mins", return_numeric = FALSE)

```

stap2flight

Compute flights from stationary periods

Description

Convert a stationary period data.frame **stap** into a flight data.frame or list. Flight are computed as the difference between the end of a stationary period to the start of the next one. Because the pressure/acceleration is labelled for "in flight", the bird was already in flight before the first label and after the last label. We account for this by adding to all flights duration half the temporal resolution of the sensor.

You can compute the flight between specific stationary periods using **include_stap_id**. In this case, the flight duration is computed as the sum of individual flights in between.

You can return the flight as a data.frame or as a list if you want to retrieve the information of all individual flight between the **include_stap_id**.

Usage

```
stap2flight(
  stap,
  include_stap_id = NULL,
  format = "df",
  units = "hours",
  return_numeric = TRUE
)
```

Arguments

stap	a stationary period data.frame (see tag_label_stap()).
include_stap_id	vector of the stationary period stap_id to consider in the flight. Default is to use stap\$stap_id[stap\$include] or stap\$stap_id if include is not available in stap .
format	character to return a list "list" or a data.frame "df" (see description)
units	character string. Units in which the results are desired. Can be abbreviated. See difftime()
return_numeric	logical to return the duration as a numeric rather than with a duration format.

Value

A list or a data.frame (see description) containing

- **start**: Start time of the (first) flight
- **end**: End time of the (last) flight
- **stap_id_s**: Source **stap_id** (i.e, start)

- `stap_id_t`: Target `stap_id` (i.e, end)
- `duration`: (Sum of the) duration of flight(s)
- (`n`: Number of flights) The value in brackets are only for the data.frame

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

# By default, return a data.frame of all individual flights
knitr::kable(stap2flight(tag$stap))

# Compute the total flight between stap 1,3 and 5. Sum flight duration in between.
knitr::kable(stap2flight(tag$stap, include_stap_id = c(1, 3, 5)))

# Can also return as a list of data.frame to access individual flights information.
knitr::kable(stap2flight(tag$stap,
  include_stap_id = c(1, 3, 5), format = "list",
  units = "secs"
))
```

tag_assert	<i>Assert the status of a tag</i>
------------	-----------------------------------

Description

This function check the condition of a `tag` object.

Usage

```
tag_assert(tag, condition = "tag", type = "abort")
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object.
<code>condition</code>	condition to assert <code>tag</code> for. One of "tag" (default), "read", "pressure", "light", "acceleration", "temperature_internal", "temperature_external", "magnetic", "label", "stap", "setmap", "map_pressure", "map_pressure_mse", "map_pressure_mask", "map_light", "map_light_twl", "mask_water", "map_magnetic", "map_magnetic_intensity", "map_magnetic_inclination", "twl_calib", and "twilight".
<code>type</code>	Message type to display. One of "abort" (default), "warn", "inform", or "logical" (return TRUE/FALSE without messaging).

Value

logical indicating the `tag` object has the relevant element

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

tag_assert(tag)

tag_assert(tag, "stap")

tag_assert(tag, "setmap", type = "warn")

tag_assert(tag, "map_pressure", type = "inform")

tag_assert(tag, "setmap", type = "logical")
```

tag_create	Create a tag object
------------	---------------------

Description

Read sensor measurements into a GeoPressureR **tag** object. Optionally correct timestamps for clock offset and drift, then crop to the recording period of interest.

Usage

```
tag_create(
  id,
  manufacturer = NULL,
  crop_start = NULL,
  crop_end = NULL,
  directory = glue::glue("./data/raw-tag/{id}"),
  pressure_file = NULL,
  light_file = NULL,
  acceleration_file = NULL,
  temperature_external_file = NULL,
  temperature_internal_file = NULL,
  magnetic_file = NULL,
  assert_pressure = TRUE,
  quiet = FALSE,
  time_shift = 0,
  time_drift = 0,
  time_reference = NULL
)
```

Arguments

id	Unique tag identifier.
manufacturer	Data format: NULL (automatic), "soi", "migratetech", "bas", "lund", "prestag", or "tabular".
crop_start	Inclusive start of the corrected recording period, as POSIXct or character in UTC. NULL keeps all earlier data.
crop_end	Exclusive end of the corrected recording period, as POSIXct or character in UTC. NULL keeps all later data.
directory	Directory containing the original sensor files.
pressure_file	Pressure input; tabular values must be in hPa. See Sensor files and Tabular input for input formats.
light_file	Optional light input.
acceleration_file	Optional acceleration input.
temperature_external_file	Optional external or air-temperature input.
temperature_internal_file	Optional internal or body-temperature input.
magnetic_file	Optional magnetic and acceleration input with axis columns.
assert_pressure	Logical; require pressure data in the returned tag.
quiet	Logical; hide progress messages and sampling-interval warnings.
time_shift	Constant correction in hours. A numeric value for all sensors or a named list for individual sensors (pressure , light , acceleration , temperature_external , temperature_internal , magnetic). Default is zero.
time_drift	Linear correction in hours per year (365.25 days). A numeric value for all sensors or a named list for individual sensors. Default is zero; -12 / 60 subtracts 12 minutes per year. Convert hours per 30-day month by multiplying by 365.25 / 30.
time_reference	Original recorded timestamp at which drift correction is zero, as POSIXct or character in UTC. NULL uses the earliest measurement across all sensors. Supply a timestamp if the clock was synchronised at another time. Only time_shift is added at the reference.

Value

A GeoPressureR **tag** containing **param** (see [param_create\(\)](#)) and the available sensor tables. Each sensor table has a UTC POSIXct **date** column and the following measurement columns:

- **pressure**: value in hPa.
- **light**, **temperature_external**, **temperature_internal**: value.

- **acceleration**: **value** and optionally **mean_acceleration_z**. For SOI, **value** is the sum of successive z-axis acceleration differences over 32 measurements at 10 Hz (jiggle); **mean_acceleration_z** is their mean z-axis acceleration.
- **magnetic**: **magnetic_x**, **magnetic_y**, **magnetic_z**, **acceleration_x**, **acceleration_y**, **acceleration_z**.

Workflow

Data are **read -> corrected -> cropped**. Keep the original files and specify clock corrections in **tag_create()**. All sensor timestamps are rounded to the nearest millisecond after correction and before cropping, including when no correction is supplied. A warning identifies sensors where rounding makes distinct timestamps identical; all measurements are kept. Pressure is required by default; use **assert_pressure = FALSE** for tags without pressure data.

Sensor files

By default, files are read from `./data/raw-tag/{id}` and the manufacturer is detected from the directory where possible. Set **manufacturer** explicitly when needed, including for PresTag. Sensor file arguments accept a file path or filename pattern, such as `*.pressure`. Supported formats are:

- SOI ("soi"): pressure `*.pressure`, light `*.glf`, acceleration `*.acceleration`, external temperature `*.temperature` or `*.airtemperature`, internal temperature `*.bodytemperature`, and magnetic data `*.magnetic`.
- Migratetech ("migratetech"): pressure and acceleration `*.deg`, light `*.lux`.
- BAS ("bas"): light `*.lig` only; use **assert_pressure = FALSE**.
- Lund CAnMove ("lund"): pressure `*_press.xlsx`, light and acceleration `*_acc.xlsx`.
- PresTag ("prestag"): pressure `*.txt`.

Manufacturer readers detect available sensor files when their arguments are NULL. For SOI, use NA to skip a sensor.

Tabular input

With **manufacturer = "tabular"**, each sensor argument accepts a `data.frame`, `tibble`, or CSV path. Tables need a UTC POSIXct **date** column and a **value** column; pressure values must be in hPa. Magnetic tables instead use the axis columns listed under Value below.

CSV files use **datetime** instead of **date**, with UTC timestamps such as "2025-01-01T00:00:00"; fractional seconds are supported. CSV files named `pressure.csv`, `light.csv`, etc. can be detected automatically from **directory**. Omit optional tabular sensors with NULL. An in-memory **pressure_file** also selects tabular input automatically.

Clock correction

- **time_shift** adds a constant offset in hours.
- **time_drift** adds a linear correction in hours per year (365.25 days).

- **time_reference** is the original recorded time at which drift correction is zero. By default, it is the earliest measurement across all sensors, before shifts or cropping.

Positive corrections add time; negative corrections subtract time. Each correction accepts one value for all sensors or a named list, such as `list(pressure = -12 / 60, light = 0)`. Sensors omitted from a list receive zero; entries for absent sensors are ignored. All sensors share the reference, regardless of recording duration. Drift uses the original timestamps, before adding the shift, and extends linearly before an explicit reference.

Correction settings and the reference used for drift are recorded in `tag$param$tag_create`. When changing corrections, rerun the analysis from the original data and regenerate labels that rely on exact timestamps, including TRAINSET labels.

Cropping

Cropping uses corrected UTC timestamps: `crop_start` is inclusive and `crop_end` is exclusive. Leave either boundary as NULL to keep all data on that side.

See Also

[GeoPressureManual](#)

Other tag: `print.tag()`, `read_stap()`, `tag_set_map()`, `tag_stap_daily()`

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  # Read available sensors.
  tag <- tag_create("18LX", quiet = TRUE)

  # Crop to the period of interest.
  tag <- tag_create("18LX",
    crop_start = "2017-08-01", crop_end = "2017-08-05", quiet = TRUE
  )

  # Select files explicitly when several match the same pattern.
  tag <- tag_create("CB621",
    pressure_file = "CB621_BAR.deg", light_file = "CB621.lux", quiet = TRUE
  )

  # Correct selected sensors using a shared recorded reference.
  tag <- tag_create("18LX",
    time_shift = list(light = 2), time_drift = list(pressure = -12 / 60),
    time_reference = "2017-06-20", quiet = TRUE
  )
})

# Read an in-memory table and apply the same correction to all sensors.
pressure <- data.frame(
  date = as.POSIXct("2025-01-01", tz = "UTC") + (0:48) * 1800,
  value = rep(1000, 49)
)
tag <- tag_create("example", pressure_file = pressure,
  time_shift = 2, time_drift = -12 / 60, quiet = TRUE
```

)

tag_download_wind	<i>Download flight data</i>
-------------------	-----------------------------

Description

This function downloads data associated to each flight from the ERA5 hourly pressure levels ([doi:10.24381/cds.bd0915c6](https://doi.org/10.24381/cds.bd0915c6)) with the [Climate Data Store \(CDS\)](#) and through the [ecmwf R package](#).

Any variable available from the ERA5 pressure level can be downloaded.

The flights are determined from the stationary periods classified `tag$stap`. It requests a single file for each flight using the exact time (hourly basis) and pressure (altitude). To make the download more efficient, `wf_request_batch()` is used to download files in parallel. CDS queue limits vary with the current system workload. If CDS reports that queued requests are temporarily limited, reduce `workers` and try again.

More information [in the GeoPressureManual](#).

Usage

```
tag_download_wind(
  tag,
  extent = tag$param$tag_set_map$extent,
  include_stap_id = NULL,
  variable = c("u_component_of_wind", "v_component_of_wind"),
  file = function(stap_id, tag_id) {

    glue::glue("./data/wind/{tag_id}/{tag_id}_{stap_id}.nc")
  },
  overwrite = FALSE,
  workers = 19,
  cds_token = lifecycle::deprecated(),
  ...
)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object.
<code>extent</code>	geographical extent of the map on which the likelihood will be computed. Vector of length 4 <code>c(xmin, xmax, ymin, ymax)</code> or <code>c(W, E, S, N)</code> .
<code>include_stap_id</code>	stationary period identifiers of the start of the flight to download. Default is to download all flights.

variable	list of variables to download from the ERA5 pressure level : "u_component_of_wind", "v_component_of_wind", "temperature", "fraction_of_cloud_cover", "relative_humidity", "vertical_velocity", "specific_cloud_ice_water_content", "specific_cloud_liquid_water_content", "specific_humidity", "specific_rain_water_content", "specific_snow_water_content", "divergence", "geopotential", "ozone_mass_mixing_ratio", "potential_vorticity", "vorticity". The same names are used by <code>edge_add_wind()</code> .
file	absolute or relative path of the ERA5 wind data file to be downloaded. Function taking as arguments (1) the stationary period identifier and (2) the tag_id.
overwrite	logical. If TRUE, file is overwritten.
workers	maximum number of simultaneous requests submitted to CDS. Defaults to 19. CDS queue limits are dynamic; use a lower value if queued requests are temporarily limited.
cds_token	[Deprecated] Enter the API token with <code>ecmwfr::wf_set_key()</code>
...	Arguments passed on to <code>ecmwfr::wf_request_batch</code>
request	nested list with query parameters following the layout as specified on the ECMWF APIs page
user	user (default = "ecmwf") provided by the ECMWF data service, used to retrieve the token set by <code>wf_set_key</code>
transfer	logical, download data TRUE or FALSE (default = TRUE)
path	path where to store the downloaded data
time_out	how long to wait on a download to start (default = 3600 seconds).
retry	polling frequency of submitted request for downloading (default = 30 seconds).
job_name	optional name to use as an RStudio job and as output variable name. It has to be a syntactically valid name.
verbose	show feedback on processing
request_list	a list of requests that will be processed in parallel.
total_timeout	overall timeout limit for all the requests in seconds.

Value

The path of the downloaded (requested file) or the an R6 object with download/transfer information

ECMWF API key

ARCO access and CDS downloads use the same ECMWF Personal Access Token. Create an ECMWF account, copy the token from your **Climate Data Store profile**, accept the dataset licence terms, and store the token once in your system keyring:

```
ecmwfr::wf_set_key("your-personal-access-token")
```

Do not put the token in a script or commit it to version control. GeoPressureR retrieves the stored token with `ecmwfr::wf_get_key()` for all ECMWF-backed functions, including `geopressure_timeseries()`, `pressurepath_create()`, and `tag_download_wind()`.

See Also

`wf_request_batch()` , `GeoPressureManual`

Other movement: `bird_create()`, `graph_set_movement()`, `plot_graph_movement()`, `speed2prob()`

Examples

```
tag_download_wind(tag)
```

tag_label

Label a tag object

Description

This function performs the following operations:

1. Read label file with `tag_label_read()` and assign the label to a new column in each sensor data.frame
2. Compute the stationary period `tag$stap` from the label and assign the corresponding `stap_id` on all sensors data.frame with `tag_label_stap()`

If the label file does not exist, the function will suggest to create it with `tag_label_write()` and use `tag_label_auto()` if acceleration data exists.

Usage

```
tag_label(
  tag,
  file = glue::glue("./data/tag-label/{tag$param$id}-labeled.csv"),
  quiet = FALSE,
  warning_flight_duration = lifecycle::deprecated(),
  warning_stap_duration = lifecycle::deprecated(),
  ...
)
```

Arguments

<code>tag</code>	a GeoPressure <code>tag</code> object.
<code>file</code>	Absolute or relative path of the label file.
<code>quiet</code>	logical to display message.
<code>warning_flight_duration</code>	<code>lifecycle::deprecated()</code>
<code>warning_stap_duration</code>	<code>lifecycle::deprecated()</code>
<code>...</code>	<code>lifecycle::deprecated()</code>

Value

Same `tag` list with

(1) a `stap` data.frame describing the STationary Period:

- `stap_id` unique identifier in increasing order 1,...,n
- `start` start date of each stationary period
- `end` end date of each stationary period

(2) an additional `label` and `stap_id` column on the sensor data.frame:

- `date` datetime of measurement as POSIXt
- `value` sensor measurement
- `label` indicates the observation to be discarded ("discard" and "flight") as well as grouped by elevation layer (`elev_*`)
- `stap_id` stationary period of the measurement matching the `tag$stap`.

See Also

[GeoPressureManual](#)

Other `tag_label`: [tag_label_auto\(\)](#), [tag_label_read\(\)](#), [tag_label_stap\(\)](#), [tag_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)

  print(tag)

  tag <- tag_label(tag)

  print(tag)

  # The labelled `tag` contains additional column on the sensor data.frame
  str(tag)
})
```

Description

This function uses acceleration data to classify migratory flights. The function uses a $k=2$ mean clustering (`kmeans()`) to identify high activity periods. Periods of high activity lasting more than `min_duration` are then considered to be migratory flight.

Additionally, we perform a post-processing step to improve the classification of low-activity period happening during a migration flight (e.g. gliding phase):

1. Mid-activity surrounded by flights. The kmeans classification classify high-activity all values above 50% of the distance between the two clusters. Here, we classify as mid-activity non-flight activity data which value are greater than `thr_reclassify` of the distance between the two kmeans clusters. Any periods of mid-activity surrounded by flight are also considered as flights.
2. To avoid stap lasting only a few datapoints, we also classify any datapoint which is surrounded by a flight before and after, within a window of $\pm$ `post_proc_window` points. As a result, no stap will be shorter than $(2 * \text{post_proc_window} + 1) * \text{dt}$.

This function is inspired by the function `classify_flap` from the [PAMlr package](#).

Usage

```
tag_label_auto(
  tag,
  min_duration = 30,
  thr_reclassify = 0.1,
  post_proc_window = 2
)
```

Arguments

<code>tag</code>	a GeoPressure <code>tag</code> object.
<code>min_duration</code>	Minimal duration (in minutes) to consider a high activity as migratory flight.
<code>thr_reclassify</code>	Post-processing threshold of activity considered for re-classification. Typically between 0 and 0.5 (no effect). See post-processing for details.
<code>post_proc_window</code>	Post-processing windows considered for re-classification. Typically 0 (no effect) to 5 in unit of the temporal resolution. See post-processing for details.

Value

Same data logger list than input `tag`, but with the column `label` filled with "flight" in the acceleration data.frame when a sustained high-activity period is detected.

See Also

[GeoPressureManual](#)

Other tag_label: [tag_label\(\)](#), [tag_label_read\(\)](#), [tag_label_stap\(\)](#), [tag_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)

  tag <- tag_label_auto(tag, min_duration = 15)
  str(tag$acceleration)
})
```

tag_label_read	<i>Read a tag label file</i>
----------------	------------------------------

Description

This function reads an exported csv file from **TRAINSET** and updates the data logger dataset `tag`.

Usage

```
tag_label_read(
  tag,
  file = glue::glue("./data/tag-label/{tag$param$id}-labeled.csv")
)
```

Arguments

<code>tag</code>	a GeoPressure <code>tag</code> object.
<code>file</code>	Absolute or relative path of the label file.

Value

Same data logger list as input, updated with the labels `tag$pressure$label` and optionally `tag$acceleration$label`.

See Also

[GeoPressureManual](#)

Other `tag_label`: [tag_label\(\)](#), [tag_label_auto\(\)](#), [tag_label_stap\(\)](#), [tag_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)

  tag <- tag_label_read(tag)

  str(tag)
})
```

tag_label_stap	Create stationary periods from a tag label
----------------	--

Description

This function computes the stationary periods from the pressure and/or acceleration label data. In most case, this function should be run directly after `tag_label_read()` in order to update `tag` to correspond to the new label file.

A stationary period is a period during which the bird is considered static relative to the spatial resolution of interest (~10-50km). They are defined by being separated by a flight of any duration (label "flight"). The `stap_id` is an integer value for stationary periods and decimal value for flight. The `stap_id` is added as a new column to each sensor data.

If an acceleration data.frame is present and contains a column `label`, the stationary period will be computed from it, otherwise, it uses the pressure data.frame.

Usage

```
tag_label_stap(
  tag,
  quiet = FALSE,
  warning_flight_duration = lifecycle::deprecated(),
  warning_stap_duration = lifecycle::deprecated()
)
```

Arguments

<code>tag</code>	a GeoPressure <code>tag</code> object.
<code>quiet</code>	logical to display warning message.
<code>warning_flight_duration</code>	<code>lifecycle::deprecated()</code>
<code>warning_stap_duration</code>	<code>lifecycle::deprecated()</code>

Value

`tag` is return with (1) a new data.frame of stationary periods `tag$stap` and (2) a new column `stap_id` for each sensor data.

See Also

[GeoPressureManual](#)

Other tag_label: [tag_label\(\)](#), [tag_label_auto\(\)](#), [tag_label_read\(\)](#), [tag_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label_read()

  tag <- tag_label_stap(tag)

  str(tag)

  str(tag$stap)
})
```

tag_label_write	<i>Write a tag label file</i>
-----------------	-------------------------------

Description

This function writes the csv file of labelled activity and pressure which can be read with **TRAINSET**. If no label data exist, it will first initialize the label data.

Optionally, it can also export a reference dataset for pressure `tag$pressure$ref` as another series to be visualized on TRAINSET, but without impacting the labelling process.

Usage

```
tag_label_write(
  tag,
  file = glue::glue("./data/tag-label/{tag$param$id}.csv"),
  quiet = FALSE
)
```

Arguments

tag	a GeoPressure tag object.
file	Absolute or relative path of the label file to be saved.
quiet	logical to display message.

Value

The file pathname is return invisibly

See Also

[GeoPressureManual](#)

Other tag_label: [tag_label\(\)](#), [tag_label_auto\(\)](#), [tag_label_read\(\)](#), [tag_label_stap\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE)

  # Writing unlabelled tag will initialize the labelling for trainset
  tag_label_write(tag)

  # Writing unlabelled tag will initialize the labelling for trainset
  tag <- tag_label_auto(tag)
  tag_label_write(tag)

  # Writing labelled tag will use the existing labels
  tag <- tag_label(tag)
  tag_label_write(tag)
})
```

tag_set_map	<i>Configure the spatial and temporal parameters of the map of a tag object</i>
-------------	---

Description

This function adds to `tag` the parameters defining the 3D grid of the map. The spatial parameters (`extent` and `scale`) define the **geographical dimensions of the map**, and the **temporal dimension** is defined based on the stationary periods built using the labels. `include_stap_id` and `include_min_duration` can be used to limit which stationary periods are computed and modelled in the analysis. By default, all stationary periods are included.

In addition, `tag` offers the possibility to define **known** locations (e.g., equipment or retrieval site). These can only be defined at the level of a stationary period (i.e., assuming constant position during the whole stationary period) but you can define as many known stationary periods as you wish. Because the index of the last stationary period is generally unknown, you can use negative indexing in `known`, i.e., `known$stap_id = -1` will be converted to `nrow(tag$stap)`.

By default, no likelihood map will be computed for these stationary periods and the trajectory model will be more constrained, saving significant computational time. You can change this using the `compute_known` parameter in `geopressure_map()`.

Usage

```
tag_set_map(
  tag,
  extent,
  scale = 10,
  known = data.frame(stap_id = integer(), known_lat = double(), known_lon = double()),
  include_stap_id = NULL,
```

```

    include_min_duration = 0
  )

```

Arguments

tag	a GeoPressureR tag object.
extent	geographical extent of the map on which the likelihood and graph model will be computed. Vector of length 4 <code>c(xmin, xmax, ymin, ymax)</code> or <code>c(W, E, S, N)</code> .
scale	number of pixels per 1° latitude-longitude. For instance, <code>scale = 10</code> for a resolution of 0.1° (~10km) and <code>scale=4</code> for a resolution of 0.25° (~30km). To avoid interpolating the ERA5 data, the scale should be equal to or smaller than 10. Read more about scale on the Google earth Engine documentation .
known	data.frame containing the known positions of the bird (e.g., equipment or retrieval site) with columns <code>stap_id</code> , <code>known_lat</code> and <code>known_lon</code> . You can set position of the last stationary period using <code>stap_id = -1</code> . Also accept list which are converted as data.frame.
include_stap_id	vector of <code>stap_id</code> defining which stationary period to model, that is, to compute in the likelihood map and use in the graph.
include_min_duration	minimum duration threshold of stationary periods to include (in hours).

Value

A GeoPressureR **tag** object with:

- **stap**: Data.frame of all stationary periods with three new columns: `known_lat` and `known_lon` define the known position during these stationary periods, and `include` defines whether the likelihood map of this stationary period should be computed and later used in the graph.
- **extent** same as input parameter `extent`
- **scale** same as input parameter `scale`

See Also

[GeoPressureManual](#)

Other tag: [print.tag\(\)](#), [read_stap\(\)](#), [tag_create\(\)](#), [tag_stap_daily\(\)](#)

Examples

```

withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

# Default tag
tag_default <- tag_set_map(tag, c(-16, 23, 0, 50))

```

```

print(tag_default)

# Customized tag, with coarse grid scale, known position for the first stationary
# period and considering only the stationary periods lasting more than 20hours.
tag_custom <- tag_set_map(tag,
  extent = c(-16, 23, 0, 50),
  scale = 1,
  include_min_duration = 20,
  known = data.frame(
    stap_id = 1,
    known_lon = 17.05,
    known_lat = 48.9
  )
)

print(tag_custom)

```

tag_stap_daily	<i>Create stationary periods from twilight midpoints</i>
----------------	--

Description

Construct stationary periods from labelled sunrise and sunset times. Each boundary is the midpoint of an assumed movement period: night midpoints when movement occurs at night, or day midpoints when movement occurs during the day. The resulting **stap** intervals therefore cover the periods between successive movements. If a **stap_long** is provided (as a `data.frame` or a CSV path), its intervals are kept as-is and the remaining gaps are filled using twilight midpoints.

Usage

```

tag_stap_daily(
  tag,
  stap_long = NULL,
  movement_period = "night",
  max_twl_gap = 23.5,
  quiet = FALSE,
  twl_grouping = lifecycle::deprecated(),
  stap0 = lifecycle::deprecated()
)

```

Arguments

tag	a <code>GeoPressureR</code> tag object with twilight .
stap_long	optional long stationary periods, supplied as a stap <code>data.frame</code> with columns start and end , a CSV path with those columns (POSIXct-compatible strings), or a tag id resolved by <code>read_stap()</code> .

<code>movement_period</code>	When the bird is assumed to be moving: "night" (default) assumes the bird is moving during the night, so sunrise-to-sunset twilights are combined into the same stationary period, while "day" assumes the bird is moving during the day, so sunset-to-sunrise twilights are combined into the same stationary period.
<code>max_twl_gap</code>	maximum allowed gap between consecutive twilights (in hours) before erroring, indicating missing twilights.
<code>quiet</code>	logical to hide messages.
<code>twl_grouping</code>	[Deprecated] Renamed to <code>movement_period</code> .
<code>stap0</code>	[Deprecated] Renamed to <code>stap_long</code> .

Value

Updated `tag` with a twilight-based `stap` and optional `stap0`. The `stap` includes a logical column `stap0` set to `TRUE` when intervals come from the provided `stap0` input.

See Also

Other tag: `print.tag()`, `read_stap()`, `tag_create()`, `tag_set_map()`

Examples

```
# Create and label the sunrise and sunset times used to define periods.
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    twilight_create() |>
    twilight_label_read()

  # Assume the bird moves at night. Preserve the long stationary periods
  # stored in data/stap-label/18LX.csv, while filling the remaining time
  # with daily periods bounded by night midpoints.
  tag <- tag_stap_daily(
    tag,
    stap_long = tag$param$id,
    movement_period = "night"
  )
})

head(tag$stap)
```

Description

Find the position of the highest value in a map, typically most probable value in a likelihood map.

Note that this path is the most likely, considering only the observation model and ignoring the movement model. Prefer to use `graph_most_likely()` for the most realistic path accounting for flight duration.

`interp` can be used to interpolate unrealistic position from short stationary periods based on the position of the longer ones. In order to preserve a reference to the grid, the interpolation is only performed at the centre of the grid cell (defined by the `likelihood` map). Note also that it is not possible to interpolate the first and last stationary period.

Usage

```
tag2path(tag, likelihood = NULL, interp = FALSE, use_known = TRUE)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object
<code>likelihood</code>	Field of the <code>tag</code> list containing the likelihood map (character). By default, uses the product of <code>map_pressure</code> and <code>map_light</code> if available, otherwise the first available likelihood map in the tag.
<code>interp</code>	the position of the stationary period shorter than <code>interp</code> will be replaced by a linear average from other position accounting for flight duration (in days).
<code>use_known</code>	If TRUE, enforce known positions from <code>tag\$stap</code> when building the path. Known positions are approximated to the map resolution so they correspond to integer indices.

Value

A path data.frame

- `stap_id` stationary period
- `ind` indices of the coordinate in the 2D grid. Useful to retrieve map or graph information.
- `lat` Latitude,
- `lon` longitude

See Also

Other path: [path2edge\(\)](#), [path2elevation\(\)](#), [path2twilight\(\)](#), [plot_path\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
  tag_label(quiet = TRUE) |>
  tag_set_map(
```

```

      extent = c(-16, 23, 0, 50),
      scale = 2
    ) |>
    geopressure_map(quiet = TRUE)
  })

# Extract a path from pressure map
path <- tag2path(tag)
plot_path(path)

# Short stationary periods (e.g. 1 day) can be unreliably
# estimated, so interpolating them is often better
path <- tag2path(tag, interp = 1)
plot_path(path)

```

trainset

Start the GeoPressure Trainset shiny app

Description

GeoPressure Trainset is a shiny app designed to help you manually label pressure and acceleration data for training machine learning models. This interactive app allows you to visualize time series data, select data points or regions, and assign behavioral labels (e.g., "flight", "discard", or custom elevation labels) to create training datasets.

Usage

```
trainset(x, launch_browser = TRUE, run_bg = TRUE, debug = FALSE)
```

Arguments

<code>x</code>	One of: • a GeoPressureR <code>tag</code> object, • a path to an interim <code>.RData</code> file containing an object called <code>tag</code>, • a path to a TRAINSET-compatible <code>.csv</code> label file, • a single-character <code>id</code>. In that case the function will first look for <code>"./data/interim/{id}.RData"</code>, and if not found will look for <code>"./data/tag-label/{id}-la"</code> and then <code>"./data/tag-label/{id}.csv"</code>.
<code>launch_browser</code>	If true (by default), the app runs in your browser, otherwise it runs on Rstudio.
<code>run_bg</code>	If true, the app runs in a background R session using the <code>callr</code> package. This allows you to continue using your R session while the app is running.
<code>debug</code>	If TRUE, prints debug messages about plot refreshes (x-range changes and point counts) to the R console. Defaults to FALSE.

Details

The app features:

- Interactive plotly visualization of pressure and acceleration time series
- Point and region selection for efficient labeling
- Support for stationary periods (STAP) navigation
- Custom label creation (elevation labels)
- Export functionality to save labeled data as CSV files

Learn more about data labeling workflows in the [GeoPressureManual](#) or explore the [GeoPressureR documentation](#).

Value

A `GeoPressureR` `tag` object (with pressure and optionally acceleration data and an `id` in `tag$param$id`). If `run_bg = TRUE`, a background process object is returned invisibly, with the `tag` attached as an attribute `attr(p, "tag")`. The labeled data can be exported directly from the app interface.

See Also

[tag_label_read\(\)](#), [tag_label_write\(\)](#), [GeoPressureManual](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})
trainset(tag, run_bg = FALSE, launch_browser = FALSE)
```

ts2mat

Format time series data frame into a matrix

Description

Format time series data frame into a matrix

Usage

```
ts2mat(
  ts,
  sensor = "time series",
  twl_offset = 0,
  value = "value",
  twl_time_tolerance = 180
)
```

Arguments

ts	data.frame of a tag , containing at least date and value .
sensor	sensor name used in messages.
twl_offset	Shift of the middle of the night compared to 00:00 UTC (in hours). If not provided, it uses the middle of all nights.
value	column name to extract
twl_time_tolerance	Maximum allowed time difference in seconds between observations and the regular grid. Observations beyond this threshold will be set to NA. Default is 180 seconds (3 minutes). Half the sampling interval is always allowed, so that drift-corrected timestamps sliding against the grid are kept.

Value

A data.frame with columns **date** and **value**.

Examples

```
mat <- ts2mat(tag$pressure)
```

twilight_create	<i>Estimate twilights from light data</i>
------------------------	---

Description

This function estimates twilight (i.e., datetime of sunrise and sunset) by searching for the first and last light of the day which is defined by the light exceeds or falls below the light threshold **twl_thr**.

Usage

```
twilight_create(
  tag,
  twl_thr = NULL,
  twl_offset = NULL,
  transform_light = TRUE,
  twl_time_tolerance = 180
)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object
<code>twl_thr</code>	Light threshold that defines twilight. By default (<code>NULL</code>), it uses the smallest value of light (i.e, first and last light of day).
<code>twl_offset</code>	Shift of the middle of the night compared to 00:00 UTC (in hours). If not provided, it uses the middle of all nights.
<code>transform_light</code>	logical to use a log transformation of light
<code>twl_time_tolerance</code>	Maximum allowed time difference in seconds between observations and the regular grid. Observations beyond this threshold will be set to NA. Default is 180 seconds (3 minutes). Half the sampling interval is always allowed, so that drift-corrected timestamps sliding against the grid are kept.

Value

a tag list containing a new data.frame `twilight` with columns:

- `twilight` (date-time of twilight)
- `rise` (logical) indicating sunrise (`TRUE`) or sunset (`FALSE`).
- `stap_id` if `stap_id` is present in `light`.

See Also

[GeoPressureManual](#) , [TwGeos::findTwilights\(\)](#)

Other geolight: [geolight_fit_location\(\)](#), [geolight_map\(\)](#), [geolight_solar\(\)](#), [plot_twl_calib\(\)](#), [twilight_label_read\(\)](#), [twilight_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |> tag_label(quiet = TRUE)
})

# Create twilight data.frame
tag <- twilight_create(tag)

str(tag$twilight)

plot(tag, type = "twilight")
```

`twilight_label_read` *Read a twilight label file*

Description

This function read an exported csv file from TRAINSET <https://trainset.raphaelnussbaumer.com/> and update the twilight data `tag$twilight`.

Usage

```
twilight_label_read(
  tag,
  file = glue::glue("./data/twilight-label/{tag$param$id}-labeled.csv")
)
```

Arguments

`tag` a GeoPressureR `tag` object
`file` csv file of the labels to be read.

Value

Same `tag` object, updated with the labels `tag$twilight$label`.

See Also

Other geolight: [geolight_fit_location\(\)](#), [geolight_map\(\)](#), [geolight_solar\(\)](#), [plot_twl_calib\(\)](#), [twilight_create\(\)](#), [twilight_label_write\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {
  tag <- tag_create("18LX", quiet = TRUE) |>
    tag_label(quiet = TRUE) |>
    twilight_create()

  tag_labelled <- twilight_label_read(tag)
})

plot(tag, type = "twilight") |> plotly::layout(title = list(text = "Before label"))

str(tag_labelled$twilight)

plot(tag_labelled, type = "twilight") |> plotly::layout(title = list(text = "After label"))
```

`twilight_label_write` *Write a twilight label file*

Description

This function writes the csv file of the labelled twilight which can be read with TRAINSET <https://trainset.raphaelnussbaumer.com/>.

Usage

```
twilight_label_write(  
  tag,  
  file = glue::glue("./data/twilight-label/{tag$param$id}.csv"),  
  quiet = FALSE  
)
```

Arguments

<code>tag</code>	a GeoPressureR <code>tag</code> object
<code>file</code>	Name of the twilight label file to be saved.
<code>quiet</code>	logical to hide messages

Value

None

See Also

Other geolight: [geolight_fit_location\(\)](#), [geolight_map\(\)](#), [geolight_solar\(\)](#), [plot_twl_calib\(\)](#), [twilight_create\(\)](#), [twilight_label_read\(\)](#)

Examples

```
withr::with_dir(system.file("extdata", package = "GeoPressureR"), {  
  tag <- tag_create("18LX", quiet = TRUE) |>  
  tag_label(quiet = TRUE) |>  
  twilight_create()  
  
  label_file <- twilight_label_write(tag)  
  
  str(utils::read.csv(label_file))  
})
```

windsupport	<i>Compute wind support and drift from a wind and ground speed vectors</i>
--------------------	--

Description

Wind support, or wind profit, is the projection of the wind vector $\vec{v}_w$ onto the groundspeed vector $\vec{v}_g$,

$$\text{Wind Support} = \frac{\vec{v}_g \cdot \vec{v}_w}{|\vec{v}_g|},$$

with $\cdot$ being the **dot product**.

A positive value indicates that wind was blowing in the direction of movement, while a negative one indicates a head wind.

windspeed and **groundspeed** should be expressed as complex value where the real part corresponds to the east-west component and imaginary part to the north-south component.

Alternatively to wind support, you can compute the drift (or crosswind component) value with **drift = TRUE**,

$$\text{Drift} = \frac{|\vec{v}_g \times \vec{v}_w|}{|\vec{v}_g|},$$

with $\times$ being the **cross product**.

A positive value of drift indicates that the wind is pushing the bird to the right of its intended path, while a negative indicates that the wind is pushing the bird to the left.

You can use **abs()** (or **Mod()**) to compute the norm (or absolute value) of the speed vector and **speed_to_bearing()** to compute the bearing/orientation of the speed vector. The latter computes the trigonometric angle of the speed vector with **Arg()** and convert this angle to a bearing (North = 0° instead of 90°).

Usage

```
windsupport(windspeed, groundspeed, drift = FALSE)
```

Arguments

windspeed	windspeed as complex value
groundspeed	groundspeed as complex value
drift	return drift instead of windsupport

Value

wind support as scalar value (same unit as **gs** and **ws**)

Examples

```
windsupport(windspeed = 5 + 2i, groundspeed = 10 + 0i)
```

Index

- * **bird**
 - bird_create, 3
 - print.bird, 70
- * **edge**
 - edge_add_wind, 4
- * **geolight**
 - geolight_fit_location, 6
 - geolight_map, 8
 - geolight_solar, 11
 - plot_twl_calib, 64
 - twilight_create, 100
 - twilight_label_read, 102
 - twilight_label_write, 103
- * **geopressure_map**
 - geopressure_map, 15
- * **geopressuretemplate**
 - geopressuretemplate, 25
- * **graph**
 - graph_add_wind, 30
 - graph_create, 32
 - graph_marginal, 34
 - graph_most_likely, 35
 - graph_set_movement, 36
 - graph_simulation, 39
 - print.graph, 71
- * **map plot_tag**
 - plot.map, 50
- * **map**
 - map_add_mask_water, 41
 - map_create, 42
 - print.map, 72
 - rast.map, 74
- * **movement**
 - bird_create, 3
 - graph_set_movement, 36
 - plot_graph_movement, 54
 - speed2prob, 77
 - tag_download_wind, 85
- * **param**
 - param_create, 45
 - print.param, 73
- * **path**
 - path2edge, 46
 - path2elevation, 47
 - path2twilight, 49
 - plot_path, 55
 - tag2path, 96
- * **plot_tag**
 - plot_tag_acceleration, 58
 - plot_tag_actogram, 59
 - plot_tag_light, 60
 - plot_tag_pressure, 61
 - plot_tag_temperature, 62
 - plot_tag_twilight, 63
- * **pressurepath**
 - geopressure_timeseries, 20
 - plot_pressurepath, 57
 - pressurepath_create, 65
 - pressurepath_variable_available, 69
- * **tag plot_tag**
 - plot.tag, 52
- * **tag_label**
 - tag_label, 87
 - tag_label_auto, 88
 - tag_label_read, 90
 - tag_label_stap, 91
 - tag_label_write, 92
- * **tag**
 - print.tag, 73
 - read_stap, 75
 - tag_create, 81
 - tag_set_map, 93
 - tag_stap_daily, 95

base::load(), 41

bird_create, 3

bird_create(), 38, 55, 70, 77, 87

- crs, 74
- ecmwfr::wf_get_key(), 23, 69, 86
- ecmwfr::wf_request_batch, 86
- ecmwfr::wf_set_key(), 21, 67, 86
- edge_add_wind, 4, 30
- edge_add_wind(), 86
- geolight_fit_location, 6
- geolight_fit_location(), 10, 12, 65, 101–103
- geolight_map, 8
- geolight_map(), 8, 12, 65, 101–103
- geolight_map_aggregate
(*geolight_map*), 8
- geolight_map_calibrate
(*geolight_map*), 8
- geolight_map_likelihood
(*geolight_map*), 8
- geolight_solar, 11
- geolight_solar(), 8, 10, 65, 101–103
- geolightviz, 13
- geopressure_map, 15
- geopressure_map(), 23, 68
- geopressure_map_likelihood
(*geopressure_map*), 15
- geopressure_map_mismatch
(*geopressure_map*), 15
- geopressure_map_mismatch(), 18, 19
- geopressure_map_preprocess, 19
- geopressure_map_preprocess(), 67
- geopressure_timeseries, 20
- geopressure_timeseries(), 23, 58, 69, 70, 86
- geopressure_timeseries_api
(*geopressure_timeseries*), 20
- geopressure_timeseries_arco
(*geopressure_timeseries*), 20
- GeoPressureR, 24
- geopressuretemplate, 25
- geopressuretemplate_config
(*geopressuretemplate*), 25
- geopressuretemplate_graph
(*geopressuretemplate*), 25
- geopressuretemplate_graph(), 26
- geopressuretemplate_pressurepath
(*geopressuretemplate*), 25
- geopressuretemplate_pressurepath(), 27
- geopressuretemplate_tag
(*geopressuretemplate*), 25
- geopressuretemplate_tag(), 26
- geopressureviz, 28
- graph_add_wind, 30
- graph_add_wind(), 33, 34, 36, 38, 39, 71
- graph_assert, 31
- graph_create, 32
- graph_create(), 31, 34, 36, 38, 39, 71
- graph_marginal, 34
- graph_marginal(), 31, 33, 36, 38, 39, 71
- graph_most_likely, 35
- graph_most_likely(), 31, 33, 34, 38, 39, 71
- graph_set_movement, 36
- graph_set_movement(), 4, 31, 33, 34, 36, 39, 55, 71, 77, 87
- graph_simulation, 39
- graph_simulation(), 31, 33, 34, 36, 38, 71
- kmeans(), 89
- load_interim, 40
- map_add_mask_water, 41
- map_add_mask_water(), 43, 72, 75
- map_create, 42
- map_create(), 42, 72, 75
- map_expand, 44
- param_create, 45
- param_create(), 73, 82
- path2edge, 46
- path2edge(), 48, 50, 57, 97
- path2elevation, 47
- path2elevation(), 47, 50, 57, 97
- path2twilight, 49
- path2twilight(), 47, 48, 57, 97
- plot.map, 50
- plot.map(), 55, 57
- plot.tag, 52
- plot.tag(), 52
- plot_graph_movement, 54
- plot_graph_movement(), 4, 38, 77, 87
- plot_path, 55
- plot_path(), 47, 48, 50, 52, 97
- plot_pressurepath, 57
- plot_pressurepath(), 23, 69, 70

plot_tag_acceleration, 58
 plot_tag_acceleration(), 60–62, 64
 plot_tag_actogram, 59
 plot_tag_actogram(), 59, 61, 62, 64
 plot_tag_light, 60
 plot_tag_light(), 59, 60, 62, 64
 plot_tag_pressure, 61
 plot_tag_pressure(), 59–62, 64
 plot_tag_temperature, 62
 plot_tag_temperature(), 59–62, 64
 plot_tag_twilight, 63
 plot_tag_twilight(), 59–62
 plot_twl_calib, 64
 plot_twl_calib(), 8, 10, 12, 101–103
 plot_twl_calib_path (*plot_twl_calib*), 64
 pressurepath_create, 65
 pressurepath_create(), 16, 23, 58, 69, 70, 86
 pressurepath_create_api (*pressurepath_create*), 65
 pressurepath_create_arco (*pressurepath_create*), 65
 pressurepath_variable_available, 69
 pressurepath_variable_available(), 23, 58, 66, 67, 69
 print.bird, 70
 print.bird(), 4
 print.graph, 71
 print.graph(), 31, 33, 34, 36, 38, 39
 print.map, 72
 print.map(), 42, 43, 75
 print.param, 73
 print.param(), 45
 print.tag, 73
 print.tag(), 75, 84, 94, 96

 rast.map, 74
 rast.map(), 42, 43, 72
 read_stap, 75
 read_stap(), 74, 84, 94, 96

 speed2bearing, 76
 speed2prob, 77
 speed2prob(), 4, 38, 55, 87
 stap2duration, 78
 stap2flight, 79
 stats::density(), 9

 tag2path, 96
 tag2path(), 47, 48, 50, 57
 tag_assert, 80
 tag_create, 81
 tag_create(), 74, 75, 94, 96
 tag_download_wind, 85
 tag_download_wind(), 4, 5, 23, 38, 55, 69, 77, 86
 tag_label, 87
 tag_label(), 89–92
 tag_label_auto, 88
 tag_label_auto(), 88, 90–92
 tag_label_read, 90
 tag_label_read(), 88, 89, 91, 92, 99
 tag_label_stap, 91
 tag_label_stap(), 79, 88–90, 92
 tag_label_write, 92
 tag_label_write(), 88–91, 99
 tag_set_map, 93
 tag_set_map(), 13, 74, 75, 84, 96
 tag_stap_daily, 95
 tag_stap_daily(), 74, 75, 84, 94
 terra::names, 74
 terra::project(), 50
 terra::SpatRaster, 74
 trainset, 98
 ts2mat, 99
 twilight_create, 100
 twilight_create(), 8, 10, 12, 65, 102, 103
 twilight_label_read, 102
 twilight_label_read(), 8, 10, 12, 65, 101, 103
 twilight_label_write, 103
 twilight_label_write(), 8, 10, 12, 65, 101, 102

 wf_set_key, 86
 windsupport, 104