Package: irishbuoys (via r-universe)

June 7, 2026

Title Analyze Irish Weather Buoy Network Data

Version 0.2.0

Description Provides tools to download, process, and analyze data from the Irish Weather Buoy Network. Includes functions for accessing real-time and historical data via the Marine Institute's ERDDAP server, storing data in DuckDB for efficient querying, and building predictive models for wave height and weather conditions.

URL https://johngavin.github.io/irishbuoys/

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Imports arrow, cli, dbplyr, DBI, dplyr, duckdb, glue, httr2, jsonlite, lubridate, pointblank, rlang, tibble, utils

Suggests blastula, copula, kendallknight, covr, crew, dygraphs, DT, evd, extRemes, forecast, gert, ggplot2, here, htmltools, knitr, mev, mgcv, mirai, nanoparquet, perspectiveR, plumber, plotly, quantreg, quarto, ranger, rmarkdown, scales, shiny, shinylive, SpatialExtremes, tarchetypes, targets, testthat (>= 3.0.0), tidyr, withr, xts, zoo

VignetteBuilder knitr, quarto

Config/testthat/edition 3

Config/pak/sysreqs cmake make libicu-dev libuv1-dev libxml2-dev libssl-dev libnode-dev xz-utils

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

Date/Publication 2026-06-07 10:15:18 UTC

RemoteUrl https://github.com/JohnGavin/irishbuoys

RemoteRef HEAD

RemoteSha eb34bb9405ec6633cd6f1af29c863f023a980735

Contents

`add_wave_metrics` *Add Wave Metrics to Data*

Description

Adds calculated wave metrics including rogue wave flag and steepness.

Usage

```
add_wave_metrics(data, rogue_threshold = 2)
```

Arguments

`data`	Data frame with wave_height, hmax, and wave_period columns
`rogue_threshold`	Threshold for rogue wave classification (default: 2.0)

Value

Data frame with additional columns: rogue_ratio, is_rogue, steepness, danger_level

Examples

```
data <- data.frame(
  wave_height = c(2.5, 3.0, 1.8),
  hmax = c(4.5, 6.5, 3.2),
  wave_period = c(8, 10, 7)
)
add_wave_metrics(data)
```

`analyze_gust_factor` *Analyze Gust Factor*

Description

Analyzes the ratio of peak gust to sustained wind speed. This is the wind equivalent of the wave Hmax/Hs ratio.

Usage

```
analyze_gust_factor(data, min_wind_speed = 5)
```

Arguments

`data`	Data frame with gust and wind_speed columns
`min_wind_speed`	Minimum sustained wind speed to consider (default: 5 m/s)

Value

List with:

- summary: summary statistics of gust factor
- extreme_gusts: observations with high gust factors
- by_wind_category: gust factor by wind speed category

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "wind_speed", "gust"))
gust_analysis <- analyze_gust_factor(data)
DBI::dbDisconnect(con)

## End(Not run)
```

`analyze_joint_extremes` *Analyze Joint Extremes Between Stations*

Description

Analyzes how often extreme events co-occur at multiple stations.

Usage

```
analyze_joint_extremes(
  data,
  variable = "wave_height",
  threshold_quantile = 0.95
)
```

Arguments

`data`	Data frame with columns: time, station_id, and the variable
`variable`	Variable to analyze (default: "wave_height")
`threshold_quantile`	Quantile threshold for "extreme" (default: 0.95)

Value

List with:

- joint_extreme_counts: matrix of joint extreme event counts
- conditional_probs: P(station j extreme | station i extreme)
- extreme_events: data frame of all extreme events

Examples

```
## Not run:
data <- data.frame(
  time = rep(seq(as.POSIXct("2024-01-01"), by = "hour", length.out = 100), 2),
  station_id = rep(c("M2", "M3"), each = 100),
  wave_height = c(rnorm(100, 3, 1), rnorm(100, 2.5, 0.8))
)
analyze_joint_extremes(data)

## End(Not run)
```

`analyze_parquet_storage` *Analyze Parquet Storage*

Description

Get statistics about Parquet file storage.

Usage

```
analyze_parquet_storage(data_path = "inst/extdata/parquet")
```

Arguments

`data_path`	Base path for Parquet files

`analyze_rogue_statistics` *Analyze Rogue Wave Statistics*

Description

Computes statistics on rogue wave occurrence rates and associated conditions.

Usage

```
analyze_rogue_statistics(con, threshold = 2, min_wave_height = 2)
```

Arguments

`con`	DBI connection to DuckDB database
`threshold`	Hmax/WaveHeight ratio threshold (default: 2.0)
`min_wave_height`	Minimum significant wave height (default: 2m)

Value

List containing rogue wave statistics by station and overall

Examples

```
## Not run:
con <- connect_duckdb()
stats <- analyze_rogue_statistics(con)
print(stats$by_station)
DBI::dbDisconnect(con)

## End(Not run)
```

---

analyze_station_pairs *Analyze All Station Pairs*

Description

Computes cross-correlations for all unique station pairs.

Usage

```
analyze_station_pairs(data, variable = "wave_height", max_lag = 48)
```

Arguments

data	Data frame with columns: time, station_id, and the variable
variable	Variable to analyze (default: "wave_height")
max_lag	Maximum lag in hours (default: 48)

Value

Data frame with one row per station pair containing:

- station1, station2: station pair
- distance_km: distance between stations
- optimal_lag: lag with max correlation
- max_correlation: correlation at optimal lag
- expected_lag: expected lag based on wave propagation (~30 km/h)

Examples

```
## Not run:
data <- data.frame(
  time = rep(seq(as.POSIXct("2024-01-01"), by = "hour", length.out = 100), 2),
  station_id = rep(c("M2", "M3"), each = 100),
  wave_height = c(rnorm(100, 3, 1), rnorm(100, 2.5, 0.8))
)
analyze_station_pairs(data)

## End(Not run)
```

`api_plumber` *Plumber API for irishbuoys*

Description

Functions for running a live REST API that serves pre-built static JSON data from the targets pipeline.

See Also

Other api: `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_spatial()`, `generate_api_status()`, `generate_api_trends()`, `run_api()`

`api_static` *Static API Generation Functions*

Description

Functions for generating static JSON API files served via GitHub Pages. These are written to `docs/api/v1/` and updated 6-hourly by CI.

See Also

Other api: `api_plumber`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_spatial()`, `generate_api_status()`, `generate_api_trends()`, `run_api()`

beaufort_to_description *Convert Beaufort Number to Description*

Description

Maps Beaufort scale integers (0-12) to standard descriptions.

Usage

```
beaufort_to_description(beaufort)
```

Arguments

beaufort	Integer vector of Beaufort numbers (0-12).

Value

Character vector of Beaufort descriptions.

See Also

Other storm-alert: create_storm_alert_email(), detect_storm_events(), fetch_all_forecasts(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), send_storm_alert(), summarise_forecast_rogue_risk()

Examples

```
beaufort_to_description(0:12)
```

buoy_tbl *Get Lazy Reference to Buoy Data Table*

Description

Returns a lazy dplyr tibble reference to the buoy_data table. All dplyr operations are translated to SQL and executed in DuckDB. Call collect() to retrieve results as a data frame.

Usage

```
buoy_tbl(con, table_name = "buoy_data")
```

Arguments

con	DBI connection to DuckDB database
table_name	Name of the table (default: "buoy_data")

Value

A lazy tibble (tbl_dbi) for use with dplyr verbs

Examples

```
## Not run:
con <- connect_duckdb()
buoy_tbl(con) |>
  dplyr::filter(station_id == "M3", wave_height > 5) |>
  dplyr::select(time, wave_height, hmax) |>
  dplyr::collect()
DBI::dbDisconnect(con)

## End(Not run)
```

---

calculate_annual_trends

Calculate Annual Trends

Description

Calculates annual statistics and fits a linear trend to detect long-term changes in the data.

Usage

```
calculate_annual_trends(data, variable = "wave_height", time_col = "time")
```

Arguments

data	Data frame with time and value columns
variable	Name of the variable (default: "wave_height")
time_col	Name of the time column (default: "time")

Value

List with:

- annual_stats: annual mean, max, etc.
- trend_model: linear model for trend
- trend_per_decade: change per decade with significance

Examples

```
set.seed(1)
data <- data.frame(
  time = seq(as.POSIXct("2020-01-01"), by = "hour", length.out = 1000),
  wave_height = 2 + sin(seq(0, 20, length.out = 1000)) + rnorm(1000, 0, 0.3)
)
result <- calculate_annual_trends(data)
result$trend_per_decade
result$p_value
```

`calculate_gpd_return_levels`

Calculate GPD Return Levels from Per-Station Fits

Description

Computes return levels from a GPD (Generalized Pareto Distribution) fit produced by `mev::fit.gpd()`. Uses the standard GPD return level formula with delta method confidence intervals.

The formula is:

$$z_T = u + \frac{\sigma}{\xi}\left[(\lambda T)^{\xi} - 1\right]$$

where u is the threshold, σ is the scale, ξ is the shape, λ is the exceedance rate, and T is the return period in years.

When shape is approximately zero, the exponential fallback is used:

$$z_T = u + \sigma \log(\lambda T)$$

Usage

```
calculate_gpd_return_levels(
  gpd_fit,
  return_periods = c(1, 5, 10),
  n_obs_per_year = 8760,
  n_total = NULL,
  exceedance_rate = NULL,
  conf_level = 0.95
)
```

Arguments

`gpd_fit`	A list from the per-station GPD fitting targets, with elements: `u` (threshold), `scale`, `shape`, `n_exceed`, and optionally `se_scale`, `se_shape`. If it contains an `error` field, NA rows are returned.
`return_periods`	Numeric vector of return periods in years (default: c(1, 5, 10))
`n_obs_per_year`	Number of observations per year for exceedance rate calculation (default: 8760 for hourly data)

n_total	Total number of observations. If NULL, estimated from n_exceed and threshold percentile.
exceedance_rate	Pre-computed exceedance rate (lambda). If NULL, estimated as n_exceed / n_total.
conf_level	Confidence level for intervals (default: 0.95)

Value

Data frame with columns: return_period, return_level, lower, upper, threshold_value, method. Returns NA return levels if the fit has an error or missing parameters.

Examples

```
fit <- list(u = 5.0, scale = 1.2, shape = 0.1, n_exceed = 500)
calculate_gpd_return_levels(fit, return_periods = c(10, 50, 100))
```

---

calculate_hs_from_elevation

Calculate Significant Wave Height from Raw Elevations

Description

Calculates Hs from a time series of surface elevation measurements using the spectral method (4 * sigma).

Usage

```
calculate_hs_from_elevation(elevations)
```

Arguments

elevations	Numeric vector of surface elevation measurements (m)

Value

Significant wave height in meters

Examples

```
# Simulated wave elevation time series
t <- seq(0, 1000, by = 0.5)  # 1000 seconds at 2Hz
elevation <- 0.5 * sin(2*pi*t/8) + 0.3 * sin(2*pi*t/12) + rnorm(length(t), 0, 0.1)
hs <- calculate_hs_from_elevation(elevation)
```

`calculate_return_levels` *Calculate Return Levels*

Description

Calculates return levels for specified return periods from a fitted extreme value model.

Usage

```
calculate_return_levels(
  fit,
  return_periods = c(10, 50, 100),
  conf_level = 0.95
)
```

Arguments

`fit`	Result from fit_gev_annual_maxima or fit_gpd_threshold
`return_periods`	Numeric vector of return periods in years (default: c(10, 50, 100))
`conf_level`	Confidence level for intervals (default: 0.95)

Value

Data frame with:

- return_period: return period in years
- return_level: estimated return level
- lower: lower confidence bound
- upper: upper confidence bound

Examples

```
## Not run:
gev_result <- fit_gev_annual_maxima(data)
levels <- calculate_return_levels(gev_result, c(10, 50, 100))
print(levels)

## End(Not run)
```

calculate_rms_wave_height
Calculate RMS Wave Height

Description

Calculates the Root Mean Square wave height, which is related to wave energy content.

Usage

```
calculate_rms_wave_height(wave_heights)
```

Arguments

wave_heights	Numeric vector of individual wave heights (m)

Details

H_rms = sqrt(mean(H^2))

Relationship to Hs (for Rayleigh distribution): H_rms = Hs / sqrt(8) ~ 0.707 * Hs

Value

RMS wave height in meters

Examples

```
heights <- c(1.2, 2.1, 0.8, 3.5, 1.9, 2.8)
h_rms <- calculate_rms_wave_height(heights)
```

calculate_seasonal_means
Calculate Seasonal Means

Description

Calculates mean values by month and season for a variable.

Usage

```
calculate_seasonal_means(data, variable = "wave_height", time_col = "time")
```

Arguments

data	Data frame with time and value columns
variable	Name of the variable (default: "wave_height")
time_col	Name of the time column (default: "time")

Value

List with:

- monthly: mean values by month
- seasonal: mean values by season (DJF, MAM, JJA, SON)

Examples

```
set.seed(1)
data <- data.frame(
  time = seq(as.POSIXct("2020-01-01"), by = "hour", length.out = 1000),
  wave_height = 2 + sin(seq(0, 20, length.out = 1000)) + rnorm(1000, 0, 0.3)
)
result <- calculate_seasonal_means(data)
result$monthly
result$seasonal
```

`calculate_wave_steepness`

Calculate Wave Steepness

Description

Calculates wave steepness, an important safety metric. Steepness > 0.07 indicates breaking waves (dangerous).

Usage

```
calculate_wave_steepness(wave_height, wave_period)
```

Arguments

`wave_height`	Significant wave height in meters
`wave_period`	Wave period in seconds

Details

Wave steepness = H / L where L = g * T^2 / (2 * pi) Simplified: steepness = H / (1.56 * T^2)

Value

Wave steepness (dimensionless)

Examples

```
# 3m wave with 8 second period
steepness <- calculate_wave_steepness(3, 8)
# steepness = 0.03 (safe)

# 3m wave with 4 second period
steepness <- calculate_wave_steepness(3, 4)
# steepness = 0.12 (dangerous - breaking waves)
```

---

ci_bootstrap_return_levels

Bootstrap Confidence Intervals for GPD Return Levels

Description

Non-parametric bootstrap (optionally block bootstrap) confidence intervals for GPD-based return levels. Resamples the raw data, refits the GPD, and computes return levels for each replicate.

Usage

```
ci_bootstrap_return_levels(
  data,
  variable,
  return_periods = c(1, 5, 10),
  n_boot = 500,
  conf_level = 0.95,
  block_size = NULL,
  threshold_quantile = 0.95,
  n_obs_per_year = 8760,
  seed = 42
)
```

Arguments

`data`	Data frame containing the variable to analyse.
`variable`	Character name of the column (e.g. `"wave_height"`).
`return_periods`	Numeric vector of return periods in years (default `c(1, 5, 10)`).
`n_boot`	Number of bootstrap replicates (default 500).
`conf_level`	Confidence level (default 0.95).
`block_size`	Integer block size for block bootstrap (observations, not hours). `NULL` or 0 for iid bootstrap. Use 48 for hourly data (2-day blocks) to preserve temporal dependence.
`threshold_quantile`	Quantile for the POT threshold (default 0.95).
`n_obs_per_year`	Observations per year for return level calculation (default 8760 for hourly).
`seed`	Random seed for reproducibility (default 42).

Details

For each bootstrap replicate:

1. Resample observations (iid or block bootstrap)
2. Compute the threshold as the `threshold_quantile` of the resample
3. Fit GPD via `mev::fit.gpd()` to exceedances
4. Compute return levels via `calculate_gpd_return_levels()`

The percentile method is used: CIs are the `alpha/2` and `1-alpha/2` quantiles of the bootstrap distribution of return levels.

Value

A data.frame with columns: `return_period`, `return_level`, `lower`, `upper`, `n_success`, `method`.

`ci_order_statistics` *Order-Statistics Confidence Intervals for Quantiles*

Description

Distribution-free confidence intervals for population quantiles using order statistics. Uses the Beta distribution to find order-statistic indices j,k such that `(X_(j), X_(k))` covers the p-th quantile with at least the specified confidence level.

Usage

```
ci_order_statistics(x, probs, conf_level = 0.95)
```

Arguments

`x`	Numeric vector of observations (NAs removed internally).
`probs`	Numeric vector of probabilities for which to compute CIs (e.g. `c(0.95, 0.99)`).
`conf_level`	Confidence level (default 0.95).

Details

For a sample of size n, the probability that the interval `(X_(j), X_(k))` contains the p-th quantile is `pbeta(p, j, n-j+1) - pbeta(p, k, n-k+1)`. We search for the tightest such interval achieving at least `conf_level` coverage.

This method is distribution-free: it requires no parametric assumptions. With ~8 years of hourly data (~70k observations), order-statistic CIs are well-defined even for extreme quantiles like the 99th percentile.

Value

A data.frame with columns: `probability`, `quantile`, `lower`, `upper`, `j`, `k`, `actual_coverage`, `method`.

Examples

```
set.seed(42)
x <- rnorm(1000)
ci_order_statistics(x, probs = c(0.95, 0.99))
```

`ci_parametric_bootstrap` *Parametric Bootstrap CIs for GPD Return Levels*

Description

Simulate from a fitted GPD, refit, and compute return levels to obtain parametric bootstrap confidence intervals.

Usage

```
ci_parametric_bootstrap(
  gpd_fit,
  n_boot = 500,
  return_periods = c(1, 5, 10),
  conf_level = 0.95,
  n_obs_per_year = 8760,
  seed = 42
)
```

Arguments

`gpd_fit`	A list with elements `u`, `scale`, `shape`, `n_exceed` (as returned by the per-station GPD targets).
`n_boot`	Number of bootstrap replicates (default 500).
`return_periods`	Numeric vector of return periods in years.
`conf_level`	Confidence level (default 0.95).
`n_obs_per_year`	Observations per year (default 8760).
`seed`	Random seed (default 42).

Details

For each replicate:

1. Simulate `n_exceed` exceedances from GPD(scale, shape)
2. Add threshold to obtain values above `u`
3. Refit GPD via `mev::fit.gpd()`
4. Compute return levels

Uses the percentile method for CIs.

Value

A data.frame with columns: `return_period`, `return_level`, `lower`, `upper`, `n_success`, `method`.

`compare_rogue_wave_gust` *Compare Rogue Wave and Rogue Gust Occurrence*

Description

Compares the occurrence rates of rogue waves (Hmax/Hs > 2) and extreme gusts (gust/wind > 2.6).

Usage

```
compare_rogue_wave_gust(data)
```

Arguments

`data`	Data frame with wave_height, hmax, wind_speed, gust columns

Value

Data frame comparing occurrence rates

Examples

```
data <- data.frame(
  wave_height = c(3, 4, 5, 2.5),
  hmax = c(5, 9, 8, 4),
  wind_speed = c(10, 15, 20, 8),
  gust = c(15, 40, 30, 12)
)
compare_rogue_wave_gust(data)
```

`compute_acf_summary` *Compute ACF Summary*

Description

Computes autocorrelation function values and returns them as a tibble. Useful for examining temporal dependence structure in buoy data.

Usage

```
compute_acf_summary(data, variable = "wave_height", max_lag = 48)
```

Arguments

data	Data frame with the variable to analyze
variable	Name of the variable (default: "wave_height")
max_lag	Maximum number of lags (default: 48)

Value

A tibble with columns lag and acf.

Examples

```
set.seed(1)
data <- data.frame(
  time = seq(as.POSIXct("2020-01-01"), by = "hour", length.out = 1000),
  wave_height = 2 + sin(seq(0, 20, length.out = 1000)) + rnorm(1000, 0, 0.3)
)
acf_result <- compute_acf_summary(data)
head(acf_result)
```

---

compute_calibration *Compute calibration metrics for predictions*

Description

Compute calibration metrics for predictions

Usage

```
compute_calibration(predictions)
```

Arguments

predictions	Tibble from read_predictions()

Value

List with: brier_score, accuracy, calibration_by_bucket, rolling_brier, n_total, n_resolved

Examples

```
preds <- tibble::tibble(
  prediction_id = paste0("pred_", 1:5),
  p_success = c(0.9, 0.7, 0.3, 0.8, 0.5),
  outcome = c(TRUE, TRUE, FALSE, TRUE, FALSE),
  outcome_binary = c(1, 1, 0, 1, 0),
  recorded_at = as.character(Sys.time() - (5:1) * 86400)
)
cal <- compute_calibration(preds)
cal$brier_score
```

compute_data_coverage *Compute Data Coverage and Gaps*

Description

Computes temporal coverage and gap analysis for buoy stations. Uses dplyr only (no raw SQL).

Usage

```
compute_data_coverage(con, start_date, end_date)
```

Arguments

con	DBI connection to DuckDB database
start_date	Start date for analysis
end_date	End date for analysis

Value

List with coverage tibble and gaps tibble

compute_extremal_dependence *Compute Pairwise Extremal Dependence Across Stations*

Description

Estimates the upper tail dependence coefficient (lambda_U) for all unique station pairs using a Gumbel copula. For Gumbel copula with parameter alpha, lambda_U = 2 - 2^(1/alpha). Bootstrap confidence intervals assess whether lambda_U is significantly greater than zero (H1: spatial coherence of extremes).

Also computes empirical chi statistics at multiple quantile levels and Kendall's tau for overall rank dependence.

Usage

```
compute_extremal_dependence(
  data,
  variable = "wave_height",
  threshold_quantile = seq(0.9, 0.99, by = 0.01),
  n_bootstrap = 100,
  boot_subsample = 5000,
  station_info = NULL
)
```

Arguments

`data`	Data frame with columns: `time` (POSIXct), `station_id` (character), and the variable specified by `variable`.
`variable`	Variable to analyze (default: `"wave_height"`).
`threshold_quantile`	Quantile levels at which to compute empirical chi (default: `seq(0.9, 0.99, by = 0.01)`).
`n_bootstrap`	Number of bootstrap replicates for lambda CI (default: 100).
`boot_subsample`	Maximum observations per bootstrap replicate. Subsampling speeds computation for large datasets (default: 5000).
`station_info`	Optional data frame with station metadata (from `get_station_info()`). If `NULL`, uses the default 5-station network.

Value

List with:

dependence_table Data frame with columns: `station1`, `station2`, `distance_km`, `kendall_tau`, `lambda_upper`, `lambda_lower`, `lambda_upper_ci_low`, `lambda_upper_ci_high`, `n_concurrent`, `copula_alpha`, `chi_q95`, `chi_q99`, `h1_significant` (logical).

method Character: `"gumbel_copula"`.

n_bootstrap Integer: number of bootstrap replicates used.

threshold_quantile Numeric vector of quantile levels for chi.

If the copula package is unavailable or no valid pairs exist, returns a list with an `error` field.

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "station_id", "wave_height"))
result <- compute_extremal_dependence(data)
result$dependence_table
DBI::dbDisconnect(con)

## End(Not run)
```

---

compute_obs_confidence

Confidence Multiplier from Observation Age

Description

Maps observation age in hours to a confidence multiplier in `(0, 1]`. Confidence is 1.0 while data is fresh, then decays linearly between breakpoints. Floor of 0.1 — we never claim zero information.

Default schedule (chosen to match the buoy fetch cadence):

- 0 - 6 h : 1.00 (well within the 6-h fetch cycle)
- 6 - 24 h : 1.00 -> 0.50 (one missed fetch up to one day)
- 24 - 72 h : 0.50 -> 0.25 (one to three missed days)
- > 72 h : 0.10 (floor)

Usage

```
compute_obs_confidence(age_hours)
```

Arguments

`age_hours`	Numeric vector of ages in hours. NA in -> NA out.

Value

Numeric vector in `[0.1, 1]`, same length as `age_hours`.

See Also

Other obs-confidence: `obs_status_label()`, `widen_ci()`

Examples

```
compute_obs_confidence(c(0, 6, 12, 24, 48, 72, 168))
```

---

connect_duckdb *Create or Connect to Irish Buoys DuckDB Database*

Description

Creates a new DuckDB database or connects to an existing one for storing Irish Weather Buoy Network data. Sets up the schema if creating new.

Usage

```
connect_duckdb(db_path = "inst/extdata/irish_buoys.duckdb", create_new = FALSE)
```

Arguments

`db_path`	Character, path to database file (default: "inst/extdata/irish_buoys.duckdb")
`create_new`	Logical, whether to create new database (default: FALSE)

Value

DBI connection object to the DuckDB database

Examples

```
# Connect to existing database
con <- connect_duckdb()

# Create new database
con <- connect_duckdb(create_new = TRUE)

# Don't forget to disconnect when done
DBI::dbDisconnect(con)
```

---

convert_duckdb_to_parquet *Convert Existing DuckDB to Parquet*

Description

One-time conversion from DuckDB database to Parquet files.

Usage

```
convert_duckdb_to_parquet(
  db_path = "inst/extdata/irish_buoys.duckdb",
  data_path = "inst/extdata/parquet"
)
```

Arguments

db_path	Path to existing DuckDB database
data_path	Output path for Parquet files

---

create_api_router *Create the irishbuoys Plumber Router*

Description

Creates and returns a plumber router without starting the server. Useful for testing and programmatic access.

Usage

```
create_api_router()
```

Value

A plumber router object.

See Also

Other api: api_plumber, api_static, generate_api_decomposition(), generate_api_extremes(), generate_api_gust_factors(), generate_api_index(), generate_api_latest(), generate_api_methods(), generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

Examples

```
## Not run:
pr <- create_api_router()
# Test endpoints programmatically

## End(Not run)
```

---

create_buoy_schema *Create Database Schema for Buoy Data*

Description

Creates the necessary tables and indexes for efficient storage and querying of buoy data.

Usage

```
create_buoy_schema(con)
```

Arguments

con	DBI connection object

Value

Invisible NULL

create_email_summary *Create HTML Email Summary*

Description

Formats the weekly summary as an HTML email using blastula.

Usage

```
create_email_summary(summary)
```

Arguments

summary	Summary object from generate_weekly_summary()

Value

blastula email object

create_plot_annual_trends *Create Annual Trends Line Plot*

Description

Create Annual Trends Line Plot

Usage

```
create_plot_annual_trends(annual_trends, date_caption = NULL)
```

Arguments

annual_trends	Annual trends from calculate_annual_trends
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
trends <- list(
  annual_stats = data.frame(
    year = 2018:2024,
    mean = runif(7, 1.5, 3.5),
    sd = runif(7, 0.3, 1.0)
  )
)
create_plot_annual_trends(trends)

## End(Not run)
```

`create_plot_gust_by_category` *Create Gust Factor by Category Plot*

Description

Create Gust Factor by Category Plot

Usage

```
create_plot_gust_by_category(gust_analysis, date_caption = NULL)
```

Arguments

`gust_analysis`	Gust factor analysis results
`date_caption`	Date range caption

Value

plotly object

Examples

```
## Not run:
gust <- list(
  by_station_category = data.frame(
    station_id = rep(c("M2", "M3"), each = 3),
    wind_category = rep(c("0-10", "10-20", "20+"), 2),
    mean_gf = runif(6, 1.1, 1.8),
    p95_gf = runif(6, 1.5, 2.5),
    n = sample(50:500, 6)
  )
)
create_plot_gust_by_category(gust)

## End(Not run)
```

create_plot_gusts_vs_waves *Create Rogue Gusts vs Rogue Waves Scatter Plot*

Description

Create Rogue Gusts vs Rogue Waves Scatter Plot

Usage

```
create_plot_gusts_vs_waves(analysis_data)
```

Arguments

analysis_data	Full analysis data with both ratios computed

Value

plotly object

Examples

```
## Not run:
data <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:20) * 3600,
  station_id = rep(c("M2", "M3"), 10),
  gust = runif(20, 15, 50),
  wind_speed = runif(20, 8, 25),
  hmax = runif(20, 5, 15),
  wave_height = runif(20, 2, 6)
)
create_plot_gusts_vs_waves(data)

## End(Not run)
```

create_plot_monthly_wave *Create Monthly Wave Height Bar Plot*

Description

Create Monthly Wave Height Bar Plot

Usage

```
create_plot_monthly_wave(seasonal_means_wave, date_caption = NULL)
```

Arguments

seasonal_means_wave	Seasonal means from calculate_seasonal_means
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
seasonal <- list(
  monthly = data.frame(
    month_name = month.abb,
    mean = runif(12, 1, 4),
    sd = runif(12, 0.3, 1.0)
  )
)
create_plot_monthly_wave(seasonal)

## End(Not run)
```

---

create_plot_monthly_wind

Create Monthly Wind Speed Bar Plot

Description

Create Monthly Wind Speed Bar Plot

Usage

```
create_plot_monthly_wind(seasonal_means_wind, date_caption = NULL)
```

Arguments

seasonal_means_wind	Seasonal means from calculate_seasonal_means
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
seasonal <- list(
  monthly = data.frame(
    month_name = month.abb,
    mean = runif(12, 5, 15),
    sd = runif(12, 1, 4)
  )
)
create_plot_monthly_wind(seasonal)

## End(Not run)
```

`create_plot_return_levels`

Create Return Levels Plot

Description

Create Return Levels Plot

Usage

```
create_plot_return_levels(
  return_levels,
  variable = "wave",
  date_caption = NULL
)
```

Arguments

`return_levels`	Return levels data frame
`variable`	Variable name for title ("wave", "wind", or "hmax")
`date_caption`	Date range caption

Value

plotly object

Examples

```
## Not run:
rl <- data.frame(
  return_period = c(1, 5, 10, 25, 50, 100),
  return_level = c(5.2, 7.1, 8.3, 9.8, 11.0, 12.5),
  lower = c(4.8, 6.3, 7.2, 8.1, 8.9, 9.6),
  upper = c(5.6, 7.9, 9.4, 11.5, 13.1, 15.4)
)
```

```
create_plot_return_levels(rl, variable = "wave")

## End(Not run)
```

`create_plot_return_levels_per_station` *Create Per-Station Return Levels Plot*

Description

Creates a horizontal dotplot showing GPD return levels by station for a given variable, with error bars for confidence intervals. Text labels at each point replace the legend for clarity.

Usage

```
create_plot_return_levels_per_station(return_levels_df, variable_filter)
```

Arguments

`return_levels_df`
: Data frame from `return_levels_per_station` target with columns: station, variable, return_period, return_level, lower, upper

`variable_filter`
: Character, which variable to plot (one of "avg_wave", "rogue_wave", "avg_wind", "wind_gust")

Value

plotly object, or NULL if no data for the requested variable

Examples

```
## Not run:
rl_df <- data.frame(
  station = rep(c("M2", "M3", "M4"), each = 3),
  variable = "avg_wave",
  variable_label = "Avg Wave Height (m)",
  return_period = rep(c(1, 5, 10), 3),
  return_level = runif(9, 4, 12),
  lower = runif(9, 3, 8),
  upper = runif(9, 9, 16)
)
create_plot_return_levels_per_station(rl_df, "avg_wave")

## End(Not run)
```

create_plot_rogue_all *Create Rogue Wave All Stations Plot*

Description

Create Rogue Wave All Stations Plot

Usage

```
create_plot_rogue_all(rogue_events)
```

Arguments

rogue_events	Data frame of rogue wave events

Value

plotly object

Examples

```
## Not run:
rogue <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:10) * 3600,
  station_id = rep(c("M2", "M3"), 5),
  rogue_ratio = runif(10, 2.0, 2.5),
  hmax = runif(10, 8, 15), wave_height = runif(10, 3, 6),
  wind_speed = runif(10, 10, 30), gust = runif(10, 15, 45)
)
create_plot_rogue_all(rogue)

## End(Not run)
```

create_plot_rogue_by_station *Create Rogue Wave By Station Subplot*

Description

Create Rogue Wave By Station Subplot

Usage

```
create_plot_rogue_by_station(rogue_events, date_caption = NULL)
```

Arguments

rogue_events	Data frame of rogue wave events
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
rogue <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:10) * 3600,
  station_id = rep(c("M2", "M3"), 5),
  rogue_ratio = runif(10, 2.0, 2.5),
  hmax = runif(10, 8, 15), wave_height = runif(10, 3, 6),
  wind_speed = runif(10, 10, 30)
)
create_plot_rogue_by_station(rogue)

## End(Not run)
```

---

create_plot_rogue_gusts

Create Rogue Gusts by Station Plot

Description

Create Rogue Gusts by Station Plot

Usage

```
create_plot_rogue_gusts(gust_analysis)
```

Arguments

gust_analysis	Gust factor analysis results

Value

plotly object

Examples

```
## Not run:
gust <- list(
  rogue_gust_threshold = 1.5,
  by_station = data.frame(
    station_id = c("M2", "M3", "M4"),
    n = c(1000, 800, 600),
    n_rogue = c(15, 12, 8),
    pct_rogue = c(1.5, 1.5, 1.3),
    mean_gf = c(1.25, 1.30, 1.22),
    max_gf = c(2.8, 3.1, 2.5)
  )
)
create_plot_rogue_gusts(gust)

## End(Not run)
```

---

create_plot_rogue_gusts_all

Create Rogue Gusts All Stations Plot

Description

Create Rogue Gusts All Stations Plot

Usage

```
create_plot_rogue_gusts_all(rogue_gust_events)
```

Arguments

rogue_gust_events	Data frame of rogue gust events

Value

plotly object

Examples

```
## Not run:
gusts <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:10) * 3600,
  station_id = rep(c("M2", "M3"), 5),
  gust_ratio = runif(10, 1.5, 3.0),
  gust = runif(10, 20, 50),
  wind_speed = runif(10, 10, 25),
  wave_height = runif(10, 2, 6)
)
```

```
create_plot_rogue_gusts_all(gusts)

## End(Not run)
```

`create_plot_rogue_gusts_by_station` *Create Rogue Gusts By Station Subplot*

Description

Create Rogue Gusts By Station Subplot

Usage

```
create_plot_rogue_gusts_by_station(rogue_gust_events, date_caption = NULL)
```

Arguments

`rogue_gust_events`	Data frame of rogue gust events
`date_caption`	Date range caption

Value

plotly object

Examples

```
## Not run:
gusts <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:10) * 3600,
  station_id = rep(c("M2", "M3"), 5),
  gust_ratio = runif(10, 1.5, 3.0),
  gust = runif(10, 20, 50),
  wind_speed = runif(10, 10, 25)
)
create_plot_rogue_gusts_by_station(gusts)

## End(Not run)
```

create_plot_stl *Create STL Decomposition Plot*

Description

Create STL Decomposition Plot

Usage

```
create_plot_stl(wave_stl, date_caption = NULL)
```

Arguments

wave_stl	STL decomposition from calculate_wave_seasonality
date_caption	Date range caption

Value

ggplot2 object

Examples

```
## Not run:
stl_data <- list(
  components = data.frame(
    time = as.POSIXct("2024-01-01") + (1:100) * 86400,
    original = sin(1:100 / 10) + rnorm(100, 0, 0.2) + 2,
    seasonal = sin(1:100 / 10),
    trend = seq(1.8, 2.2, length.out = 100),
    remainder = rnorm(100, 0, 0.2)
  )
)
create_plot_stl(stl_data)

## End(Not run)
```

create_plot_time_of_day *Create Time of Day Bar Plot*

Description

Create Time of Day Bar Plot

Usage

```
create_plot_time_of_day(rogue_conditions, date_caption = NULL)
```

Arguments

rogue_conditions	Data frame with rogue wave conditions
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
conditions <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:20) * 3600,
  time_of_day = rep(c("Morning", "Afternoon", "Evening", "Night"), 5),
  rogue_ratio = runif(20, 2.0, 2.5)
)
create_plot_time_of_day(conditions)

## End(Not run)
```

---

create_plot_week_of_year

Create Week of Year Stacked Bar Plot

Description

Create Week of Year Stacked Bar Plot

Usage

```
create_plot_week_of_year(rogue_conditions, date_caption = NULL)
```

Arguments

rogue_conditions	Data frame with rogue wave conditions
date_caption	Date range caption

Value

plotly object

Examples

```
## Not run:
conditions <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:30) * 86400,
  rogue_ratio = runif(30, 2.0, 2.5),
  hmax = runif(30, 8, 15)
)
create_plot_week_of_year(conditions)

## End(Not run)
```

`create_plot_wind_beaufort` *Create Wind Speed by Beaufort Scale Plot*

Description

Create Wind Speed by Beaufort Scale Plot

Usage

```
create_plot_wind_beaufort(rogue_conditions, date_caption = NULL)
```

Arguments

`rogue_conditions`	Data frame with rogue wave conditions
`date_caption`	Date range caption

Value

plotly object

Examples

```
## Not run:
conditions <- data.frame(
  time = as.POSIXct("2024-01-01") + (1:10) * 3600,
  station_id = rep(c("M2", "M3"), 5),
  wind_speed = runif(10, 5, 35),
  rogue_ratio = runif(10, 2.0, 2.5),
  hmax = runif(10, 8, 15), wave_height = runif(10, 3, 6)
)
create_plot_wind_beaufort(conditions)

## End(Not run)
```

`create_return_level_plot_data` *Create Return Level Plot Data*

Description

Generates data for a return level plot showing the fitted distribution and confidence intervals.

Usage

```
create_return_level_plot_data(fit, max_return_period = 200, n_points = 100)
```

Arguments

`fit`	Result from fit_gev_annual_maxima or fit_gpd_threshold
`max_return_period`	Maximum return period to plot (default: 200)
`n_points`	Number of points for the curve (default: 100)

Value

Data frame suitable for plotting

`create_storm_alert_email` *Create Storm Alert Email*

Description

Composes an HTML email showing forecasts for ALL stations, with storm stations highlighted. Stations sorted by max Beaufort descending.

Usage

```
create_storm_alert_email(
  storm_events,
  station_info = get_station_info(),
  all_forecasts = NULL,
  threshold_knots = 41,
  met_warnings = NULL,
  forecast_rogue_summary = NULL
)
```

Arguments

storm_events	Tibble from detect_storm_events() (above-threshold events).
station_info	Data frame from get_station_info() (default).
all_forecasts	Full forecast tibble from fetch_all_forecasts() for all stations (used to show context for calm stations). If NULL, only storm stations shown.
threshold_knots	Numeric threshold used for triggering (default 41).
met_warnings	Character vector from fetch_met_eireann_warnings(), or NULL.

Value

A blastula email object.

See Also

Other storm-alert: beaufort_to_description(), detect_storm_events(), fetch_all_forecasts(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), send_storm_alert(), summarise_forecast_rogue_risk()

Examples

```
## Not run:
events <- tibble::tibble(
  station_id = "M2", time = Sys.time(),
  wind_speed_kn = 40, wind_gust_kn = 55,
  beaufort = 8L, description = "Gale", is_gust_driven = FALSE
)
create_storm_alert_email(events)

## End(Not run)
```

---

create_validation_summary

Create a validation summary for the pipeline

Description

Generates a summary of all validation results that can be included in dashboards or reports.

Usage

```
create_validation_summary(...)
```

Arguments

...	Named validation agents from interrogate()

Value

A tibble summarizing validation results

`cross_correlation_stations` *Calculate Cross-Correlation Between Two Stations*

Description

Computes cross-correlation function (CCF) between two stations for a given variable, identifying the optimal lag for prediction.

Usage

```
cross_correlation_stations(
  data,
  station1,
  station2,
  variable = "wave_height",
  max_lag = 48
)
```

Arguments

`data`	Data frame with columns: time, station_id, and the variable
`station1, station2`	Station IDs to compare
`variable`	Variable to analyze (default: "wave_height")
`max_lag`	Maximum lag in hours to test (default: 48)

Value

List with:

- ccf: cross-correlation values at each lag
- optimal_lag: lag (hours) with maximum correlation
- max_correlation: correlation at optimal lag
- lag_hours: vector of lag values

Examples

```
## Not run:
data <- data.frame(
  time = rep(seq(as.POSIXct("2024-01-01"), by = "hour", length.out = 100), 2),
  station_id = rep(c("M2", "M3"), each = 100),
  wave_height = c(rnorm(100, 3, 1), rnorm(100, 2.5, 0.8))
)
cross_correlation_stations(data, "M2", "M3")

## End(Not run)
```

---

decompose_stl *Perform STL Decomposition*

Description

Applies Seasonal-Trend decomposition using Loess (STL) to a time series. This separates the signal into seasonal, trend, and remainder components.

Usage

```
decompose_stl(
  data,
  variable = "wave_height",
  time_col = "time",
  frequency = "daily"
)
```

Arguments

data	Data frame with time and value columns
variable	Name of the variable to decompose (default: "wave_height")
time_col	Name of the time column (default: "time")
frequency	Seasonal frequency (default: "daily" = 24 hours)

Value

List with:

- decomposition: stl object
- components: data frame with time, seasonal, trend, remainder
- summary: summary statistics of each component

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, stations = "M3")
stl_result <- decompose_stl(data)
DBI::dbDisconnect(con)

## End(Not run)
```

`detect_anomalies` *Detect Anomalies*

Description

Identifies anomalous values using standard deviation thresholds relative to seasonal norms.

Usage

```
detect_anomalies(
  data,
  variable = "wave_height",
  time_col = "time",
  threshold = 3
)
```

Arguments

`data`	Data frame with time and value columns
`variable`	Name of the variable (default: "wave_height")
`time_col`	Name of the time column (default: "time")
`threshold`	Number of standard deviations for anomaly detection (default: 3)

Value

List with:

- anomalies: data frame of anomalous observations
- seasonal_norms: monthly mean and sd used as baseline
- summary: count of anomalies by month

Examples

```
set.seed(1)
data <- data.frame(
  time = seq(as.POSIXct("2020-01-01"), by = "hour", length.out = 1000),
  wave_height = 2 + sin(seq(0, 20, length.out = 1000)) + rnorm(1000, 0, 0.3)
)
result <- detect_anomalies(data)
nrow(result$anomalies)
result$summary
```

`detect_outliers_iqr` *Detect Outliers using IQR Method*

Description

Identifies outliers using the interquartile range (IQR) method. Values beyond Q1 - multiplier*IQR or Q3 + multiplier*IQR are flagged.

Usage

```
detect_outliers_iqr(data, variable = "wave_height", multiplier = 1.5)
```

Arguments

`data`	Data frame with the variable to check
`variable`	Name of the variable (default: "wave_height")
`multiplier`	IQR multiplier for outlier threshold (default: 1.5)

Value

The input data frame with an additional `is_outlier` logical column.

Examples

```
data <- data.frame(x = c(1:20, 100))
detect_outliers_iqr(data, variable = "x")
```

`detect_rogue_waves` *Rogue Wave Detection and Analysis*

Description

Functions for detecting and analyzing rogue waves from buoy data. Rogue waves are defined as waves where Hmax > threshold * WaveHeight.

Standard definition: Hmax > 2.0 * significant wave height Extreme definition: Hmax > 2.2 * significant wave height Detect Rogue Waves in Buoy Data

Identifies rogue wave events based on the ratio of maximum wave height (Hmax) to significant wave height (WaveHeight). Uses dplyr verbs translated to SQL for efficient DuckDB execution.

Usage

```
detect_rogue_waves(
  con,
  threshold = 2,
  min_wave_height = 2,
  start_date = NULL,
  end_date = NULL,
  stations = NULL
)
```

Arguments

`con`	DBI connection to DuckDB database
`threshold`	Hmax/WaveHeight ratio threshold (default: 2.0)
`min_wave_height`	Minimum significant wave height to consider (default: 2m)
`start_date`	Optional start date filter
`end_date`	Optional end date filter
`stations`	Optional vector of station IDs to filter

Value

Data frame of rogue wave events with associated conditions

Examples

```
## Not run:
con <- connect_duckdb()
rogues <- detect_rogue_waves(con, threshold = 2.0)
DBI::dbDisconnect(con)

## End(Not run)
```

`detect_storm_events` *Detect Storm Events from Forecast Data*

Description

Filters forecast data for wind speeds at or above the storm threshold. Threshold is resolved in order: `threshold_knots` parameter, then `STORM_ALERT_THRESHOLD_KNOTS` env var, then default of 41 knots (Beaufort 9).

Usage

```
detect_storm_events(forecasts, threshold_knots = NULL, use_gusts = FALSE)
```

Arguments

`forecasts`	Tibble from `fetch_all_forecasts()` or `fetch_open_meteo_forecast()`.
`threshold_knots`	Numeric threshold in knots (default NULL, uses env var or 41).
`use_gusts`	Logical; if TRUE, also flag rows where gusts exceed threshold. Default FALSE — only sustained wind speed triggers alerts.

Value

Tibble with columns: station_id, time, wind_speed_kn, wind_gust_kn, beaufort, description, is_gust_driven. Empty tibble if no storms detected.

See Also

Other storm-alert: `beaufort_to_description()`, `create_storm_alert_email()`, `fetch_all_forecasts()`, `fetch_all_marine_forecasts()`, `fetch_met_eireann_warnings()`, `fetch_open_meteo_forecast()`, `fetch_open_meteo_marine()`, `knots_to_beaufort()`, `p_hmax_exceedance()`, `send_storm_alert()`, `summarise_forecast_rogue_risk()`

Examples

```
forecasts <- tibble::tibble(
  station_id = "M2",
  time = Sys.time() + 3600 * 1:3,
  wind_speed_kn = c(20, 38, 50),
  wind_gust_kn = c(25, 45, 60)
)
detect_storm_events(forecasts)
```

`download_buoy_data` *Download Data from Irish Weather Buoy Network ERDDAP Server*

Description

Downloads data from the Marine Institute's ERDDAP server for the Irish Weather Buoy Network. Supports filtering by date range, stations, and variables.

Usage

```
download_buoy_data(
  start_date = Sys.Date() - 30,
  end_date = Sys.Date(),
  stations = NULL,
  variables = NULL,
  format = "csv"
)
```

Arguments

`start_date`	Character or Date, start of date range (default: 30 days ago)
`end_date`	Character or Date, end of date range (default: today)
`stations`	Character vector of station IDs (default: all stations)
`variables`	Character vector of variable names (default: all variables)
`format`	Character, output format: "csv", "json", or "tsv" (default: "csv")

Value

Data frame containing the requested buoy data

Examples

```
## Not run:
# Get last 7 days of data for all stations
data <- download_buoy_data(
  start_date = Sys.Date() - 7,
  end_date = Sys.Date()
)

# Get specific variables for M3 buoy
wave_data <- download_buoy_data(
  stations = "M3",
  variables = c("time", "WaveHeight", "WavePeriod", "Hmax")
)

## End(Not run)
```

evaluate_wave_model *Evaluate Wave Height Model*

Description

Evaluates model performance on test data.

Usage

```
evaluate_wave_model(model_result, data, target = "wave_height")
```

Arguments

model_result	Result from train_wave_model
data	Full prepared data frame
target	Target variable name (default: "wave_height")

Value

Data frame with performance metrics

explain_hourly_averaging *Explain Hourly Averaging Process*

Description

Educational function explaining how raw measurements become hourly values.

Usage

```
explain_hourly_averaging()
```

Value

Character string with explanation

Examples

```
cat(explain_hourly_averaging())
```

explain_hs_formula *Explain Why Hs Equals 4 Times Standard Deviation*

Description

Educational function explaining the physical and statistical basis for the relationship Hs = 4 * sigma.

Usage

```
explain_hs_formula()
```

Value

Character string with explanation

Examples

```
cat(explain_hs_formula())
```

explain_measurement_period *Explain the 17.5-Minute Measurement Period*

Description

Educational function explaining why wave measurements use specific time periods for statistical validity.

Usage

```
explain_measurement_period()
```

Value

Character string with explanation

Examples

```
cat(explain_measurement_period())
```

explain_wave_height_measurement

Explain How Individual Wave Heights Are Measured (Zero-Crossing Method)

Description

Educational function explaining how individual wave heights like Hmax are measured, and how this differs from the statistical Hs calculation.

Usage

```
explain_wave_height_measurement()
```

Value

Character string with explanation

Examples

```
cat(explain_wave_height_measurement())
```

fetch_all_forecasts

Fetch Forecasts for All Buoy Stations

Description

Loops over all stations from `get_station_info()` and fetches wind forecasts.

Usage

```
fetch_all_forecasts(
  station_info = get_station_info(),
  forecast_days = 7,
  timeout = 30
)
```

Arguments

`station_info`	Data frame with station_id, lat, lon columns (default from `get_station_info()`).
`forecast_days`	Integer number of forecast days (default 7).
`timeout`	Numeric request timeout in seconds (default 30).

Value

Combined tibble of all station forecasts.

See Also

Other storm-alert: beaufort_to_description(), create_storm_alert_email(), detect_storm_events(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), send_storm_alert(), summarise_forecast_rogue_risk()

Examples

```
## Not run:
fetch_all_forecasts()

## End(Not run)
```

---

fetch_all_marine_forecasts

Fetch Marine Wave Forecasts for All Buoy Stations

Description

Loops over all stations from get_station_info() and fetches Open-Meteo Marine API forecasts. Soft dependency: any per-station failure is logged and skipped, never aborts.

Usage

```
fetch_all_marine_forecasts(
  station_info = get_station_info(),
  forecast_days = 7,
  timeout = 30
)
```

Arguments

station_info	Data frame with station_id, lat, lon columns.
forecast_days	Integer number of forecast days (default 7).
timeout	Numeric request timeout in seconds (default 30).

Value

Combined tibble of all station marine forecasts. Empty tibble if every station failed.

See Also

Other storm-alert: beaufort_to_description(), create_storm_alert_email(), detect_storm_events(), fetch_all_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), send_storm_alert(), summarise_forecast_rogue_risk()

Examples

```
## Not run:
fetch_all_marine_forecasts()

## End(Not run)
```

`fetch_met_eireann_warnings` *Fetch Met Eireann Marine Warnings*

Description

Fetches the latest marine forecast/warning text from Met Eireann's open data. Returns NULL on any error (best-effort supplementary info).

Usage

```
fetch_met_eireann_warnings(timeout = 10)
```

Arguments

`timeout`	Numeric request timeout in seconds (default 10).

Value

Character vector of warning lines, or NULL if unavailable.

See Also

Other storm-alert: `beaufort_to_description()`, `create_storm_alert_email()`, `detect_storm_events()`, `fetch_all_forecasts()`, `fetch_all_marine_forecasts()`, `fetch_open_meteo_forecast()`, `fetch_open_meteo_marine()`, `knots_to_beaufort()`, `p_hmax_exceedance()`, `send_storm_alert()`, `summarise_forecast_rogue_risk()`

Examples

```
## Not run:
fetch_met_eireann_warnings()

## End(Not run)
```

`fetch_open_meteo_forecast` *Fetch Wind Forecast from Open-Meteo for a Single Station*

Description

Queries the Open-Meteo API for hourly wind speed and gust forecasts at a given latitude/longitude. Returns an empty tibble on error.

Usage

```
fetch_open_meteo_forecast(
  lat,
  lon,
  station_id,
  forecast_days = 7,
  timeout = 30
)
```

Arguments

`lat`	Latitude in decimal degrees.
`lon`	Longitude in decimal degrees.
`station_id`	Character station identifier (e.g. "M2").
`forecast_days`	Integer number of forecast days (1-16, default 7).
`timeout`	Numeric request timeout in seconds (default 30).

Value

Tibble with columns: station_id, time, wind_speed_kn, wind_gust_kn, forecast_fetched_at. Empty tibble on error.

See Also

Other storm-alert: `beaufort_to_description()`, `create_storm_alert_email()`, `detect_storm_events()`, `fetch_all_forecasts()`, `fetch_all_marine_forecasts()`, `fetch_met_eireann_warnings()`, `fetch_open_meteo_marine()`, `knots_to_beaufort()`, `p_hmax_exceedance()`, `send_storm_alert()`, `summarise_forecast_rogue_risk()`

Examples

```
## Not run:
fetch_open_meteo_forecast(51.22, -9.99, "M2", forecast_days = 1)

## End(Not run)
```

`fetch_open_meteo_marine` *Fetch Marine Wave Forecast from Open-Meteo for a Single Station*

Description

Queries the Open-Meteo Marine Weather API for hourly significant wave height, wave period, wind-wave and swell components at a given lat/lon. Returns an empty tibble on error (soft dependency — never aborts the pipeline).

Source: https://open-meteo.com/en/docs/marine-weather-api. Underlying models are DWD EWAM (European) and GWAM (global), ~25 km grid.

Usage

```
fetch_open_meteo_marine(lat, lon, station_id, forecast_days = 7, timeout = 30)
```

Arguments

`lat`	Latitude in decimal degrees.
`lon`	Longitude in decimal degrees.
`station_id`	Character station identifier (e.g. "M2").
`forecast_days`	Integer number of forecast days (1-8 for marine, default 7).
`timeout`	Numeric request timeout in seconds (default 30).

Value

Tibble with columns: station_id, time, wave_height_m, wave_period_s, wind_wave_height_m, swell_wave_height_m, forecast_fetched_at. Empty tibble on error.

See Also

Other storm-alert: `beaufort_to_description()`, `create_storm_alert_email()`, `detect_storm_events()`, `fetch_all_forecasts()`, `fetch_all_marine_forecasts()`, `fetch_met_eireann_warnings()`, `fetch_open_meteo_forecast()`, `knots_to_beaufort()`, `p_hmax_exceedance()`, `send_storm_alert()`, `summarise_forecast_rogue_risk()`

Examples

```
## Not run:
fetch_open_meteo_marine(51.22, -9.99, "M2", forecast_days = 1)

## End(Not run)
```

fit_bivariate_copula *Fit Bivariate Copula for Joint Extremes*

Description

Fits a copula model to capture the joint dependence structure between two stations, especially in the tails (extremes).

Usage

```
fit_bivariate_copula(
  data,
  station1,
  station2,
  variable = "wave_height",
  copula_family = "gumbel"
)
```

Arguments

data	Data frame with columns: time, station_id, and the variable
station1, station2	Station IDs to analyze
variable	Variable to analyze (default: "wave_height")
copula_family	Copula family: "gaussian", "t", "clayton", "gumbel", "frank"

Value

List with:

- copula: fitted copula object
- parameters: copula parameters
- tau: Kendall's tau (rank correlation)
- tail_dependence: lower and upper tail dependence coefficients

fit_gev_annual_maxima *Fit GEV Distribution to Annual Maxima*

Description

Fits a Generalized Extreme Value distribution to annual maximum values. This is the Block Maxima approach to extreme value analysis.

Usage

```
fit_gev_annual_maxima(
  data,
  variable = "wave_height",
  time_col = "time",
  min_years = 5
)
```

Arguments

data	Data frame with columns: time, value (the variable to analyze)
variable	Name of the variable column (default: "wave_height")
time_col	Name of the time column (default: "time")
min_years	Minimum years of data required (default: 5)

Value

List with:

- fit: extRemes fevd object
- annual_maxima: data frame of annual maxima
- parameters: GEV parameters (location, scale, shape)
- diagnostics: model diagnostic information

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "wave_height"))
gev_result <- fit_gev_annual_maxima(data)
DBI::dbDisconnect(con)

## End(Not run)
```

---

fit_gpd_threshold *Fit GPD Distribution to Threshold Exceedances*

Description

Fits a Generalized Pareto Distribution to values exceeding a threshold. This is the Peaks Over Threshold (POT) approach.

Usage

```
fit_gpd_threshold(
  data,
  variable = "wave_height",
  threshold = NULL,
  decluster = TRUE,
  decluster_hours = 48
)
```

Arguments

data	Data frame with the variable to analyze
variable	Name of the variable column (default: "wave_height")
threshold	Threshold value (default: NULL, uses 95th percentile)
decluster	Logical, whether to decluster exceedances (default: TRUE)
decluster_hours	Minimum hours between independent exceedances (default: 48)

Value

List with:

- fit: extRemes fevd object
- exceedances: data frame of exceedances
- threshold: threshold used
- parameters: GPD parameters (scale, shape)

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "wave_height"))
gpd_result <- fit_gpd_threshold(data, threshold = 6)
DBI::dbDisconnect(con)

## End(Not run)
```

---

fit_spatial_maxstable *Fit a Max-Stable Process to Station Annual Maxima*

Description

Fits a Brown-Resnick max-stable process model to annual block maxima of wave heights across multiple stations. Margins are first transformed to unit Frechet using the empirical CDF. If the Brown-Resnick model fails to converge, a Schlather model (Whittle-Matern covariance) is tried as fallback.

Limitation: Max-stable models require many spatial locations (typically= 20) for reliable estimation. With only 5 buoy stations, results areillustrative and the information matrix is often singular.

Usage

```
fit_spatial_maxstable(
  data,
  variable = "wave_height",
  station_info = NULL,
  min_years = 5
)
```

Arguments

`data`	Data frame with columns: `time` (POSIXct), `station_id` (character), and the variable specified by `variable`.
`variable`	Variable to analyze (default: `"wave_height"`).
`station_info`	Optional data frame with station metadata (from `get_station_info()`). Must contain `station_id`, `lat`, `lon`. If `NULL`, uses the default 5-station network.
`min_years`	Minimum number of complete years required across all stations (default: 5).

Value

List with:

fitted Logical: whether a max-stable model was successfully fitted.

fit The fitted model object (from `SpatialExtremes::fitmaxstab`), or `NULL` if fitting failed.

model_type Character: `"brown_resnick"`, `"schlather"`, or `NA`.

parameters Named numeric vector of fitted parameters, or `NULL`.

annual_maxima Data frame of annual maxima per station (long format).

coords Coordinate matrix (lon, lat) used for fitting.

limitation Character string describing the illustrative nature of results with few stations.

If fitting fails entirely, `fitted = FALSE` and a `reason` field explains why.

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "station_id", "wave_height"))
result <- fit_spatial_maxstable(data)
if (result$fitted) print(result$parameters)
DBI::dbDisconnect(con)

## End(Not run)
```

`generate_and_send_summary` *Generate and Send Summary Email*

Description

Main function to generate summary and send via email. Requires GMAIL_USERNAME and GMAIL_APP_PASSWORD environment variables.

Usage

```
generate_and_send_summary(
  recipient = Sys.getenv("GMAIL_USERNAME"),
  sender = Sys.getenv("GMAIL_USERNAME")
)
```

Arguments

`recipient`	Email recipient (default from GMAIL_USERNAME env var)
`sender`	Email sender (default from GMAIL_USERNAME env var)

`generate_api_decomposition` *Generate Decomposition Endpoint*

Description

Returns STL decomposition results per station, downsampled to daily resolution to keep JSON under 1MB.

Usage

```
generate_api_decomposition(decomp_per_station)
```

Arguments

`decomp_per_station`
: Named list of per-station decomposition results. Each element should have `components` (data.frame with time, seasonal, trend, remainder), `summary`, and `variable`.

Value

A list with `_meta` and `data` fields.

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_extremes(), generate_api_gust_factor
generate_api_index(), generate_api_latest(), generate_api_methods(), generate_api_sources(),
generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

`generate_api_extremes` *Generate Extremes Endpoint*

Description

Combines GPD return levels and CI comparison (delta, bootstrap, order-statistics) into a single endpoint.

Usage

```
generate_api_extremes(return_levels_per_station, ci_comparison_per_station)
```

Arguments

`return_levels_per_station`
: Tibble with columns `return_period`, `return_level`, `lower`, `upper`, `station`, `variable`, `variable_label`.

`ci_comparison_per_station`
: Tibble with columns `return_period`, `return_level`, `lower`, `upper`, `station`, `variable`, `method`.

Value

A list with `_meta` and `data` fields.

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_decomposition(),
generate_api_gust_factors(), generate_api_index(), generate_api_latest(), generate_api_methods(),
generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(),
run_api()

generate_api_gust_factors *Generate Gust Factors Endpoint*

Description

Returns gust factor analysis results per station, capped at 500 extreme events to keep JSON under 1MB.

Usage

```
generate_api_gust_factors(gust_analysis)
```

Arguments

gust_analysis	List from analyze_gust_factor() containing summary, extreme_gusts, by_station, rogue_gust_threshold, n_rogue_gusts, pct_rogue_gusts.

Value

A list with _meta and data fields.

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_decomposition(), generate_api_extremes(), generate_api_index(), generate_api_latest(), generate_api_methods(), generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

generate_api_index *Generate API Index*

Description

Creates a JSON-serialisable list describing all available API endpoints. Used to generate index.json at the API root.

Usage

```
generate_api_index(
  base_url = "https://johngavin.github.io/irishbuoys/api/v1/",
  endpoints = NULL
)
```

Arguments

base_url	Character, base URL for the API (default: "https://johngavin.github.io/irishbuoys/api/v1/")
endpoints	Named list of endpoint metadata. Each element should have description and optionally fields. If NULL, uses default endpoints.

Value

A list suitable for jsonlite::toJSON().

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_decomposition(), generate_api_extremes(), generate_api_gust_factors(), generate_api_latest(), generate_api_methods(), generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

Examples

```
## Not run:
idx <- generate_api_index()
jsonlite::toJSON(idx, pretty = TRUE, auto_unbox = TRUE)

## End(Not run)
```

---

generate_api_latest *Generate Latest Observations*

Description

Queries DuckDB for the most recent n observations per station. Returns a tibble suitable for JSON serialisation.

Usage

```
generate_api_latest(db_path = "inst/extdata/irish_buoys.duckdb", n = 1L)
```

Arguments

db_path	Character, path to the DuckDB database file (default: "inst/extdata/irish_buoys.duckdb")
n	Integer, number of most recent observations per station to return (default: 1L)

Value

A tibble with n rows per station, ordered by station and time (most recent first).

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_decomposition(), generate_api_extremes(), generate_api_gust_factors(), generate_api_index(), generate_api_methods(), generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

Examples

```
## Not run:
latest <- generate_api_latest(n = 1)
latest_5 <- generate_api_latest(n = 5)

## End(Not run)
```

---

generate_api_methods *Generate Methods Endpoint*

Description

Returns statistical methods documentation: thresholds, formulas, references. Pure function with no upstream target dependency.

Usage

```
generate_api_methods()
```

Value

A list with _meta and data fields.

See Also

Other api: api_plumber, api_static, create_api_router(), generate_api_decomposition(), generate_api_extremes(), generate_api_gust_factors(), generate_api_index(), generate_api_latest(), generate_api_sources(), generate_api_spatial(), generate_api_status(), generate_api_trends(), run_api()

Examples

```
methods <- generate_api_methods()
names(methods)
```

`generate_api_sources` *Generate Data Sources Endpoint*

Description

Returns data provenance constants: ERDDAP URL, dataset ID, update frequency, license, and citation.

Usage

```
generate_api_sources(update_frequency = NULL)
```

Arguments

`update_frequency`	Character, human-readable update schedule. If NULL (default), uses "Every 6 hours (0:00, 6:00, 12:00, 18:00 UTC)". Typically supplied dynamically from the `api_update_schedule` target.

Value

A list with `_meta` and `data` fields suitable for `jsonlite::toJSON()`.

See Also

Other api: `api_plumber`, `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_spatial()`, `generate_api_status()`, `generate_api_trends()`, `run_api()`

Examples

```
## Not run:
src <- generate_api_sources()
jsonlite::toJSON(src, pretty = TRUE, auto_unbox = TRUE)

## End(Not run)
```

`generate_api_spatial` *Generate Spatial Correlations Endpoint*

Description

Returns cross-station correlation matrices for wave height, wind speed, and Hmax.

Usage

```
generate_api_spatial(pair_wave, pair_wind, pair_hmax)
```

Arguments

`pair_wave`	Data frame from `analyze_station_pairs()` for wave height.
`pair_wind`	Data frame from `analyze_station_pairs()` for wind speed.
`pair_hmax`	Data frame from `analyze_station_pairs()` for Hmax.

Value

A list with `_meta` and `data` fields.

See Also

Other api: `api_plumber`, `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_status()`, `generate_api_trends()`, `run_api()`

Examples

```
## Not run:
dm <- data.frame(station1 = "M2", station2 = "M3", distance_km = 150)
cr <- data.frame(station1 = "M2", station2 = "M3", correlation = 0.85)
ed <- data.frame(station1 = "M2", station2 = "M3", chi = 0.3)
generate_api_spatial(dm, cr, ed)

## End(Not run)
```

generate_api_status *Generate Station Status Endpoint*

Description

Returns per-station operational status including record counts and date ranges. Reuses `dashboard_stats` target output.

Usage

```
generate_api_status(dashboard_stats)
```

Arguments

`dashboard_stats`	List, output from the `dashboard_stats` target containing `station` (tibble) and `overall` (list) elements.

Value

A list with `_meta` and `data` fields.

See Also

Other api: `api_plumber`, `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_spatial()`, `generate_api_trends()`, `run_api()`

generate_api_trends *Generate Trends Endpoint*

Description

Returns Mann-Kendall trend tests per station/variable and overall annual trend statistics for wave height and wind speed.

Usage

```
generate_api_trends(
  mann_kendall_per_station,
  annual_trends_wave,
  annual_trends_wind
)
```

Arguments

`mann_kendall_per_station`
: Named list of per-station Mann-Kendall results. Each element is a station, containing named sub-elements for each variable (e.g. `wave_height`, `wind_speed`), each with `tau`, `p_value`, `trend_direction`.

`annual_trends_wave`
: List with `annual_stats`, `trend_per_decade`, `p_value`, `r_squared`.

`annual_trends_wind`
: Same structure as `annual_trends_wave`.

Value

A list with `_meta` and `data` fields.

See Also

Other api: `api_plumber`, `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_spatial()`, `generate_api_status()`, `run_api()`

`generate_validation_reports` *Generate validation reports for website*

Description

Creates pointblank validation reports and saves them to the docs directory for inclusion in the pkgdown/GitHub Pages website.

Usage

```
generate_validation_reports(
  analysis_data,
  rogue_events,
  output_dir = "docs/articles"
)
```

Arguments

`analysis_data`	The analysis_data tibble to validate
`rogue_events`	The rogue_wave_events tibble to validate
`output_dir`	Directory to save reports (default: "docs/articles")

Value

A list with paths to generated reports

generate_weekly_summary *Generate Weekly Summary Statistics*

Description

Compares recent data against historical averages to identify trends and anomalies. Optionally includes data ingestion statistics.

Usage

```
generate_weekly_summary(
  db_path = "inst/extdata/irish_buoys.duckdb",
  lookback_days = 7,
  qc_filter = NULL,
  update_result = NULL
)
```

Arguments

db_path	Path to DuckDB database
lookback_days	Number of days to analyze (default: 7)
qc_filter	QC flag filter: 1 = good only, 0 = include unverified, NULL = no filter
update_result	Optional result from incremental_update() containing ingestion stats

Value

List containing summary statistics and comparisons

get_data_dictionary *Irish Weather Buoy Network Data Dictionary*

Description

This function returns a comprehensive data dictionary for all variables available in the Irish Weather Buoy Network dataset. Each entry includes the variable name, units, data type, description, and typical range.

Usage

```
get_data_dictionary()
```

Value

A data frame containing the complete data dictionary with columns:

- variable: Variable name as used in the dataset
- category: Category (dimension, meteorological, oceanographic, quality)
- units: Measurement units
- data_type: R data type
- description: Detailed description of the variable
- typical_range: Typical or valid range of values

Examples

```
dict <- get_data_dictionary()
print(dict)
```

---

get_database_stats *Get Database Statistics*

Description

Returns summary statistics about the current state of the database.

Usage

```
get_database_stats(db_path = "inst/extdata/irish_buoys.duckdb")
```

Arguments

db_path	Path to DuckDB database file

Value

List with database statistics

Examples

```
## Not run:
stats <- get_database_stats()
print(stats)

## End(Not run)
```

get_latest_timestamp *Get Latest Data Timestamp from ERDDAP*

Description

Queries the ERDDAP server to find the most recent data timestamp available for the Irish Weather Buoy Network.

Usage

```
get_latest_timestamp(station = NULL)
```

Arguments

station	Optional station ID to check specific buoy

Value

POSIXct timestamp of most recent data

Examples

```
## Not run:
latest <- get_latest_timestamp()
latest_m3 <- get_latest_timestamp("M3")

## End(Not run)
```

get_station_info *Station Information with Coordinates*

Description

Returns a data frame with station metadata including coordinates and depths.

Usage

```
get_station_info()
```

Value

Data frame with columns: station_id, location, lat, lon, depth_m, distance_km

Examples

```
get_station_info()
```

get_stations *Get Available Stations*

Description

Returns a data frame with information about all available weather buoy stations.

Usage

```
get_stations()
```

Value

Data frame with station metadata

Examples

```
## Not run:
stations <- get_stations()

## End(Not run)
```

get_variable_docs *Get Detailed Variable Documentation*

Description

Returns extended documentation for specific variables including scientific context, calculation methods, and usage notes.

Usage

```
get_variable_docs(variable = NULL)
```

Arguments

variable	Character string specifying the variable name

Value

List containing detailed documentation

Examples

```
doc <- get_variable_docs("WaveHeight")
```

haversine_distance *Calculate Distance Between Two Stations*

Description

Calculates the great-circle distance between two points using the Haversine formula.

Usage

```
haversine_distance(lat1, lon1, lat2, lon2)
```

Arguments

lat1, lon1	Coordinates of first point (degrees)
lat2, lon2	Coordinates of second point (degrees)

Value

Distance in kilometers

Examples

```
# Distance from M6 to M2
haversine_distance(53.07, -15.93, 51.22, -9.99)
```

hs_from_rms *Estimate Hs from RMS Wave Height*

Description

Converts RMS wave height to significant wave height using the theoretical relationship for Rayleigh-distributed waves.

Usage

```
hs_from_rms(h_rms)
```

Arguments

h_rms	RMS wave height in meters

Details

For Rayleigh-distributed waves: Hs = H_rms * sqrt(8) ~ 2.83 * H_rms

Value

Significant wave height in meters

Examples

```
h_rms <- 1.5
hs <- hs_from_rms(h_rms)  # Returns ~4.24 m
```

`ib_hf_connect` *Create a DuckDB connection for reading HuggingFace Parquet*

Description

Returns a DBI connection to an ephemeral DuckDB instance. DuckDB 0.10+ supports `hf://datasets/...` natively. httpfs is loaded as a fallback for non-HF HTTPS URLs.

Usage

```
ib_hf_connect()
```

Value

DBI connection object

See Also

Other huggingface: `ib_hf_online()`, `ib_hf_url()`

Examples

```
## Not run:
con <- ib_hf_connect()
dplyr::tbl(con, ib_hf_url()) |> dplyr::glimpse()
DBI::dbDisconnect(con)

## End(Not run)
```

ib_hf_online *Check if HuggingFace dataset is reachable*

Description

Returns TRUE if the HF API responds within 5 seconds. Used by tests and examples to fall back to local sample data.

Usage

```
ib_hf_online()
```

Value

Logical

See Also

Other huggingface: ib_hf_connect(), ib_hf_url()

Examples

```
ib_hf_online()
```

ib_hf_url *Construct HuggingFace dataset URL for buoy data*

Description

DuckDB 0.10+ supports hf://datasets/... natively — no httpfs extension needed, 34% faster than resolve/main/ URLs.

Usage

```
ib_hf_url(filename = "buoy_data.parquet")
```

Arguments

filename	Parquet filename (default: "buoy_data.parquet")

Value

hf://datasets/{repo}/{filename} URL string

See Also

Other huggingface: ib_hf_connect(), ib_hf_online()

Examples

```
ib_hf_url()
ib_hf_url("stations.json")
```

`incremental_update` *Perform Incremental Data Update*

Description

Downloads new data since the last update and appends it to the database. Designed to be run on a schedule (e.g., daily or weekly via cron/GitHub Actions).

Usage

```
incremental_update(
  db_path = "inst/extdata/irish_buoys.duckdb",
  lookback_hours = 48
)
```

Arguments

`db_path`	Path to DuckDB database file
`lookback_hours`	Number of hours to look back for safety (default: 48) This ensures we don't miss data due to delays in ERDDAP updates

Value

List with update statistics

Examples

```
## Not run:
# Perform incremental update
result <- incremental_update()

# Check what was updated
print(result$summary)

## End(Not run)
```

incremental_update_parquet *Incremental Update with Parquet Storage*

Description

Efficiently append new data to Parquet files. Only writes new partitions or updates existing ones.

Usage

```
incremental_update_parquet(new_data, data_path = "inst/extdata/parquet")
```

Arguments

new_data	New data to append
data_path	Base path for Parquet files

init_parquet_storage *Parquet-based Storage Backend for Irish Buoys Data*

Description

Uses Parquet files as storage backend with DuckDB as query engine. This provides excellent compression (5-10x) while maintaining query performance.

The architecture:

- Raw data stored in partitioned Parquet files (by year/month)
- DuckDB used as query engine (reads Parquet directly)
- Optional: DuckDB database for metadata and indexes only Initialize Parquet Storage Structure

Usage

```
init_parquet_storage(
  data_path = "inst/extdata/parquet",
  db_path = "inst/extdata/metadata.duckdb"
)
```

Arguments

data_path	Base path for Parquet files
db_path	Optional path for metadata database

initialize_database *Initialize Database with Historical Data*

Description

Downloads and loads a larger set of historical data into the database. Use this for initial setup or to rebuild the database.

Usage

```
initialize_database(
  db_path = "inst/extdata/irish_buoys.duckdb",
  start_date = Sys.Date() - 365,
  end_date = Sys.Date(),
  chunk_days = 365
)
```

Arguments

db_path	Path to DuckDB database file
start_date	Start date for historical data (default: 1 year ago)
end_date	End date for historical data (default: today)
chunk_days	Number of days to download at once (default: 30)

Value

Total number of records loaded

Examples

```
## Not run:
# Initialize with last year of data
records <- initialize_database(start_date = "2023-01-01")

## End(Not run)
```

irishbuoys_ggplotly *Apply Irish Buoys theme to ggplotly object*

Description

Wrapper for ggplotly that applies the standard irishbuoys theme. Useful when converting ggplot2 plots to plotly.

Usage

```
irishbuoys_ggplotly(gg, title = NULL, ...)
```

Arguments

gg	A ggplot2 object
title	Optional title to override ggplot title
...	Additional arguments passed to plotly::ggplotly()

Value

A styled plotly object

Examples

```
## Not run:
p <- ggplot2::ggplot(mtcars, ggplot2::aes(wt, mpg)) +
  ggplot2::geom_point()
irishbuoys_ggplotly(p)

## End(Not run)
```

---

irishbuoys_layout *Standard Plotly Theme for Irish Buoys Package*

Description

Applies consistent dark styling to all plotly plots in the irishbuoys package. Uses black background with white grid lines to match the Quarto cosmo dashboard theme. Bottom-positioned horizontal legend with dark hoverlabels.

Usage

```
irishbuoys_layout(p, title = NULL, ...)
```

Arguments

p	A plotly object
title	Optional title string
...	Additional arguments passed to plotly::layout()

Value

A styled plotly object

Examples

```
## Not run:
library(plotly)
p <- plot_ly(data = mtcars, x = ~wt, y = ~mpg, type = "scatter", mode = "markers")
p |> irishbuoys_layout(title = "Weight vs MPG")

## End(Not run)
```

`joint_analysis_summary` *Create Joint Analysis Summary*

Description

Comprehensive summary of joint dependencies across all stations.

Usage

```
joint_analysis_summary(data, variable = "wave_height")
```

Arguments

`data`	Data frame with buoy data
`variable`	Variable to analyze (default: "wave_height")

Value

List containing all joint analysis results

Examples

```
## Not run:
data <- data.frame(
  time = rep(seq(as.POSIXct("2024-01-01"), by = "hour", length.out = 100), 2),
  station_id = rep(c("M2", "M3"), each = 100),
  wave_height = c(rnorm(100, 3, 1), rnorm(100, 2.5, 0.8))
)
joint_analysis_summary(data)

## End(Not run)
```

knots_to_beaufort *Convert Wind Speed in Knots to Beaufort Scale*

Description

Vectorized conversion from wind speed in knots to the Beaufort scale (0-12).

Usage

```
knots_to_beaufort(wind_speed_kn)
```

Arguments

wind_speed_kn	Numeric vector of wind speeds in knots.

Value

Integer vector of Beaufort numbers (0-12).

See Also

Other storm-alert: beaufort_to_description(), create_storm_alert_email(), detect_storm_events(), fetch_all_forecasts(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), p_hmax_exceedance(), send_storm_alert(), summarise_forecast_rogue_risk()

Examples

```
knots_to_beaufort(c(0, 5, 20, 34, 48, 64))
```

load_to_duckdb *Load Data into DuckDB Database*

Description

Loads buoy data from a data frame into the DuckDB database. Handles duplicates by using ON CONFLICT DO NOTHING.

Usage

```
load_to_duckdb(data, con, update_metadata = TRUE)
```

Arguments

data	Data frame containing buoy data
con	DBI connection object
update_metadata	Logical, whether to update station metadata (default: TRUE)

Value

Number of rows inserted

Examples

```
## Not run:
# Download and load data
data <- download_buoy_data(start_date = "2024-01-01")
con <- connect_duckdb()
rows_added <- load_to_duckdb(data, con)
DBI::dbDisconnect(con)

## End(Not run)
```

---

mann_kendall_test *Mann-Kendall Trend Test*

Description

Performs a non-parametric Mann-Kendall trend test on a time series variable. Uses Kendall's tau via `stats::cor.test(method = "kendall")`.

Usage

```
mann_kendall_test(data, variable = "wave_height", time_col = "time")
```

Arguments

`data`	Data frame with time and value columns
`variable`	Name of the variable (default: "wave_height")
`time_col`	Name of the time column (default: "time")

Value

List with tau, p_value, and trend_direction ("increasing", "decreasing", or "no trend").

Examples

```
## Not run:
data <- data.frame(
  time = seq(as.POSIXct("2020-01-01"), by = "day", length.out = 365),
  wave_height = seq(2, 3, length.out = 365) + rnorm(365, 0, 0.2)
)
mann_kendall_test(data)

## End(Not run)
```

obs_status_label *Status Label from Observation Confidence*

Description

Maps a confidence multiplier to a short human-readable status label and a suggested colour for dashboard badges.

Usage

```
obs_status_label(confidence)
```

Arguments

confidence	Numeric vector of confidence values in [0.1, 1].

Value

List with label (character) and color (character hex), both the same length as confidence.

See Also

Other obs-confidence: compute_obs_confidence(), widen_ci()

Examples

```
obs_status_label(c(1, 0.7, 0.4, 0.15))
```

p_hmax_exceedance *Short-Term Probability of Maximum Wave Height Exceedance (Forristall)*

Description

Computes P(H_max > h | H_s, T_z, D) for a stationary sea state of significant wave height H_s, mean zero-crossing period T_z, lasting duration D, using the Forristall (1978) Weibull short-term distribution for individual wave heights.

Forristall (1978) gives P(H > h | H_s) = exp(-(h / (alpha * H_s))^beta) with alpha = 0.681 and beta = 2.126 (calibrated on Gulf of Mexico storm data). For N independent waves in the window, P(H_max <= h) = (1 - P(H > h))^N, so P(H_max > h) = 1 - (1 - exp(-(h/(alpha*H_s))^beta))^N, with N = D / T_z.

Reference: Forristall, G. Z. (1978). On the statistical distribution of wave heights in a storm. *Journal of Geophysical Research*, 83(C5), 2353-2358.

Usage

```
p_hmax_exceedance(h, hs, tz, duration_s = 3600, alpha = 0.681, beta = 2.126)
```

Arguments

h	Numeric vector of wave heights to evaluate (m).
hs	Numeric significant wave height (m), length 1 or length(h).
tz	Numeric mean zero-crossing period (s), length 1 or length(h).
duration_s	Numeric window duration in seconds (default 3600 = 1 hour).
alpha	Forristall scale parameter (default 0.681).
beta	Forristall shape parameter (default 2.126).

Value

Numeric vector of P(H_max > h) values in `[0, 1]`. Returns NA where `hs <= 0`, `tz <= 0`, or any input is NA.

See Also

Other storm-alert: `beaufort_to_description()`, `create_storm_alert_email()`, `detect_storm_events()`, `fetch_all_forecasts()`, `fetch_all_marine_forecasts()`, `fetch_met_eireann_warnings()`, `fetch_open_meteo_forecast()`, `fetch_open_meteo_marine()`, `knots_to_beaufort()`, `send_storm_alert()`, `summarise_forecast_rogue_risk()`

Examples

```
# Probability of a 20 m wave during a 1-hour window with Hs = 10 m, Tz = 9 s
p_hmax_exceedance(20, hs = 10, tz = 9, duration_s = 3600)
```

---

predict_station_lagged

Predict Station from Another with Optimal Lag

Description

Uses one station to predict another at the optimal lag. Particularly useful for M6 (offshore) predicting coastal stations.

Usage

```
predict_station_lagged(
  data,
  predictor_station,
  target_station,
  variable = "wave_height",
  lag_hours = NULL
)
```

Arguments

`data`	Data frame with columns: time, station_id, and the variable
`predictor_station`	Station to use as predictor (e.g., "M6")
`target_station`	Station to predict (e.g., "M2")
`variable`	Variable to predict (default: "wave_height")
`lag_hours`	Lag in hours (positive = predictor leads target)

Value

List with:

- model: lm object
- r_squared: R-squared of prediction
- rmse: Root mean squared error
- predictions: data frame with actual and predicted values

Examples

```
## Not run:
data <- data.frame(
  time = rep(seq(as.POSIXct("2024-01-01"), by = "hour", length.out = 200), 2),
  station_id = rep(c("M6", "M2"), each = 200),
  wave_height = c(rnorm(200, 3, 1), rnorm(200, 2.5, 0.8))
)
predict_station_lagged(data, "M6", "M2", lag_hours = 6)

## End(Not run)
```

`predict_wave_height` *Predict Wave Height*

Description

Predicts wave height for new observations.

Usage

```
predict_wave_height(model_result, new_data)
```

Arguments

`model_result`	Result from train_wave_model
`new_data`	Data frame with predictor values

Value

Numeric vector of predicted wave heights

prepare_wave_features *Prepare Features for Wave Height Prediction*

Description

Creates lagged features and derived variables for wave height prediction.

Usage

```
prepare_wave_features(data, lags = 1:3)
```

Arguments

data	Data frame with buoy observations
lags	Integer vector of lag periods in hours (default: 1:3)

Value

Data frame with additional lagged and derived features

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, qc_filter = FALSE)
features <- prepare_wave_features(data)
DBI::dbDisconnect(con)

## End(Not run)
```

query_buoy_data *Query Buoy Data from Database*

Description

Flexible querying of buoy data with various filtering options. Uses dplyr verbs translated to SQL for efficient DuckDB execution.

Usage

```
query_buoy_data(
  con,
  stations = NULL,
  start_date = NULL,
  end_date = NULL,
  variables = NULL,
  qc_filter = TRUE
)
```

Arguments

con	DBI connection object
stations	Character vector of station IDs (default: all)
start_date	Start date for query
end_date	End date for query
variables	Character vector of variables to return
qc_filter	Logical, filter for good quality data only (default: TRUE)

Value

Data frame with query results

Examples

```
## Not run:
con <- connect_duckdb()
# Get recent M3 wave data
waves <- query_buoy_data(
  con,
  stations = "M3",
  variables = c("time", "wave_height", "wave_period"),
  start_date = Sys.Date() - 7
)

## End(Not run)
```

---

query_parquet *Query Parquet Files with DuckDB*

Description

DuckDB can query Parquet files directly without importing. This provides excellent performance with minimal memory usage.

Usage

```
query_parquet(
  query = NULL,
  data_path = "inst/extdata/parquet/by_year_month",
  stations = NULL,
  date_range = NULL
)
```

Arguments

query	SQL query or NULL for interactive connection
data_path	Path to Parquet files
stations	Filter for specific stations
date_range	Date range as c(start_date, end_date)

Examples

```
## Not run:
# Query recent data
df <- query_parquet(
  "SELECT * FROM buoy_data WHERE wave_height > 5",
  date_range = c(Sys.Date() - 30, Sys.Date())
)

## End(Not run)
```

---

read_predictions *Read and reconcile predictions from JSONL*

Description

Reads a project's prediction JSONL file and reconciles outcomes. When multiple records share the same `prediction_id`, the latest non-null outcome wins (allows appending outcome updates).

Usage

```
read_predictions(project_slug = NULL)
```

Arguments

project_slug	Character project slug. If NULL, reads all files in `~/.claude/predictions/`.

Value

Tibble of predictions with one row per unique prediction_id

Examples

```
## Not run:
preds <- read_predictions("my-project-slug")
head(preds)

## End(Not run)
```

rogue_wave_report *Get Rogue Wave Summary Report*

Description

Generates a formatted summary report of rogue wave analysis.

Usage

```
rogue_wave_report(con, days = 30)
```

Arguments

con	DBI connection to DuckDB database
days	Number of days to analyze (default: 30)

Value

Character string with formatted report

run_api *Run the irishbuoys REST API*

Description

Starts a plumber API server that serves pre-computed JSON files from `docs/api/v1/`. Requires the `plumber` package.

Usage

```
run_api(port = 8080, host = "0.0.0.0")
```

Arguments

port	Integer port number (default: 8080)
host	Character host address (default: "0.0.0.0")

Value

Invisibly returns the plumber router (runs until interrupted).

See Also

Other api: `api_plumber`, `api_static`, `create_api_router()`, `generate_api_decomposition()`, `generate_api_extremes()`, `generate_api_gust_factors()`, `generate_api_index()`, `generate_api_latest()`, `generate_api_methods()`, `generate_api_sources()`, `generate_api_spatial()`, `generate_api_status()`, `generate_api_trends()`

Examples

```
## Not run:
run_api()
# API available at http://localhost:8080
# Swagger docs at http://localhost:8080/__docs__/

## End(Not run)
```

`save_to_parquet` *Save Data to Parquet with Optimal Compression*

Description

Save Data to Parquet with Optimal Compression

Usage

```
save_to_parquet(
  data,
  data_path = "inst/extdata/parquet",
  partition_by = "year_month",
  compression = "zstd"
)
```

Arguments

`data`	Data frame to save
`data_path`	Base path for Parquet files
`partition_by`	How to partition: "year_month", "station", or "both"
`compression`	Compression algorithm: "snappy", "gzip", "zstd", "lz4"

`send_storm_alert` *Send Storm Alert Email*

Description

Main orchestrator: fetches forecasts, detects storms, and sends an email alert if strong gale winds (Beaufort 9+) are forecast. If no storms are detected, no email is sent. Uses the same Gmail SMTP pattern as the weekly email report.

Usage

```
send_storm_alert(
  threshold_knots = NULL,
  recipient = Sys.getenv("GMAIL_USERNAME"),
  sender = Sys.getenv("GMAIL_USERNAME"),
  dry_run = FALSE
)
```

Arguments

threshold_knots	Numeric threshold in knots (default NULL, uses env var or 41).
recipient	Email recipient (default from GMAIL_USERNAME env var).
sender	Email sender (default from GMAIL_USERNAME env var).
dry_run	Logical; if TRUE, saves HTML preview to tempdir instead of sending.

Value

List with: status ("sent", "no_storms", "preview", "error"), n_storms, stations_affected, preview_file (if dry_run), error (if failed).

See Also

Other storm-alert: beaufort_to_description(), create_storm_alert_email(), detect_storm_events(), fetch_all_forecasts(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), summarise_forecast_rogue_risk()

Examples

```
## Not run:
# Check with very high threshold (likely no storms)
send_storm_alert(threshold_knots = 999)

# Dry run with low threshold (likely produces alert)
send_storm_alert(threshold_knots = 20, dry_run = TRUE)

## End(Not run)
```

---

station_distance_matrix

Calculate Distance Matrix Between All Stations

Description

Creates a matrix of distances between all station pairs.

Usage

```
station_distance_matrix(station_info = NULL)
```

Arguments

station_info	Data frame from get_station_info() or NULL to use default

Value

Named matrix of distances in km

Examples

```
station_distance_matrix()
```

---

summarise_forecast_rogue_risk

Summarise Forecast Rogue-Wave Risk per Station

Description

Given an Open-Meteo marine forecast tibble, computes per-station summaries: the peak forecast hour, peak H_s, peak P(H_max > 20 m), peak P(H_max > 25 m). Uses p_hmax_exceedance() applied independently per forecast hour, then takes the maximum across the forecast horizon.

This is a *forecast-derived* risk surrogate — it should always be presented alongside the deterministic-NWP caveat (lead-time skill drops sharply after day 2-3, no ensemble spread).

Usage

```
summarise_forecast_rogue_risk(
  marine_forecasts,
  thresholds = c(10, 15, 20, 25),
  duration_s = 3600
)
```

Arguments

marine_forecasts	Tibble from fetch_all_marine_forecasts().
thresholds	Numeric vector of H_max thresholds in metres (default c(20, 25)).
duration_s	Window duration in seconds for each forecast hour (default 3600).

Value

Tibble with one row per station: station_id, peak_time, peak_hs_m, peak_period_s, p_hmax_gt_10, p_hmax_gt_15, p_hmax_gt_20, p_hmax_gt_25, n_forecast_hours. Empty tibble if marine_forecasts is empty.

See Also

Other storm-alert: beaufort_to_description(), create_storm_alert_email(), detect_storm_events(), fetch_all_forecasts(), fetch_all_marine_forecasts(), fetch_met_eireann_warnings(), fetch_open_meteo_forecast(), fetch_open_meteo_marine(), knots_to_beaufort(), p_hmax_exceedance(), send_storm_alert()

---

test_rogue_propagation

Test Spatial Propagation of Rogue Wave Events

Description

Tests whether rogue wave events at one station are followed by rogue events at another station within a time window consistent with wave propagation. Uses a permutation test: the null hypothesis is that rogue events at the second station are uniformly distributed over time (no clustering with the first station).

For each station pair, the theoretical propagation lag is estimated as `distance_km / propagation_speed_kmh` (default 30 km/h for deep-water swell group velocity). Co-occurrence is counted when a station-2 rogue event falls within `[lag - tolerance, lag + tolerance]` hours of a station-1 event.

Usage

```
test_rogue_propagation(
  data,
  rogue_threshold = 2,
  min_wave_height = 2,
  station_pairs = NULL,
  propagation_speed_kmh = 30,
  n_permutations = 500,
  station_info = NULL
)
```

Arguments

`data`	Data frame with columns: `time` (POSIXct), `station_id` (character), `wave_height` (numeric), `hmax` (numeric).
`rogue_threshold`	Hmax/Hs ratio threshold for rogue classification (default: 2.0).
`min_wave_height`	Minimum significant wave height in metres for a qualifying observation (default: 2.0).
`station_pairs`	Optional list of character vectors, each of length 2, specifying directed pairs `c(source, receiver)`. If `NULL`, uses default focus pairs: M6->M2, M6->M3, M6->M5, M2->M3, M3->M5.
`propagation_speed_kmh`	Assumed deep-water group velocity in km/h (default: 30).

`n_permutations` Number of permutations for the test (default: 500).

`station_info` Optional data frame from `get_station_info()`. If `NULL`, uses the default 5-station network.

Value

List with:

h3_table Data frame with columns: `station1`, `station2`, `distance_km`, `theoretical_lag_hrs`, `n_rogue_s1`, `n_rogue_s2`, `co_occurrence_count`, `co_occurrence_rate`, `marginal_rate`, `perm_mean_rate`, `p_value`, `h3_significant` (logical), `h3_verdict`.

rogue_events Data frame of all detected rogue wave events.

n_rogue_total Total number of rogue events across all stations.

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, variables = c("time", "station_id", "wave_height", "hmax"))
result <- test_rogue_propagation(data)
result$h3_table
DBI::dbDisconnect(con)

## End(Not run)
```

`train_wave_model` *Train Wave Height Prediction Model*

Description

Trains a Random Forest model using ranger to predict wave height.

Usage

```
train_wave_model(
  data,
  target = "wave_height",
  predictors = NULL,
  train_fraction = 0.7,
  seed = 42,
  ...
)
```

Arguments

data	Data frame with prepared features (from prepare_wave_features)
target	Target variable name (default: "wave_height")
predictors	Character vector of predictor names (default: NULL uses standard set)
train_fraction	Fraction of data for training (default: 0.7)
seed	Random seed for reproducibility (default: 42)
...	Additional arguments passed to ranger::ranger

Value

List with model, train/test indices, and feature importance

Examples

```
## Not run:
con <- connect_duckdb()
data <- query_buoy_data(con, qc_filter = FALSE)
features <- prepare_wave_features(data)
model_result <- train_wave_model(features)
DBI::dbDisconnect(con)

## End(Not run)
```

---

trend_summary_report *Create Trend Summary Report*

Description

Generates a formatted summary of trend analysis results.

Usage

```
trend_summary_report(seasonal_means, annual_trends, anomalies = NULL)
```

Arguments

seasonal_means	Result from calculate_seasonal_means
annual_trends	Result from calculate_annual_trends
anomalies	Result from detect_anomalies (optional)

Value

Character string with formatted report

Examples

```
## Not run:
data <- data.frame(
  time = seq(as.POSIXct("2015-01-01"), by = "hour", length.out = 5000),
  wave_height = 2 + sin(seq(0, 40, length.out = 5000)) + rnorm(5000, 0, 0.3)
)
seasonal <- calculate_seasonal_means(data)
annual <- calculate_annual_trends(data)
trend_summary_report(seasonal, annual)

## End(Not run)
```

`validate_buoy_data` *Validate analysis data with pointblank*

Description

Performs comprehensive validation of the analysis_data target using pointblank's interrogation framework. Checks for:

- Minimum row count
- Required columns exist
- No NULL values in key columns
- Value ranges for physical measurements
- Valid station IDs

Usage

```
validate_buoy_data(
  data,
  target_name = "analysis_data",
  min_rows = 100,
  report_path = NULL
)
```

Arguments

`data`	A data frame or tibble to validate
`target_name`	Name of the target for error messages (default: "analysis_data")
`min_rows`	Minimum expected rows (default: 100)
`report_path`	Optional path to save HTML validation report

Value

The original data if validation passes, otherwise aborts with error

Examples

```
## Not run:
# Basic validation
validated_data <- validate_buoy_data(my_data)

# With custom settings and report
validated_data <- validate_buoy_data(
  my_data,
  target_name = "custom_target",
  min_rows = 1000,
  report_path = "validation_report.html"
)

## End(Not run)
```

`validate_email_freshness` *Validate Email Data Freshness*

Description

Checks that the latest observation timestamps in ingestion_stats are within an acceptable window of the current time.

Usage

```
validate_email_freshness(ingestion_stats, max_stale_hours = 96)
```

Arguments

`ingestion_stats`	Tibble with `station_id` and `latest` columns
`max_stale_hours`	Maximum acceptable age of data in hours (default: 96)

Value

ingestion_stats (invisibly), or aborts if ALL stations are stale

Examples

```
stats <- tibble::tibble(
  station_id = c("M2", "M3"),
  latest = Sys.time() - c(1, 2) * 3600
)
validate_email_freshness(stats)
```

validate_rogue_events *Validate rogue wave events data*

Description

Validates rogue wave detection results with specific checks for the rogue_ratio column and event characteristics.

Usage

```
validate_rogue_events(
  data,
  target_name = "rogue_wave_events",
  min_rows = 1,
  report_path = NULL
)
```

Arguments

data	A data frame of rogue wave events
target_name	Name of the target for error messages
min_rows	Minimum expected rows (default: 1)
report_path	Optional path to save HTML validation report

Value

The original data if validation passes

wave_glossary *Glossary of Wave Measurement Terms*

Description

Returns a data frame of acronyms and definitions used in wave measurement.

Usage

```
wave_glossary()
```

Value

Data frame with columns: acronym, term, definition, unit

Examples

```
glossary <- wave_glossary()
print(glossary)
```

wave_model *Wave Height Prediction Model*

Description

Functions for building and using a Random Forest model to predict significant wave height from meteorological variables.

wave_model_report *Generate Wave Model Report*

Description

Creates a formatted summary report of the wave height prediction model.

Usage

```
wave_model_report(model_result, eval_result)
```

Arguments

model_result	Result from train_wave_model
eval_result	Result from evaluate_wave_model

Value

Character string with formatted report

wave_science_documentation *Generate Wave Science Documentation*

Description

Returns a comprehensive markdown document explaining wave measurement science, suitable for inclusion in vignettes.

Usage

```
wave_science_documentation()
```

Value

Character string with markdown-formatted documentation

Examples

```
docs <- wave_science_documentation()
names(docs)
```

`widen_ci` *Widen a Confidence Interval by an Obs-Confidence Factor*

Description

Inflates the half-width of an existing CI by `1 / confidence`. Useful when the underlying point estimate is from observations and you want the displayed band to grow as the data ages, *without* refitting the model.

This is a heuristic display device, not a proper Bayesian update. It preserves the median estimate and only stretches the band. Stretching is applied symmetrically about the point estimate.

Usage

```
widen_ci(point, lower, upper, confidence)
```

Arguments

`point`	Numeric vector of point estimates.
`lower`	Numeric vector of original CI lower bounds.
`upper`	Numeric vector of original CI upper bounds.
`confidence`	Numeric multiplier in `(0, 1]` (e.g. from `compute_obs_confidence()`). Vectorised — recycled to length of `point`.

Value

List with `lower` and `upper` numeric vectors, widened symmetrically about `point`.

See Also

Other obs-confidence: `compute_obs_confidence()`, `obs_status_label()`

Examples

```
widen_ci(point = 10, lower = 8, upper = 12, confidence = 0.5)
```

Index