Package: measure (via r-universe)

October 1, 2026

Title A Recipes-Style Interface to Tidymodels for Analytical Measurements

Version 0.0.1.9002

Description Provides preprocessing steps for analytical measurement data such as spectroscopy and chromatography within the 'tidymodels' framework. Extends 'recipes' with steps for common spectral preprocessing techniques.

License MIT + file LICENSE

URL https://jameshwade.github.io/measure/, https://github.com/JamesHWade/measure

BugReports https://github.com/JamesHWade/measure/issues

Depends R (>= 4.1.0), recipes

Imports cli, dials, dplyr, generics, ggplot2, glue, IDPmisc, lifecycle, MASS, pracma, purrr, rlang, scales, tibble, tidyr, tidyselect, vctrs

Suggests covr, dtw, knitr, mcr, modeldata, multiway, parsnip, patchwork, prospectr, ptw, quarto, reticulate, rmarkdown, roxygen2, testthat (>= 3.0.0), tidymodels, tidyverse, wavethresh, withr, workflows

VignetteBuilder knitr

Config/reticulate list(packages = list(list(package = ``pybaselines'')))

Config/testthat/edition 3

Encoding UTF-8

LazyData true

LazyDataCompression xz

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Config/pak/sysreqs libicu-dev

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

Date/Publication 2026-04-04 17:53:34 UTC

RemoteUrl https://github.com/JamesHWade/measure

RemoteRef main

RemoteSha aac4cf0c6b99b8c7ce868fb43f30478fd663bf25

Contents

add_param_jitter . . . 7
add_validation_section . . . 8
align_max_shift . . . 9
all_pass . . . 9
assess_deconv_quality . . . 10
augment.measure_calibration . . . 11
autoplot-measure . . . 12
autoplot.measure_bland_altman . . . 13
autoplot.measure_calibration . . . 14
autoplot.measure_control_chart . . . 15
autoplot.measure_deming_regression . . . 15
autoplot.measure_linearity . . . 16
autoplot.measure_matrix_effect . . . 16
autoplot.measure_proficiency_score . . . 17
autoplot.measure_uncertainty_budget . . . 17
baseline_lambda . . . 18
bigaussian_peak_model . . . 19
bin_width . . . 20
check_axis_consistency . . . 20
check_measure_recipe . . . 21
check_quality_gates . . . 23
create_peak_model . . . 24
criteria_presets . . . 24
criterion . . . 26
derivative_order . . . 27
emg_peak_model . . . 28
find_measure_cols . . . 29
find_measure_nd_cols . . . 29
find_peaks_cols . . . 30
fortify-measure . . . 30
fortify.measure_calibration . . . 31
gaussian_peak_model . . . 32
get_failures . . . 33
get_measure_col_ndim . . . 33
get_peak_algorithm . . . 34
get_validation_section . . . 35
glance.measure_calibration . . . 35
glucose_bioreactors . . . 36
has_measure_col . . . 37

has_peak_algorithm . . . 38
has_peak_model . . . 38
has_validation_section . . . 39
hplc_chromatograms . . . 39
infer_axis_type . . . 41
initialize_peak_params . . . 42
is_measure_calibration . . . 43
is_measure_list . . . 43
is_measure_nd_list . . . 44
is_measure_nd_tbl . . . 45
is_measure_tbl . . . 45
is_peak_model . . . 46
is_peaks_list . . . 46
lorentzian_peak_model . . . 47
maldi_spectra . . . 48
measure_accuracy . . . 49
measure_apply . . . 51
measure_assess . . . 52
measure_axis_info . . . 54
measure_bland_altman . . . 55
measure_calibration . . . 56
measure_calibration_fit . . . 57
measure_calibration_predict . . . 59
measure_calibration_verify . . . 61
measure_carryover . . . 63
measure_column_patterns . . . 64
measure_column_summary . . . 65
measure_control_chart . . . 66
measure_control_limits . . . 67
measure_criteria . . . 69
measure_deming_regression . . . 70
measure_detect_drift . . . 72
measure_dim_names . . . 73
measure_dim_units . . . 74
measure_fold . . . 75
measure_gage_rr . . . 76
measure_grid_info . . . 77
measure_identify_columns . . . 78
measure_intermediate_precision . . . 80
measure_is_regular . . . 81
measure_linearity . . . 82
measure_lod . . . 84
measure_lod_loq . . . 86
measure_loq . . . 87
measure_map . . . 89
measure_map_safely . . . 91
measure_matrix_effect . . . 92
measure_ndim . . . 95

measure_packs . . . 95
measure_passing_bablok . . . 96
measure_plot_summary . . . 98
measure_proficiency_score . . . 99
measure_project . . . 101
measure_quality_summary . . . 102
measure_repeatability . . . 102
measure_reproducibility . . . 104
measure_sample_types . . . 105
measure_slice . . . 105
measure_standardize_sample_type . . . 106
measure_steps . . . 108
measure_summarize . . . 109
measure_system_suitability . . . 110
measure_uncertainty . . . 112
measure_uncertainty_budget . . . 113
measure_unfold . . . 114
measure_validate_metadata . . . 116
measure_validation_report . . . 118
meats_long . . . 122
new_measure_list . . . 123
new_measure_nd_list . . . 124
new_measure_nd_tbl . . . 125
new_measure_tbl . . . 126
new_peak_model . . . 126
optimize_deconvolution . . . 128
outlier_threshold . . . 129
peak_algorithms . . . 130
peak_location_min . . . 131
peak_model_area . . . 132
peak_model_bounds . . . 133
peak_model_gradient . . . 134
peak_model_gradient_numerical . . . 134
peak_model_initial_guess . . . 135
peak_model_param_names . . . 136
peak_model_value . . . 136
peak_models . . . 137
plot_measure_comparison . . . 138
print.measure_validation_report . . . 139
register_measure_pack . . . 140
register_measure_step . . . 141
register_peak_algorithm . . . 142
register_peak_model . . . 143
render_validation_report . . . 144
sec_calibration . . . 146
sec_chromatograms . . . 147
set_measure_roles . . . 149
smooth_window . . . 150

step_measure_absorbance . . . 151
step_measure_align_cow . . . 153
step_measure_align_dtw . . . 154
step_measure_align_ptw . . . 156
step_measure_align_reference . . . 157
step_measure_align_shift . . . 159
step_measure_augment_noise . . . 160
step_measure_augment_scale . . . 162
step_measure_augment_shift . . . 163
step_measure_baseline_airpls . . . 164
step_measure_baseline_als . . . 166
step_measure_baseline_arpls . . . 168
step_measure_baseline_aspls . . . 170
step_measure_baseline_auto . . . 172
step_measure_baseline_custom . . . 173
step_measure_baseline_fastchrom . . . 176
step_measure_baseline_gpc . . . 177
step_measure_baseline_iarpls . . . 180
step_measure_baseline_minima . . . 181
step_measure_baseline_morph . . . 183
step_measure_baseline_morphological . . . 184
step_measure_baseline_poly . . . 186
step_measure_baseline_py . . . 188
step_measure_baseline_rf . . . 191
step_measure_baseline_rolling . . . 193
step_measure_baseline_snip . . . 194
step_measure_baseline_tophat . . . 196
step_measure_batch_reference . . . 198
step_measure_bin . . . 200
step_measure_calibrate_x . . . 201
step_measure_calibrate_y . . . 204
step_measure_center . . . 206
step_measure_channel_align . . . 208
step_measure_channel_combine . . . 210
step_measure_channel_ratio . . . 212
step_measure_derivative . . . 214
step_measure_derivative_gap . . . 215
step_measure_despike . . . 217
step_measure_detrend . . . 219
step_measure_dilution_correct . . . 221
step_measure_drift_linear . . . 223
step_measure_drift_qc_loess . . . 225
step_measure_drift_spline . . . 227
step_measure_emsc . . . 229
step_measure_exclude . . . 231
step_measure_filter_fourier . . . 232
step_measure_impute . . . 234
step_measure_input_long . . . 235

step_measure_input_wide . . . 238
step_measure_integrals . . . 240
step_measure_interpolate . . . 242
step_measure_kubelka_munk . . . 243
step_measure_log . . . 245
step_measure_map . . . 246
step_measure_mcr_als . . . 249
step_measure_moments . . . 251
step_measure_msc . . . 253
step_measure_mw_averages . . . 255
step_measure_mw_distribution . . . 257
step_measure_mw_fractions . . . 259
step_measure_normalize_auc . . . 260
step_measure_normalize_istd . . . 262
step_measure_normalize_max . . . 264
step_measure_normalize_peak . . . 266
step_measure_normalize_range . . . 268
step_measure_normalize_sum . . . 269
step_measure_normalize_vector . . . 271
step_measure_osc . . . 272
step_measure_output_long . . . 274
step_measure_output_wide . . . 276
step_measure_parafac . . . 278
step_measure_peaks_deconvolve . . . 280
step_measure_peaks_detect . . . 282
step_measure_peaks_filter . . . 285
step_measure_peaks_integrate . . . 286
step_measure_peaks_properties . . . 288
step_measure_peaks_to_table . . . 289
step_measure_qc_bracket . . . 291
step_measure_qc_outlier . . . 293
step_measure_qc_saturated . . . 295
step_measure_qc_snr . . . 296
step_measure_ratio_reference . . . 298
step_measure_ratios . . . 300
step_measure_resample . . . 301
step_measure_savitzky_golay . . . 303
step_measure_scale_auto . . . 305
step_measure_scale_pareto . . . 307
step_measure_scale_range . . . 309
step_measure_scale_vast . . . 311
step_measure_smooth_gaussian . . . 313
step_measure_smooth_ma . . . 314
step_measure_smooth_median . . . 316
step_measure_smooth_wavelet . . . 317
step_measure_snv . . . 319
step_measure_standard_addition . . . 321
step_measure_subtract_blank . . . 323

step_measure_subtract_reference . . . 326
step_measure_surrogate_recovery . . . 327
step_measure_transmittance . . . 330
step_measure_trim . . . 331
step_measure_tucker . . . 333
subtract_rf_baseline . . . 335
sum_peak_models . . . 336
summary.measure_validation_report . . . 336
tidy.measure_calibration . . . 337
tidy.measure_lod . . . 338
tidy.measure_uncertainty_budget . . . 339
tidy.measure_validation_report . . . 339
uncertainty_component . . . 340
uncertainty_type_a . . . 342
uncertainty_type_b_expanded . . . 343
uncertainty_type_b_rectangular . . . 344
unregister_peak_algorithm . . . 344
unregister_peak_model . . . 345
validate_measure . . . 345
validate_peak_model_params . . . 346
window_side . . . 347

Index **348**

`add_param_jitter` *Add Jitter to Parameters for Multi-Start Optimization*

Description

Perturbs initialized parameters to create diverse starting points for multi-start optimization strategies.

Usage

```
add_param_jitter(params_list, scale = 0.1, method = c("gaussian", "uniform"))
```

Arguments

`params_list`	List of parameter lists (one per peak).
`scale`	Jitter scale (fraction of parameter value).
`method`	Jitter method: `"gaussian"` or `"uniform"`.

Value

List of jittered parameter lists.

See Also

Other peak-deconvolution: `assess_deconv_quality()`, `check_quality_gates()`, `initialize_peak_params()`, `optimize_deconvolution()`

`add_validation_section` *Add or update a validation section*

Description

Add or update a validation section

Usage

```
add_validation_section(report, section, data)
```

Arguments

`report`	A `measure_validation_report` object.
`section`	Section name.
`data`	Section data to add.

Value

Updated `measure_validation_report` object.

Examples

```
report <- measure_validation_report(title = "Test Report")
# Add custom section
report <- add_validation_section(
  report,
  "custom_study",
  list(results = data.frame(x = 1:3, y = 4:6))
)
```

align_max_shift *Parameters for alignment steps*

Description

`align_max_shift()` controls the maximum shift allowed in alignment. `align_segment_length()` controls segment size for COW alignment.

Usage

```
align_max_shift(range = c(1L, 50L), trans = NULL)

align_segment_length(range = c(10L, 100L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
align_max_shift()
align_segment_length()
```

all_pass *Check if All Criteria Pass*

Description

A convenience function to check if all criteria in an assessment passed.

Usage

```
all_pass(assessment, na_pass = FALSE)
```

Arguments

`assessment`	A `measure_assessment` object from `measure_assess()`.
`na_pass`	Logical. Should NA results count as pass? Default is FALSE.

Value

Logical: TRUE if all criteria passed, FALSE otherwise.

Examples

```
crit <- measure_criteria(cv = 15, rsd = 20)
results <- list(cv = 10, rsd = 15)
assessment <- measure_assess(results, crit)
all_pass(assessment)
```

---

assess_deconv_quality

Assess Deconvolution Quality

Description

Calculates comprehensive quality metrics for a peak deconvolution fit, including goodness-of-fit statistics, information criteria, per-peak quality, and residual diagnostics.

Usage

```
assess_deconv_quality(x, y, result, models)
```

Arguments

`x`	Numeric vector of x-axis values.
`y`	Numeric vector of observed y-axis values.
`result`	Deconvolution result list from `optimize_deconvolution()`.
`models`	List of `peak_model` objects used in deconvolution.

Value

A list of class `deconv_quality` containing:

- `goodness_of_fit`: R-squared, RMSE, MAE, chi-squared
- `information_criteria`: AIC, BIC, AICc
- `peak_quality`: Per-peak purity, overlap, area
- `residual_analysis`: Autocorrelation, heteroscedasticity, normality tests
- `overall_grade`: Letter grade (A/B/C/D/F)
- `convergence_info`: Optimization convergence details

See Also

Other peak-deconvolution: `add_param_jitter()`, `check_quality_gates()`, `initialize_peak_params()`, `optimize_deconvolution()`

Examples

```
# Create synthetic data and fit
x <- seq(0, 20, by = 0.1)
true_y <- 1.5 * exp(-0.5 * ((x - 8) / 1)^2) +
  0.8 * exp(-0.5 * ((x - 12) / 1.5)^2)
y <- true_y + rnorm(length(x), sd = 0.05)

models <- list(gaussian_peak_model(), gaussian_peak_model())
init_params <- list(
  list(height = 1.2, center = 7.5, width = 1.2),
  list(height = 0.6, center = 12.5, width = 1.8)
)

result <- optimize_deconvolution(x, y, models, init_params)
quality <- assess_deconv_quality(x, y, result, models)
print(quality)
```

---

augment.measure_calibration

Augment Calibration Data

Description

Add fitted values and residuals to calibration data.

Usage

```
## S3 method for class 'measure_calibration'
augment(x, ...)
```

Arguments

x	A measure_calibration object.
...	Additional arguments (unused).

Value

A tibble with the original calibration data plus:

- `.fitted`: Fitted values
- `.resid`: Residuals
- `.std_resid`: Standardized residuals
- `.hat`: Leverage values
- `.cooksd`: Cook's distance

`autoplot-measure` *Autoplot Methods for Measure Objects*

Description

Create ggplot2 visualizations of spectral/chromatographic data stored in measure objects.

Usage

```
## S3 method for class 'measure_tbl'
autoplot(object, ...)

## S3 method for class 'measure_list'
autoplot(object, summary = FALSE, max_spectra = 50, alpha = 0.3, ...)

## S3 method for class 'recipe'
autoplot(object, n_samples = 10, which = c("before_after", "summary"), ...)
```

Arguments

`object`	A `measure_tbl`, `measure_list`, or `recipe` object.
`...`	Additional arguments passed to specific plot types.
`summary`	Logical. If TRUE, add mean +/- SD ribbon. Default FALSE.
`max_spectra`	Maximum number of individual spectra to plot. Default 50. Set to NULL for no limit.
`alpha`	Transparency for individual spectrum lines. Default 0.3.
`n_samples`	Number of samples to show in before/after comparison. Default 10.
`which`	Which comparison to show: `"before_after"` (default) shows side-by-side before/after comparison, `"summary"` shows summary statistics (mean +/- SD) for the processed data.

Details

For `measure_tbl` (single spectrum):

- Plots location vs value as a line

For `measure_list` (multiple spectra):

- Plots all spectra with optional summary ribbon
- Use `summary = TRUE` for mean +/- SD ribbon
- Use `max_spectra` to limit number of individual lines

For `recipe`:

- Shows before/after comparison of preprocessing
- Requires a prepped recipe
- Use `n_samples` to control number of samples shown

Value

A ggplot2 object.

Examples

```
## Not run:
library(ggplot2)

# Single spectrum
spec <- new_measure_tbl(location = 1:100, value = sin(1:100 / 10) + rnorm(100, sd = 0.1))
autoplot(spec)

# Multiple spectra with summary
rec <- recipe(water ~ ., data = meats_long) |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()
baked <- bake(rec, new_data = NULL)
autoplot(baked$.measures, summary = TRUE)

# Recipe before/after comparison
rec <- recipe(water ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_snv() |>
  prep()
autoplot(rec, n_samples = 10)

## End(Not run)
```

---

autoplot.measure_bland_altman

Plot Bland-Altman Analysis

Description

Creates a Bland-Altman plot showing differences vs means with limits of agreement.

Usage

```
## S3 method for class 'measure_bland_altman'
autoplot(object, show_loa = TRUE, show_ci = FALSE, ...)
```

Arguments

`object`	A `measure_bland_altman` object.
`show_loa`	Show limits of agreement? Default TRUE.
`show_ci`	Show confidence intervals for LOA? Default FALSE.
`...`	Additional arguments (unused).

Value

A ggplot object.

`autoplot.measure_calibration`
Plot Calibration Curve Diagnostics

Description

Creates diagnostic plots for a calibration curve using ggplot2.

Usage

```
## S3 method for class 'measure_calibration'
autoplot(object, type = c("curve", "residuals", "qq", "all"), ...)
```

Arguments

`object`	A measure_calibration object.
`type`	Type of plot: • `"curve"` (default): Calibration curve with data points • `"residuals"`: Residuals vs concentration • `"qq"`: Normal Q-Q plot of residuals • `"all"`: All diagnostic plots combined
`...`	Additional arguments passed to ggplot2 functions.

Value

A ggplot object.

Examples

```
library(ggplot2)
data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3)
)
cal <- measure_calibration_fit(data, response ~ nominal_conc)
autoplot(cal)
autoplot(cal, type = "residuals")
```

autoplot.measure_control_chart
Plot Control Chart

Description

Creates a control chart visualization showing data points, control limits, and any rule violations.

Usage

```
## S3 method for class 'measure_control_chart'
autoplot(object, ...)
```

Arguments

object	A measure_control_chart object.
...	Additional arguments (unused).

Value

A ggplot object showing the control chart.

autoplot.measure_deming_regression
Plot Method Comparison Regression

Description

Creates a scatter plot with regression line for method comparison.

Usage

```
## S3 method for class 'measure_deming_regression'
autoplot(object, show_identity = TRUE, ...)

## S3 method for class 'measure_passing_bablok'
autoplot(object, show_identity = TRUE, ...)
```

Arguments

object	A measure_deming_regression or measure_passing_bablok object.
show_identity	Show y = x identity line? Default TRUE.
...	Additional arguments (unused).

Value

A ggplot object.

`autoplot.measure_linearity` *Plot Linearity Assessment Results*

Description

Creates diagnostic plots for linearity assessment.

Usage

```
## S3 method for class 'measure_linearity'
autoplot(object, type = c("fit", "residuals"), ...)
```

Arguments

`object`	A linearity assessment result from `measure_linearity()`.
`type`	Type of plot: `"fit"` for fitted vs actual, or `"residuals"`.
`...`	Additional arguments (unused).

Value

A ggplot object.

`autoplot.measure_matrix_effect` *Plot Matrix Effects*

Description

Creates a visualization of matrix effects showing suppression/enhancement.

Usage

```
## S3 method for class 'measure_matrix_effect'
autoplot(object, type = c("bar", "point", "forest"), show_limits = TRUE, ...)
```

Arguments

`object`	A `measure_matrix_effect` object.
`type`	Plot type: ”bar”, ”point”, or ”forest”. Default ”bar”.
`show_limits`	Show acceptable limits (80-120%)? Default TRUE.
`...`	Additional arguments (unused).

Value

A ggplot object.

autoplot.measure_proficiency_score

Plot Proficiency Test Scores

Description

Creates a bar chart or dot plot of proficiency scores with threshold lines.

Usage

```
## S3 method for class 'measure_proficiency_score'
autoplot(object, type = c("bar", "point"), ...)
```

Arguments

`object`	A `measure_proficiency_score` object.
`type`	Plot type: "bar" or "point". Default "bar".
`...`	Additional arguments (unused).

Value

A ggplot object.

autoplot.measure_uncertainty_budget

Plot Uncertainty Budget

Description

Creates a Pareto chart showing the relative contribution of each uncertainty component to the combined uncertainty.

Usage

```
## S3 method for class 'measure_uncertainty_budget'
autoplot(object, ...)
```

Arguments

`object`	A measure_uncertainty_budget object.
`...`	Additional arguments (unused).

Value

A ggplot object showing the Pareto chart.

Examples

```
library(ggplot2)
u1 <- uncertainty_component("Repeatability", 0.05, type = "A", df = 9)
u2 <- uncertainty_component("Calibrator", 0.02, type = "B")
u3 <- uncertainty_component("Temperature", 0.03, type = "B")
budget <- measure_uncertainty_budget(u1, u2, u3)

autoplot(budget)
```

---

baseline_lambda *Parameters for baseline correction steps*

Description

`baseline_lambda()` controls the smoothness penalty in ALS baseline correction. `baseline_asymmetry()` controls the asymmetry parameter in ALS. `baseline_degree()` controls the polynomial degree for baseline fitting.

Usage

```
baseline_lambda(range = c(2, 9), trans = scales::transform_log10())

baseline_asymmetry(range = c(0.001, 0.1), trans = NULL)

baseline_degree(range = c(1L, 6L), trans = NULL)

baseline_half_window(range = c(5L, 100L), trans = NULL)

baseline_span(range = c(0.1, 0.9), trans = NULL)

baseline_alpha(range = c(0, 1), trans = NULL)

baseline_window(range = c(10L, 200L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
baseline_lambda()
baseline_asymmetry()
baseline_degree()
baseline_span()
```

`bigaussian_peak_model` *Create Bi-Gaussian Peak Model*

Description

Creates a Bi-Gaussian peak model with four parameters: height, center, width_left, and width_right.

Usage

```
bigaussian_peak_model()
```

Details

The Bi-Gaussian function uses different widths on the left and right sides of the peak, providing flexible asymmetry.

Value

A `bigaussian_peak_model` object.

See Also

Other peak-models: `emg_peak_model()`, `gaussian_peak_model()`, `lorentzian_peak_model()`

Examples

```
model <- bigaussian_peak_model()
x <- seq(0, 10, by = 0.1)
params <- list(height = 1, center = 5, width_left = 0.8, width_right = 1.2)
y <- peak_model_value(model, x, params)
plot(x, y, type = "l")
```

bin_width *Parameters for feature engineering and scatter correction*

Description

`bin_width()` controls the width of bins in spectral binning. `emsc_degree()` controls the polynomial degree for EMSC correction. `osc_n_components()` controls the number of orthogonal components in OSC.

Usage

```
bin_width(range = c(1, 20), trans = NULL)

emsc_degree(range = c(0L, 4L), trans = NULL)

osc_n_components(range = c(1L, 10L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
bin_width()
emsc_degree()
osc_n_components()
```

check_axis_consistency *Check axis consistency across samples*

Description

Validates that all samples in a `measure_list` have consistent axes (same locations). This is important for matrix operations that assume aligned data.

Usage

```
check_axis_consistency(
  x,
  tolerance = 1e-10,
  action = c("error", "warn", "message")
)
```

Arguments

x	A measure_list or data frame with measure column.
tolerance	Numeric tolerance for location comparison. Default is 1e-10.
action	What to do when validation fails: "error" (default), "warn", or "message".

Value

Invisibly returns a list with:

- consistent: Logical indicating if axes are consistent
- reference_locations: The reference locations (from first sample)
- inconsistent_samples: Indices of samples with different axes
- max_deviation: Maximum deviation from reference locations

Examples

```
# Consistent axes
specs <- new_measure_list(list(
  new_measure_tbl(location = 1:10, value = rnorm(10)),
  new_measure_tbl(location = 1:10, value = rnorm(10))
))
check_axis_consistency(specs)

# Inconsistent axes
specs_bad <- new_measure_list(list(
  new_measure_tbl(location = 1:10, value = rnorm(10)),
  new_measure_tbl(location = 1:11, value = rnorm(11))
))
try(check_axis_consistency(specs_bad))
```

---

check_measure_recipe *Check Measure Recipe Structure*

Description

Validates that a recipe is properly structured for measure operations. Checks for common issues like missing input steps, incompatible column types, and role conflicts.

Usage

```
check_measure_recipe(recipe, strict = TRUE)
```

Arguments

recipe	A recipe object to validate.
strict	Logical. If TRUE (default), returns errors as a tibble. If FALSE, issues cli warnings and returns the recipe invisibly.

Details

The following checks are performed:

Errors (will cause failures):

- No input step (`step_measure_input_*`)
- Output step before input step
- Multiple input steps

Warnings (may cause issues):

- No output step (data stays in internal format)
- Processing steps after output step
- No predictor columns identified

Info (suggestions):

- Large number of measurement columns (consider dimension reduction)
- No ID column identified

Value

If `strict = TRUE`, returns a tibble with columns:

level Severity: "error", "warning", or "info"

check Name of the check that triggered the message

message Description of the issue

If `strict = FALSE`, returns the recipe invisibly after printing warnings.

Examples

```
## Not run:
library(recipes)

# Check a properly structured recipe
rec <- recipe(outcome ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_snv() |>
  step_measure_output_wide()
```

```
check_measure_recipe(rec)

# Check a recipe with issues
bad_rec <- recipe(outcome ~ ., data = my_data) |>
  step_measure_snv()  # Missing input step!

check_measure_recipe(bad_rec)

## End(Not run)
```

`check_quality_gates` *Check if Fit Passes Quality Gates*

Description

Evaluates a deconvolution quality assessment against configurable thresholds to determine if the fit is acceptable.

Usage

```
check_quality_gates(quality, reject_threshold = 0.85, warn_threshold = 0.95)
```

Arguments

`quality`	A `deconv_quality` object from `assess_deconv_quality()`.
`reject_threshold`	Minimum R-squared to accept (default 0.85).
`warn_threshold`	R-squared threshold for warning (default 0.95).

Value

A list with:

- `status`: `"pass"`, `"warn"`, or `"reject"`
- `pass`, `warn`, `reject`: Logical flags
- `messages`: Character vector of issues found
- `grade`: Overall quality grade

See Also

Other peak-deconvolution: `add_param_jitter()`, `assess_deconv_quality()`, `initialize_peak_params()`, `optimize_deconvolution()`

create_peak_model *Create a Peak Model by Name*

Description

Creates a peak model object from a registered model name.

Usage

```
create_peak_model(name)
```

Arguments

name	Name of the model (e.g., ”gaussian”, ”emg”, ”bigaussian”).

Value

A peak_model object.

See Also

peak_models(), register_peak_model()

Examples

```
model <- create_peak_model("gaussian")
print(model)
```

criteria_presets *Preset Acceptance Criteria*

Description

Factory functions that return commonly-used criteria sets for analytical validation workflows.

Usage

```
criteria_bioanalytical(
  cv_qc = 15,
  cv_calibration = 20,
  r_squared = 0.99,
  recovery_range = c(80, 120),
  accuracy_bias = 15
)
```

```
criteria_ich_q2(
  cv_repeatability = 2,
  cv_intermediate = 5,
  recovery_range = c(98, 102),
  r_squared = 0.999
)

criteria_bland_altman(
  loa_width = NULL,
  bias_max = NULL,
  proportional_bias_p = 0.05
)

criteria_method_comparison(
  slope_range = c(0.9, 1.1),
  intercept_range = NULL,
  r_squared = 0.95
)

criteria_proficiency_testing(max_z_score = 2, pct_satisfactory = 100)

criteria_matrix_effects(me_range = c(80, 120), me_cv = 15)

criteria_surrogate_recovery(surrogate_recovery = c(70, 130))
```

Arguments

`cv_qc`	Maximum allowable CV for QC samples (default 15%, bioanalytical).
`cv_calibration`	Maximum allowable CV for calibration replicates (default 20%).
`r_squared`	Minimum R-squared for calibration curve.
`recovery_range`	Acceptable recovery range as c(lower, upper).
`accuracy_bias`	Maximum allowable bias (default 15%).
`cv_repeatability`	Maximum allowable CV for repeatability (default 2%, ICH Q2).
`cv_intermediate`	Maximum allowable CV for intermediate precision (default 5%, ICH Q2).
`loa_width`	Maximum acceptable limits of agreement width.
`bias_max`	Maximum acceptable mean bias.
`proportional_bias_p`	Significance level for proportional bias test.
`slope_range`	Acceptable range for regression slope (default c(0.9, 1.1)).
`intercept_range`	Acceptable range for regression intercept.
`max_z_score`	Maximum acceptable absolute z-score.

`pct_satisfactory`
: Minimum percentage of satisfactory results.

`me_range`
: Acceptable matrix effect range (default c(80, 120)).

`me_cv`
: Maximum acceptable CV of matrix effects.

`surrogate_recovery`
: Acceptable surrogate recovery range.

Value

A measure_criteria object.

Examples

```
# Default bioanalytical criteria
criteria_bioanalytical()

# Custom thresholds
criteria_bioanalytical(cv_qc = 20, r_squared = 0.98)
```

`criterion` *Create an Acceptance Criterion*

Description

Defines a single acceptance criterion for analytical validation. Criteria are used with `measure_assess()` to produce pass/fail decisions.

Usage

```
criterion(
  name,
  operator = c("<", "<=", ">", ">=", "==", "!=", "between", "outside"),
  threshold,
  description = NULL,
  priority = c("major", "critical", "minor")
)
```

Arguments

`name`
: Character string naming this criterion (e.g., "cv_qc", "r_squared").

`operator`
: Comparison operator: `"<"`, `"<="`, `">"`, `">="`, `"=="`, `"!="`, `"between"`, or `"outside"`.

`threshold`
: Numeric threshold value. For `"between"` and `"outside"`, provide a length-2 vector `c(lower, upper)`.

`description`
: Optional human-readable description of the criterion.

`priority`
: Optional priority level: `"critical"`, `"major"`, `"minor"`. Affects how failures are reported.

Value

A `measure_criterion` object.

See Also

`measure_criteria()` for combining multiple criteria, `measure_assess()` for evaluating criteria.

Examples

```
# QC coefficient of variation must be < 15%
criterion("cv_qc", "<", 15, description = "QC CV < 15%")

# R-squared must be >= 0.99
criterion("r_squared", ">=", 0.99)

# Recovery must be between 80% and 120%
criterion("recovery", "between", c(80, 120), priority = "critical")
```

`derivative_order` *Parameters for derivative steps*

Description

`derivative_order()` controls the order of differentiation in `step_measure_derivative()` (1 = first derivative, 2 = second derivative). `derivative_gap()` and `derivative_segment()` control the gap derivative (Norris-Williams) parameters in `step_measure_derivative_gap()`.

Usage

```
derivative_order(range = c(1L, 2L), trans = NULL)

derivative_gap(range = c(1L, 10L), trans = NULL)

derivative_segment(range = c(1L, 5L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
derivative_order()
derivative_gap()
derivative_segment()
```

`emg_peak_model` *Create EMG Peak Model*

Description

Creates an Exponentially Modified Gaussian peak model with four parameters: height, center, width (sigma), and tau (exponential decay constant).

Usage

```
emg_peak_model()
```

Details

The EMG function models asymmetric peaks with tailing, common in chromatography. It is the convolution of a Gaussian with an exponential decay function.

Value

An `emg_peak_model` object.

See Also

Other peak-models: `bigaussian_peak_model()`, `gaussian_peak_model()`, `lorentzian_peak_model()`

Examples

```
model <- emg_peak_model()
x <- seq(0, 15, by = 0.1)
params <- list(height = 1, center = 5, width = 0.5, tau = 0.3)
y <- peak_model_value(model, x, params)
plot(x, y, type = "l")
```

find_measure_cols *Find measure columns in a data frame*

Description

Finds all columns in a data frame that contain measurement data (i.e., are of class `measure_list`).

Usage

```
find_measure_cols(data)
```

Arguments

`data`	A data frame.

Value

Character vector of column names containing measure data. Returns empty character vector if no measure columns found.

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

result <- bake(rec, new_data = NULL)
find_measure_cols(result)  # ".measures"
```

find_measure_nd_cols *Find n-dimensional measure columns in a data frame*

Description

Returns the names of columns that contain `measure_nd_list` objects.

Usage

```
find_measure_nd_cols(data)
```

Arguments

`data`	A data frame.

Value

Character vector of column names.

Examples

```
# After using step_measure_input_long with multiple location columns
# find_measure_nd_cols(result)
```

---

find_peaks_cols *Find peaks columns in a data frame*

Description

Find peaks columns in a data frame

Usage

```
find_peaks_cols(data)
```

Arguments

data	A data frame.

Value

Character vector of column names.

---

fortify-measure *Convert Measure Objects to Data Frames for Plotting*

Description

These methods convert measure objects to data frames suitable for use with ggplot2.

Usage

```
## S3 method for class 'measure_tbl'
fortify(model, data = NULL, ...)

## S3 method for class 'measure_list'
fortify(model, data = NULL, ...)
```

Arguments

`model`	A `measure_tbl` or `measure_list` object.
`data`	Ignored. Present for compatibility with generic.
`...`	Additional arguments (currently unused).

Value

A tibble with columns `location` and `value` (for `measure_tbl`) or `location`, `value`, and `sample` (for `measure_list`).

Examples

```
## Not run:
library(ggplot2)

# Single spectrum
spec <- new_measure_tbl(location = 1:100, value = rnorm(100))
ggplot(fortify(spec), aes(location, value)) + geom_line()

# Multiple spectra (from recipe output)
rec <- recipe(water ~ ., data = meats_long) |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()
baked <- bake(rec, new_data = NULL)
ggplot(fortify(baked$.measures), aes(location, value, group = sample)) +
  geom_line(alpha = 0.5)

## End(Not run)
```

`fortify.measure_calibration` *Extract Calibration Curve Data*

Description

S3 method to extract the underlying data from a calibration object in a format suitable for ggplot2.

Usage

```
## S3 method for class 'measure_calibration'
fortify(model, data = NULL, ...)
```

Arguments

`model`	A measure_calibration object.
`data`	Ignored.
`...`	Additional arguments (unused).

Value

A data frame with the calibration data and fitted values/residuals.

`gaussian_peak_model` *Create Gaussian Peak Model*

Description

Creates a symmetric Gaussian peak model with three parameters: height, center, and width (sigma).

Usage

```
gaussian_peak_model()
```

Details

The Gaussian function is:

$$f(x) = h \cdot \exp\left(-\frac{(x-c)^2}{2\sigma^2}\right)$$

where `h` is height, `c` is center, and `sigma` is width.

Value

A `gaussian_peak_model` object.

See Also

Other peak-models: `bigaussian_peak_model()`, `emg_peak_model()`, `lorentzian_peak_model()`

Examples

```
model <- gaussian_peak_model()
x <- seq(0, 10, by = 0.1)
params <- list(height = 1, center = 5, width = 1)
y <- peak_model_value(model, x, params)
plot(x, y, type = "l")
```

get_failures *Extract Failed Criteria*

Description

Returns only the criteria that failed assessment.

Usage

```
get_failures(assessment)
```

Arguments

assessment	A measure_assessment object from measure_assess().

Value

A filtered measure_assessment tibble containing only failures.

Examples

```
crit <- measure_criteria(cv = 15, rsd = 20)
results <- list(cv = 18, rsd = 25)  # Both fail
assessment <- measure_assess(results, crit)
get_failures(assessment)
```

get_measure_col_ndim *Get the dimensionality of a measure column*

Description

Returns the number of dimensions (1 for measure_list, 2+ for measure_nd_list) of a measure column in a data frame.

Usage

```
get_measure_col_ndim(data, col)
```

Arguments

data	A data frame.
col	Character string naming the measure column.

Value

Integer indicating the number of dimensions.

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

result <- bake(rec, new_data = NULL)
get_measure_col_ndim(result, ".measures")  # 1
```

---

get_peak_algorithm *Get a Peak Detection Algorithm*

Description

Retrieves a registered peak detection algorithm by name.

Usage

```
get_peak_algorithm(name)
```

Arguments

`name`	Algorithm name.

Value

A list with components:

- `name`: Algorithm name
- `algorithm_fn`: The algorithm function
- `pack_name`: Source package name
- `description`: Brief description
- `default_params`: List of default parameter values
- `param_info`: List of parameter descriptions
- `technique`: Technique name (or `NULL`)

Returns `NULL` if algorithm not found.

See Also

`peak_algorithms()`, `register_peak_algorithm()`

Examples

```
algo <- get_peak_algorithm("prominence")
if (!is.null(algo)) {
  print(algo$description)
}
```

---

get_validation_section

Get validation section data

Description

Get validation section data

Usage

```
get_validation_section(report, section)
```

Arguments

report	A measure_validation_report object.
section	Section name to retrieve.

Value

The section data, or NULL if not found.

Examples

```
report <- measure_validation_report(title = "Test Report")
get_validation_section(report, "calibration")  # NULL
```

---

glance.measure_calibration

Glance at Calibration Curve Summary

Description

Extract one-row summary statistics from a calibration curve.

Usage

```
## S3 method for class 'measure_calibration'
glance(x, ...)
```

Arguments

x	A measure_calibration object.
...	Additional arguments (unused).

Value

A tibble with columns:

- `r_squared`: Coefficient of determination
- `adj_r_squared`: Adjusted R-squared
- `sigma`: Residual standard error
- `df`: Degrees of freedom
- `model_type`: Model type (linear/quadratic)
- `weights_type`: Weighting scheme
- `n_points`: Number of calibration points
- `n_outliers`: Number of flagged outliers

Examples

```
data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3)
)
cal <- measure_calibration_fit(data, response ~ nominal_conc)
glance(cal)
```

`glucose_bioreactors` *Raman Spectra Bioreactor Data*

Description

Kuhn and Johnson (2013) used these two data sets to model the glucose yeild in large- and small-scale bioreactors:

Details

- Fifteen small-scale (5 liters) bioreactors were seeded with cells and were monitored daily for 14 days.
- Three large-scale bioreactors were also seeded with cells from the same batch and monitored daily for 14 days.

Samples were collected each day from all bioreactors and glucose was measured. The goal would be to create models on the data from the more numerous small-scale bioreactors and then evaluate if these results can accurately predict what is happening in the large-scale bioreactors.

Value

Two tibbles. For each, there are 2,651 columns whose names are numbers and these are the measured assay values (and the names are the wave numbers). The numeric column `glucose` has the outcome data, `day` is the number of days in the bioreactor, the `batch_id` is the reactor identifier (with "L" for large and "S" for small), and `batch_sample` that is the ID and the day.

Source

Kuhn and Johnson (2020), *Feature Engineering and Selection*, Chapman and Hall/CRC . https://bookdown.org/max/FES/ and https://github.com/topepo/FES

Examples

```
data(glucose_bioreactors)
dim(bioreactors_small)
```

---

has_measure_col *Check if data frame has measure column(s)*

Description

Checks whether a data frame contains at least one measure column. This is the recommended way to validate data in step functions.

Usage

```
has_measure_col(data)
```

Arguments

`data`	A data frame.

Value

Invisibly returns the names of measure columns found. Throws an error if no measure columns are found.

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

result <- bake(rec, new_data = NULL)
```

```
has_measure_col(result)  # TRUE (returns invisibly)
```

`has_peak_algorithm` *Check if a Peak Algorithm Exists*

Description

Checks whether a peak detection algorithm is registered.

Usage

```
has_peak_algorithm(name)
```

Arguments

`name`	Algorithm name.

Value

Logical `TRUE` if algorithm exists, `FALSE` otherwise.

See Also

`peak_algorithms()`

Examples

```
has_peak_algorithm("prominence")  # TRUE
has_peak_algorithm("nonexistent") # FALSE
```

`has_peak_model` *Check if a Peak Model Exists*

Description

Check if a Peak Model Exists

Usage

```
has_peak_model(name)
```

Arguments

`name`	Model name.

Value

Logical `TRUE` if model exists.

has_validation_section *Check if validation report has a section*

Description

Check if validation report has a section

Usage

```
has_validation_section(report, section)
```

Arguments

report	A measure_validation_report object.
section	Section name to check.

Value

Logical indicating if section exists and has data.

Examples

```
report <- measure_validation_report(title = "Test Report")
has_validation_section(report, "calibration")  # FALSE
```

hplc_chromatograms *Simulated HPLC Chromatography Data*

Description

Simulated HPLC-UV chromatogram data for demonstration of chromatographic preprocessing and peak analysis. The dataset represents a separation of five phenolic compounds (caffeine, theobromine, catechin, epicatechin, and quercetin) with 20 samples of varying concentrations.

Format

A tibble with 30,020 observations and 8 variables:

sample_id Integer sample identifier (1-20)
time_min Retention time in minutes (0-15, 0.01 min resolution)
absorbance_mAU UV absorbance signal in milli-absorbance units
caffeine_conc True caffeine concentration (mg/L) for calibration
theobromine_conc True theobromine concentration (mg/L)
catechin_conc True catechin concentration (mg/L)
epicatechin_conc True epicatechin concentration (mg/L)
quercetin_conc True quercetin concentration (mg/L)

Details

The chromatograms include realistic features such as:

- Gaussian peak shapes with compound-specific widths
- Baseline drift
- Instrumental noise
- Small retention time variations between runs
- Concentration-dependent peak heights

This dataset is useful for demonstrating:

- Baseline correction methods
- Peak detection and integration
- Calibration curve construction
- Retention time alignment

The peaks appear at approximately these retention times:

- Caffeine: ~2.5 min
- Theobromine: ~4.2 min
- Catechin: ~6.8 min
- Epicatechin: ~9.1 min
- Quercetin: ~12.3 min

Source

Simulated data generated for the measure package. See `data-raw/generate_datasets.R` for the generation script.

See Also

`sec_chromatograms` for SEC/GPC chromatography data

Examples

```
data(hplc_chromatograms)

# View structure
str(hplc_chromatograms)

# Get a single chromatogram
library(dplyr)
chrom_1 <- hplc_chromatograms |> filter(sample_id == 1)

# Plot (if ggplot2 available)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(chrom_1, aes(x = time_min, y = absorbance_mAU)) +
```

```
    geom_line() +
    labs(x = "Retention Time (min)", y = "Absorbance (mAU)",
         title = "HPLC Chromatogram")
}
```

`infer_axis_type` *Infer axis type from location values*

Description

Attempts to infer the type of measurement axis based on the range and characteristics of location values. This is a heuristic that helps guide appropriate preprocessing choices.

Usage

```
infer_axis_type(location)
```

Arguments

`location`	Numeric vector of location values.

Value

Character string indicating inferred axis type:

- `"wavelength_nm"`: Visible/NIR wavelengths (typically 300-2500 nm)
- `"wavenumber"`: Mid-IR wavenumbers (typically 400-4000 cm^-1)
- `"retention_time"`: Chromatography retention time (typically 0-60 min)
- `"mass_charge"`: Mass spectrometry m/z (typically 50-2000+)
- `"ppm"`: NMR chemical shift (typically -2 to 14 ppm)
- `"two_theta"`: XRD diffraction angle (typically 5-90 degrees)
- `"temperature"`: Thermal analysis (typically 20-1000 C)
- `"unknown"`: Could not determine axis type

Examples

```
# NIR wavelengths
infer_axis_type(seq(1000, 2500, by = 2))

# Mid-IR wavenumbers
infer_axis_type(seq(4000, 400, by = -4))

# Retention time (minutes)
infer_axis_type(seq(0, 30, by = 0.01))

# NMR chemical shift
infer_axis_type(seq(0, 12, by = 0.001))
```

`initialize_peak_params` *Smart Parameter Initialization for Peak Deconvolution*

Description

Initializes peak model parameters using actual peak properties from the data rather than naive guesses, improving optimization convergence.

Usage

```
initialize_peak_params(
  x,
  y,
  n_peaks,
  models,
  peak_indices = NULL,
  smooth = TRUE,
  smooth_span = 0.05
)
```

Arguments

`x`	Numeric vector of x-axis values.
`y`	Numeric vector of y-axis values.
`n_peaks`	Number of peaks to initialize.
`models`	List of `peak_model` objects (one per peak).
`peak_indices`	Optional integer vector of peak indices (if already known).
`smooth`	Logical. If `TRUE`, smooth data before peak detection.
`smooth_span`	Smoothing span for LOESS (if `smooth = TRUE`).

Value

List of initialized parameter lists (one per peak).

See Also

Other peak-deconvolution: `add_param_jitter()`, `assess_deconv_quality()`, `check_quality_gates()`, `optimize_deconvolution()`

Examples

```
# Create synthetic data with two peaks
x <- seq(0, 20, by = 0.1)
y <- 1.5 * exp(-0.5 * ((x - 8) / 1)^2) +
  0.8 * exp(-0.5 * ((x - 12) / 1.5)^2)
```

```
models <- list(gaussian_peak_model(), gaussian_peak_model())
init_params <- initialize_peak_params(x, y, n_peaks = 2, models = models)
```

`is_measure_calibration` *Test if Object is a Calibration Curve*

Description

Test if Object is a Calibration Curve

Usage

```
is_measure_calibration(x)
```

Arguments

`x`	Object to test.

Value

Logical: TRUE if `x` is a `measure_calibration` object.

Examples

```
# After fitting a calibration curve
data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3)
)
cal <- measure_calibration_fit(data, response ~ nominal_conc)
is_measure_calibration(cal)
```

`is_measure_list` *Test if object is a measure list*

Description

Test if object is a measure list

Usage

```
is_measure_list(x)
```

Arguments

x	Object to test.

Value

Logical indicating if x inherits from measure_list.

Examples

```
# After using step_measure_input_*, the .measures column is a measure_list
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

result <- bake(rec, new_data = NULL)
is_measure_list(result$.measures)
```

---

is_measure_nd_list *Test if object is an n-dimensional measure list*

Description

Test if object is an n-dimensional measure list

Usage

```
is_measure_nd_list(x)
```

Arguments

x	Object to test.

Value

Logical indicating if x inherits from measure_nd_list.

Examples

```
# Create and test a measure_nd_list
meas1 <- new_measure_nd_tbl(
  location_1 = 1:5,
  location_2 = rep(1, 5),
  value = rnorm(5)
)
ml <- new_measure_nd_list(list(meas1))
is_measure_nd_list(ml)  # TRUE
```

is_measure_nd_tbl *Test if object is an n-dimensional measure tibble*

Description

Test if object is an n-dimensional measure tibble

Usage

```
is_measure_nd_tbl(x)
```

Arguments

x	Object to test.

Value

Logical indicating if `x` inherits from `measure_nd_tbl`.

Examples

```
# Create a 2D measure tibble
mt <- new_measure_nd_tbl(
  location_1 = 1:10,
  location_2 = rep(1:2, each = 5),
  value = rnorm(10)
)
is_measure_nd_tbl(mt)  # TRUE

# Regular tibbles are not measure_nd_tbl
is_measure_nd_tbl(tibble::tibble(x = 1:5))  # FALSE
```

is_measure_tbl *Test if object is a measure tibble*

Description

Test if object is a measure tibble

Usage

```
is_measure_tbl(x)
```

Arguments

x	Object to test.

Value

Logical indicating if `x` inherits from `measure_tbl`.

Examples

```
# Create a measure tibble
mt <- measure:::new_measure_tbl(location = 1:5, value = rnorm(5))
is_measure_tbl(mt)

# Regular tibbles are not measure tibbles
is_measure_tbl(tibble::tibble(location = 1:5, value = rnorm(5)))
```

---

is_peak_model *Test if Object is a Peak Model*

Description

Test if Object is a Peak Model

Usage

```
is_peak_model(x)
```

Arguments

`x`	Object to test.

Value

Logical indicating if `x` is a `peak_model`.

---

is_peaks_list *Test if object is a peaks list*

Description

Test if object is a peaks list

Usage

```
is_peaks_list(x)
```

Arguments

`x`	Object to test.

Value

Logical.

`lorentzian_peak_model` *Create Lorentzian Peak Model*

Description

Creates a Lorentzian (Cauchy) peak model with three parameters: height, center, and gamma (half-width at half-maximum).

Usage

```
lorentzian_peak_model()
```

Details

The Lorentzian function has heavier tails than Gaussian and is commonly used in spectroscopy.

Value

A `lorentzian_peak_model` object.

See Also

Other peak-models: `bigaussian_peak_model()`, `emg_peak_model()`, `gaussian_peak_model()`

Examples

```
model <- lorentzian_peak_model()
x <- seq(0, 10, by = 0.1)
params <- list(height = 1, center = 5, gamma = 0.5)
y <- peak_model_value(model, x, params)
plot(x, y, type = "l")
```

`maldi_spectra` *Simulated MALDI-TOF Mass Spectrometry Data*

Description

Simulated MALDI-TOF (Matrix-Assisted Laser Desorption/Ionization Time-of-Flight) mass spectrometry data for demonstration of mass spectral preprocessing. The dataset represents protein/peptide analysis from four experimental groups with four replicates each.

Format

A tibble with 304,016 observations and 5 variables:

sample_id Sample identifier combining group and replicate

group Experimental group ("Control", "Treatment_A", "Treatment_B", "Treatment_C")

replicate Replicate number (1-4)

mz Mass-to-charge ratio (m/z) in Daltons (1000-20000 Da)

intensity Signal intensity (arbitrary units)

Details

MALDI-TOF is a soft ionization technique commonly used for analyzing biomolecules such as proteins, peptides, and polymers. The technique provides mass-to-charge (m/z) ratios that can be used for identification and quantification.

The spectra include realistic features such as:

- Multiple peptide/protein peaks at different m/z values
- Baseline variation
- Chemical noise
- Peak width proportional to m/z (resolution effects)
- Replicate variation

This dataset is useful for demonstrating:

- Baseline correction methods
- Peak detection for mass spectra
- Normalization between samples
- Differential analysis between groups

Each group has a characteristic peak pattern:

- **Control**: Peptides at m/z ~1200, 1450, 1800, 2200, 3500, 5800, 8400, 12000
- **Treatment_A**: Peptides at m/z ~1100, 1650, 2100, 2800, 4200, 6500, 9200, 14000
- **Treatment_B**: Proteins at m/z ~2500, 4000, 5500, 8000, 11000, 15000, 18000
- **Treatment_C**: Peptides at m/z ~1050, 1280, 1520, 1890, 2340, 2980, 3650, 4500

The m/z resolution is approximately 500 ppm (parts per million), typical for linear MALDI-TOF instruments. Note that simulated spectra include baseline noise and minor peaks in addition to the characteristic peaks listed above.

Source

Simulated data generated for the measure package. See `data-raw/generate_datasets.R` for the generation script.

See Also

`hplc_chromatograms` for HPLC chromatography data `meats_long` for NIR spectroscopy data

Examples

```
data(maldi_spectra)

# View structure
str(maldi_spectra)

# Get unique samples
unique(maldi_spectra$sample_id)

# Get one spectrum
library(dplyr)
spec_1 <- maldi_spectra |> filter(sample_id == "Control_1")

# Plot (if ggplot2 available)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(spec_1, aes(x = mz, y = intensity)) +
    geom_line() +
    labs(x = "m/z (Da)", y = "Intensity",
         title = "MALDI-TOF Mass Spectrum")
}

# Compare groups
if (requireNamespace("ggplot2", quietly = TRUE)) {
  # Get one replicate per group
  comparison <- maldi_spectra |>
    filter(replicate == 1)

  ggplot(comparison, aes(x = mz, y = intensity, color = group)) +
    geom_line(alpha = 0.7) +
    facet_wrap(~group, ncol = 1) +
    labs(x = "m/z (Da)", y = "Intensity",
         title = "MALDI-TOF Spectra by Group")
}
```

---

measure_accuracy *Accuracy Assessment*

Description

Calculates accuracy metrics including bias, recovery, and confidence intervals for method validation.

Usage

```
measure_accuracy(
  data,
  measured_col,
  reference_col,
  group_col = NULL,
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing measured and reference values.
`measured_col`	Name of the column containing measured values.
`reference_col`	Name of the column containing reference/nominal values.
`group_col`	Optional grouping column (e.g., concentration level).
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Accuracy expresses the closeness of agreement between a measured value and a reference value. It is typically assessed using:

- **Bias**: Systematic difference from the reference value
- **Recovery**: Percentage of the reference value that is measured

ICH Q2 Requirements:

Accuracy should be assessed at a minimum of 3 concentration levels covering the specified range (typically 80-120% of the target).

Value

A `measure_accuracy` object containing:

- `n`: Number of observations
- `mean_measured`: Mean of measured values
- `mean_reference`: Mean of reference values
- `bias`: Absolute bias (measured - reference)
- `bias_pct`: Relative bias as percentage
- `recovery`: Recovery percentage (measured/reference * 100)
- `recovery_ci_lower`, `recovery_ci_upper`: Confidence interval for recovery

See Also

measure_linearity(), measure_carryover()

Other accuracy: measure_carryover(), measure_linearity()

Examples

```
# Accuracy at multiple levels
set.seed(123)
data <- data.frame(
  level = rep(c("low", "mid", "high"), each = 5),
  nominal = rep(c(10, 50, 100), each = 5),
  measured = c(
    rnorm(5, 10.2, 0.3),
    rnorm(5, 49.5, 1.5),
    rnorm(5, 101, 3)
  )
)

result <- measure_accuracy(data, "measured", "nominal", group_col = "level")
print(result)
```

---

measure_apply *Apply a function to measurement data along dimensions*

Description

Central dispatcher that enables 1D preprocessing operations to work on n-dimensional measurement data. For 1D data, it applies the function directly. For nD data, it slices along the specified dimensions, applies the function to each 1D slice, and rebuilds the nD structure.

Usage

```
measure_apply(x, fn, along = 1L, ...)
```

Arguments

x	A measure_tbl, measure_nd_tbl, measure_list, or measure_nd_list object.
fn	A function that accepts a measure_tbl and returns a measure_tbl. The function signature should be fn(x, ...).
along	Integer vector specifying which dimensions to apply the function along. For 2D data, along = 1 applies along dimension 1 (e.g., time in LC-DAD), treating dimension 2 slices as independent. Default is 1L (apply along the first dimension).
...	Additional arguments passed to fn.

Details

The `measure_apply()` function is the workhorse for making 1D preprocessing steps work on nD data. It handles:

- **1D data**: Direct function application
- **nD data**: Slice-apply-rebuild pattern

For nD data, the function extracts 1D slices along the specified dimension(s), applies the transformation function to each slice, and reassembles the result into the original nD structure.

Value

An object of the same class as the input, with the function applied to each 1D slice.

Examples

```
# Create a simple 2D measurement
m2d <- new_measure_nd_tbl(
  location_1 = rep(1:10, each = 3),
  location_2 = rep(1:3, times = 10),
  value = rnorm(30)
)

# Define a simple smoothing function for 1D data
smooth_1d <- function(x) {
  x$value <- stats::filter(x$value, rep(1/3, 3), sides = 2)
  x[!is.na(x$value), ]
}

# Apply smoothing along dimension 1
result <- measure_apply(m2d, smooth_1d, along = 1)
```

`measure_assess` *Assess Data Against Acceptance Criteria*

Description

Evaluates a set of values against acceptance criteria and returns a detailed assessment table with pass/fail status.

Usage

```
measure_assess(data, criteria, action = c("return", "warn", "error"))
```

Arguments

`data`	A named list or data frame containing the values to assess. Names must match criterion names.
`criteria`	A `measure_criteria()` object defining the acceptance criteria.
`action`	What to do on failure: `"return"` (default) returns the assessment table, `"warn"` issues a warning for failures, `"error"` stops on any critical failures.

Value

A tibble with class `measure_assessment` containing:

- `criterion`: Name of the criterion
- `value`: The observed value
- `threshold`: The threshold value(s)
- `operator`: The comparison operator
- `pass`: Logical indicating pass/fail
- `priority`: Priority level of the criterion
- `description`: Human-readable description

See Also

`measure_criteria()` for creating criteria, `criterion()` for individual criteria.

Examples

```
# Define criteria
crit <- measure_criteria(
  cv_qc = list("<", 15),
  r_squared = list(">=", 0.99),
  recovery = list("between", c(80, 120))
)

# Assess results
results <- list(cv_qc = 12.5, r_squared = 0.995, recovery = 98.2)
measure_assess(results, crit)

# Assess with some failures
results_bad <- list(cv_qc = 18.3, r_squared = 0.985, recovery = 75)
measure_assess(results_bad, crit)
```

`measure_axis_info` *Get axis information from measure data*

Description

Extracts metadata about the axis (location dimension) of measure data, including range, spacing, direction, and inferred axis type.

Usage

```
measure_axis_info(x, sample = 1L)
```

Arguments

`x`	A `measure_tbl`, `measure_list`, or data frame with measure column.
`sample`	Integer index of sample to analyze (for `measure_list`). Default is 1.

Value

A list with:

- `min`, `max`: Range of location values
- `n_points`: Number of data points
- `spacing`: Median absolute spacing between points
- `direction`: "increasing", "decreasing", or "mixed"
- `regular`: Logical indicating if spacing is regular (within tolerance)
- `axis_type`: Inferred axis type (see `infer_axis_type()`)

Examples

```
# NIR spectrum
spec <- new_measure_tbl(
  location = seq(1000, 2500, by = 2),
  value = rnorm(751)
)
measure_axis_info(spec)

# Chromatogram
chrom <- new_measure_tbl(
  location = seq(0, 30, by = 0.01),
  value = rnorm(3001)
)
measure_axis_info(chrom)
```

`measure_bland_altman` *Bland-Altman Method Comparison*

Description

Performs Bland-Altman analysis to compare two measurement methods. This calculates the mean bias, limits of agreement, and optionally tests for proportional bias.

Usage

```
measure_bland_altman(
  data,
  method1_col,
  method2_col,
  id_col = NULL,
  conf_level = 0.95,
  regression = c("none", "linear", "quadratic")
)
```

Arguments

`data`	A data frame containing paired measurements from both methods.
`method1_col`	Name of the column containing method 1 (reference) values.
`method2_col`	Name of the column containing method 2 (test) values.
`id_col`	Optional name of a column identifying paired observations.
`conf_level`	Confidence level for intervals. Default is 0.95.
`regression`	Test for proportional bias: • `"none"` (default): No regression test • `"linear"`: Test for linear trend in bias • `"quadratic"`: Test for quadratic trend

Details

Interpretation:

The Bland-Altman plot shows the difference between methods against their mean. Key features:

- **Mean bias**: Average difference (systematic error)
- **Limits of agreement (LOA)**: Range containing 95% of differences
- **Proportional bias**: Trend in differences with concentration

Acceptance Criteria:

Methods are typically considered interchangeable if:

- Mean bias is clinically/analytically insignificant
- LOA width is acceptable for the intended use
- No significant proportional bias

Value

A `measure_bland_altman` object containing:

- `data`: Tibble with mean, difference, and LOA for each observation
- `statistics`: List of summary statistics (bias, SD, LOA, CIs)
- `regression`: Regression results if requested (model, p-value)

See Also

`measure_deming_regression()`, `measure_passing_bablok()`

Other method-comparison: `measure_deming_regression()`, `measure_passing_bablok()`, `measure_proficiency_score()`

Examples

```
# Compare two blood glucose meters
set.seed(123)
data <- data.frame(
  patient_id = 1:30,
  meter_A = rnorm(30, mean = 100, sd = 15),
  meter_B = rnorm(30, mean = 102, sd = 16)
)

ba <- measure_bland_altman(
  data,
  method1_col = "meter_A",
  method2_col = "meter_B",
  regression = "linear"
)

print(ba)
tidy(ba)

# Visualize
ggplot2::autoplot(ba)
```

`measure_calibration` *Calibration Curve Object*

Description

A calibration curve object stores the fitted model, diagnostics, and metadata for quantitation workflows. Created by `measure_calibration_fit()`.

Structure

A `measure_calibration` object is a list containing:

model The underlying fitted model (lm object)

model_type Character: "linear" or "quadratic"

weights_type Character: weighting scheme used

formula The model formula

data The calibration data used for fitting

diagnostics List of diagnostic statistics

outliers Data frame of flagged outliers (if any)

call The original function call

See Also

`measure_calibration_fit()` for creating calibration objects, `measure_calibration_predict()` for prediction, `tidy.measure_calibration()` for extracting coefficients, `autoplot.measure_calibration()` for diagnostic plots.

`measure_calibration_fit` *Fit a Calibration Curve*

Description

Fits a weighted or unweighted calibration curve for quantitation. Supports linear and quadratic models with various weighting schemes.

Usage

```
measure_calibration_fit(
  data,
  formula,
  model = c("linear", "quadratic"),
  weights = c("none", "1/x", "1/x2", "1/y", "1/y2"),
  origin = FALSE,
  outlier_method = c("none", "studentized", "cook"),
  outlier_threshold = NULL,
  outlier_action = c("flag", "remove"),
  sample_type_col = NULL
)
```

Arguments

`data`	A data frame containing calibration data.
`formula`	A formula specifying the model. The left-hand side should be the response variable, and the right-hand side should be the concentration variable (e.g., `response ~ nominal_conc`).
`model`	Model type: `"linear"` (default) or `"quadratic"`.
`weights`	Weighting scheme: • `"none"` (default): Unweighted regression • `"1/x"`: Weight by 1/concentration • `"1/x2"`: Weight by 1/concentration^2 • `"1/y"`: Weight by 1/response • `"1/y2"`: Weight by 1/response^2 • A numeric vector of custom weights (must match data rows)
`origin`	Logical. If TRUE, force the curve through the origin (zero intercept). Default is FALSE.
`outlier_method`	Method for flagging outliers: • `"none"` (default): No outlier detection • `"studentized"`: Flag points with |studentized residual| > `outlier_threshold` • `"cook"`: Flag points with Cook's distance > `outlier_threshold`
`outlier_threshold`	Threshold for outlier detection. Default is 2.5 for studentized residuals or 1 for Cook's distance.
`outlier_action`	What to do with outliers: • `"flag"` (default): Flag but include in fit • `"remove"`: Remove from fit (with audit trail)
`sample_type_col`	Optional column name for sample type. If provided, only rows with `sample_type == "standard"` are used for fitting.

Details

Weighting:

Weighting is essential when response variance changes with concentration (heteroscedasticity). Common patterns:

- Constant CV: Use `"1/x2"` or `"1/y2"`
- Constant absolute error: Use `"none"`
- Proportional error: Use `"1/x"` or `"1/y"`

Outlier Handling:

By default, outliers are flagged but NOT removed. This follows the principle of "flag, don't drop" for analytical data. If removal is enabled, the removed points are stored in the result for audit purposes.

Value

A measure_calibration object containing the fitted model, diagnostics, and metadata.

See Also

measure_calibration_predict() for prediction, autoplot.measure_calibration() for diagnostic plots, tidy.measure_calibration() for extracting coefficients.

Examples

```
# Simple linear calibration
data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100, 200),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3, 295.7)
)
cal <- measure_calibration_fit(data, response ~ nominal_conc)
print(cal)

# Weighted calibration (1/x^2)
cal_weighted <- measure_calibration_fit(
  data,
  response ~ nominal_conc,
  weights = "1/x2"
)

# Quadratic model
cal_quad <- measure_calibration_fit(
  data,
  response ~ nominal_conc,
  model = "quadratic"
)
```

---

measure_calibration_predict

Predict Concentrations from Calibration Curve

Description

Uses a fitted calibration curve to predict concentrations from responses.

Usage

```
measure_calibration_predict(
  object,
  newdata,
  interval = c("none", "confidence", "prediction"),
  level = 0.95,
  ...
)
```

Arguments

`object`	A measure_calibration object from `measure_calibration_fit()`.
`newdata`	A data frame containing the response values to predict from. Must contain a column with the same name as the response variable in the calibration formula.
`interval`	Type of interval to calculate: • `"none"` (default): Point estimates only • `"confidence"`: Confidence intervals • `"prediction"`: Prediction intervals
`level`	Confidence level for intervals (default 0.95).
`...`	Additional arguments (unused).

Details

For inverse prediction (response -> concentration), the function uses root-finding when the model is quadratic. For linear models, direct algebraic inversion is used.

Interval Calculation:
Intervals are calculated using the delta method for the inverse prediction. For quadratic models, intervals are approximate.

Value

A tibble with columns:

- `.pred_conc`: Predicted concentration
- `.pred_lower`: Lower bound (if intervals requested)
- `.pred_upper`: Upper bound (if intervals requested)

See Also

`measure_calibration_fit()` for fitting calibration curves.

Examples

```
# Fit calibration curve
cal_data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3)
)
cal <- measure_calibration_fit(cal_data, response ~ nominal_conc)

# Predict concentrations from new responses
unknowns <- data.frame(response = c(45, 85, 120))
measure_calibration_predict(cal, unknowns)

# With prediction intervals
measure_calibration_predict(cal, unknowns, interval = "prediction")
```

`measure_calibration_verify` *Verify Calibration Curve Performance*

Description

Evaluates the performance of a calibration curve using verification samples (continuing calibration verification - CCV, or independent QC samples). This function assesses whether the calibration remains valid during or between analytical runs.

Usage

```
measure_calibration_verify(
  calibration,
  verification_data,
  nominal_col = "nominal_conc",
  acceptance_pct = 15,
  acceptance_pct_lloq = 20,
  lloq = NULL,
  sample_type_col = NULL,
  criteria = NULL
)
```

Arguments

`calibration`	A measure_calibration object from `measure_calibration_fit()`.
`verification_data`	A data frame containing verification samples with known concentrations.
`nominal_col`	Name of the column containing nominal (known) concentrations. Default is `"nominal_conc"`.
`acceptance_pct`	Acceptance criterion as percent deviation from nominal. Default is 15 (i.e., ±15%).
`acceptance_pct_lloq`	Acceptance criterion for samples at the lower limit of quantitation (LLOQ). Default is 20 (i.e., ±20%).
`lloq`	Lower limit of quantitation. Samples at or near this level use `acceptance_pct_lloq`. Default is NULL (use same criterion for all).
`sample_type_col`	Optional column indicating sample types. Only samples with type containing ”qc” or ”ccv” will be used if specified.
`criteria`	Optional measure_criteria object for custom acceptance criteria. If provided, overrides `acceptance_pct` settings.

Details

Verification Workflow:

Calibration verification is typically performed:

1. At the beginning and end of analytical batches
2. After every N unknown samples (e.g., every 10)
3. When instrument performance is in question

Acceptance Criteria:

Default criteria are based on bioanalytical guidelines:

- Standard samples: ±15% of nominal
- LLOQ samples: ±20% of nominal

For more stringent applications (e.g., clinical chemistry), consider using ±10% or providing custom criteria.

Value

A `measure_calibration_verify` object (a tibble) containing:

- Predicted concentrations
- Accuracy (%nominal)
- Deviation from nominal (%)
- Pass/fail status for each sample
- Overall verification status

See Also

`measure_calibration_fit()` for fitting calibration curves, `measure_calibration_predict()` for prediction, `measure_criteria()` for custom acceptance criteria.

Examples

```
# Fit calibration
cal_data <- data.frame(
  nominal_conc = c(1, 5, 10, 50, 100, 500),
  response = c(1.2, 5.8, 11.3, 52.1, 105.2, 498.7)
)
cal <- measure_calibration_fit(cal_data, response ~ nominal_conc)

# Verify with QC samples
qc_data <- data.frame(
  sample_id = c("QC_Low", "QC_Mid", "QC_High"),
  nominal_conc = c(3, 75, 400),
  response = c(3.3, 77.2, 385.1)
)

verify_result <- measure_calibration_verify(cal, qc_data)
print(verify_result)
```

`measure_carryover` *Carryover Assessment*

Description

Evaluates carryover by analyzing blank samples run after high-concentration samples.

Usage

```
measure_carryover(
  data,
  response_col,
  sample_type_col,
  run_order_col,
  blank_type = "blank",
  high_type = "high",
  threshold = 20,
  lloq = NULL
)
```

Arguments

`data`	A data frame containing the run sequence with blanks after highs.
`response_col`	Name of the column containing response values.
`sample_type_col`	Name of the column identifying sample types.
`run_order_col`	Name of the column containing run order.
`blank_type`	Value identifying blank samples. Default is `"blank"`.
`high_type`	Value identifying high-concentration samples. Default is `"high"`.
`threshold`	Carryover threshold as percentage of LLOQ or high response. Default is 20 (meaning 20% of LLOQ).
`lloq`	Optional LLOQ value for threshold calculation.

Details

Carryover is the appearance of analyte in a blank sample due to contamination from a previous high-concentration sample. It is typically assessed by analyzing blank samples immediately after the highest calibration standard or QC sample.

Acceptance Criteria (ICH M10):

Carryover in the blank sample following the high concentration should not exceed:

- 20% of the LLOQ (for the analyte)
- 5% of the internal standard response

Value

A `measure_carryover` object containing:

- `blank_responses`: Response values in blanks after high samples
- `mean_blank`: Mean blank response
- `max_blank`: Maximum blank response
- `high_responses`: High sample responses
- `carryover_pct`: Carryover as percentage of high or LLOQ
- `pass`: Whether carryover is within acceptable limits

See Also

`measure_accuracy()`, `measure_system_suitability()`

Other accuracy: `measure_accuracy()`, `measure_linearity()`

Examples

```
# Carryover assessment
data <- data.frame(
  run_order = 1:10,
  sample_type = c("std", "std", "std", "high", "blank",
                  "qc", "qc", "high", "blank", "std"),
  response = c(100, 500, 1000, 5000, 5, 500, 510, 4900, 8, 100)
)

result <- measure_carryover(
  data,
  response_col = "response",
  sample_type_col = "sample_type",
  run_order_col = "run_order",
  lloq = 50
)
print(result)
```

---

measure_column_patterns

Common column naming patterns for analytical data

Description

Named list of regex patterns for detecting measurement column types. Used by `measure_identify_columns()` for auto-detection. Users can extend or modify these patterns and pass them to detection functions.

Usage

```
measure_column_patterns
```

Format

Named list with regex patterns:

wavenumber `wn_` prefix for IR wavenumber (cm^-1)

wavelength `nm_` prefix for wavelength (nm)

retention_time `rt_` prefix for chromatography retention time

mz `mz_` prefix for mass-to-charge ratio (MS)

ppm `ppm_` prefix for NMR chemical shift

channel `ch_` prefix for numbered channels

generic `x_` prefix for generic/unknown axis

Examples

```
# View default patterns
measure_column_patterns

# Create custom patterns
my_patterns <- c(measure_column_patterns, list(custom = "^my_prefix_"))
```

---

measure_column_summary

Get Column Summary by Type

Description

Summarizes columns by their detected type, useful for understanding the structure of analytical datasets.

Usage

```
measure_column_summary(data, patterns = measure_column_patterns)
```

Arguments

`data`	A data frame to analyze.
`patterns`	Named list of regex patterns. Defaults to `measure_column_patterns`.

Value

A tibble summarizing each detected type:

type Column type

n_columns Number of columns of this type

example_cols First 3 column names of this type

Examples

```
df <- data.frame(
  id = 1:5,
  wn_1000 = rnorm(5), wn_1001 = rnorm(5), wn_1002 = rnorm(5),
  concentration = rnorm(5)
)
measure_column_summary(df)
```

---

measure_control_chart

Generate Control Chart

Description

Creates a control chart with optional multi-rule (Westgard) violation detection.

Usage

```
measure_control_chart(
  data,
  response_col,
  order_col,
  limits = NULL,
  rules = c("1_3s", "2_2s", "R_4s", "4_1s", "10x"),
  group_col = NULL
)
```

Arguments

`data`	A data frame containing QC measurements.
`response_col`	Name of the column containing QC values.
`order_col`	Name of the column containing run order/sequence.
`limits`	Optional `measure_control_limits` object. If NULL, calculated from the data.
`rules`	Character vector of Westgard rules to apply. Default is `c("1_3s", "2_2s", "R_4s", "4_1s", "10x")`.
`group_col`	Optional grouping column.

Details

Westgard Rules:

The function supports common Westgard multi-rules:

- **1:3s**: One point beyond 3 sigma (action required)
- **2:2s**: Two consecutive points beyond 2 sigma (warning)

- **R:4s**: Range of two consecutive points > 4 sigma
- **4:1s**: Four consecutive points beyond 1 sigma (same side)
- **10x**: Ten consecutive points on same side of mean

Interpretation:

- Violations are flagged with the specific rule that was triggered
- Multiple rules can be triggered by the same point
- A run is considered "in control" if no violations are detected

Value

A `measure_control_chart` object containing:

- `data`: The input data with added violation flags
- `limits`: The control limits used
- `violations`: Summary of rule violations
- `rules_applied`: Which rules were checked

See Also

`measure_control_limits()`, `autoplot.measure_control_chart()`

Other control-charts: `measure_control_limits()`, `measure_system_suitability()`

Examples

```
# Generate control chart with Westgard rules
set.seed(123)
qc_data <- data.frame(
  run_order = 1:50,
  qc_value = c(rnorm(45, 100, 2), rnorm(5, 106, 2))  # Last 5 shifted
)
chart <- measure_control_chart(qc_data, "qc_value", "run_order")
print(chart)
```

`measure_control_limits`

Calculate Control Limits

Description

Calculates control limits for quality control monitoring using Shewhart rules and optionally EWMA or CUSUM statistics.

Usage

```
measure_control_limits(
  data,
  response_col,
  group_col = NULL,
  type = c("shewhart", "ewma", "cusum"),
  n_sigma = 3,
  target = NULL,
  lambda = 0.2,
  k = 0.5,
  h = 5
)
```

Arguments

`data`	A data frame containing QC measurements.
`response_col`	Name of the column containing QC values.
`group_col`	Optional grouping column (e.g., for different QC levels).
`type`	Type of control chart: `"shewhart"` (default), `"ewma"`, or `"cusum"`.
`n_sigma`	Number of standard deviations for control limits. Default is 3.
`target`	Optional target value. If NULL, calculated from data mean.
`lambda`	EWMA smoothing parameter (0 < lambda <= 1). Default is 0.2.
`k`	CUSUM slack parameter. Default is 0.5 (in sigma units).
`h`	CUSUM decision interval. Default is 5 (in sigma units).

Details

Shewhart Charts:

Classic control charts with limits at mean +/- n*sigma:

- UCL/LCL: Action limits (typically 3 sigma)
- UWL/LWL: Warning limits (typically 2 sigma)

EWMA Charts:

Exponentially weighted moving average, more sensitive to small shifts:

- Control limits narrow as more data is collected
- Lambda parameter controls weight of recent observations

CUSUM Charts:

Cumulative sum chart for detecting persistent shifts:

- Upper and lower CUSUM statistics track cumulative deviations
- Decision interval h determines sensitivity

Value

A `measure_control_limits` object containing:

- `center`: Center line (target or mean)
- `lcl`: Lower control limit
- `ucl`: Upper control limit
- `lwl`: Lower warning limit (2 sigma)
- `uwl`: Upper warning limit (2 sigma)
- `sigma`: Estimated standard deviation
- Additional statistics depending on chart type

See Also

`measure_control_chart()`, `measure_system_suitability()`

Other control-charts: `measure_control_chart()`, `measure_system_suitability()`

Examples

```
# Calculate Shewhart control limits
set.seed(123)
qc_data <- data.frame(
  run_order = 1:30,
  qc_value = rnorm(30, mean = 100, sd = 2)
)
limits <- measure_control_limits(qc_data, "qc_value")
print(limits)

# EWMA control limits
limits_ewma <- measure_control_limits(qc_data, "qc_value", type = "ewma")
```

`measure_criteria` *Create a Set of Acceptance Criteria*

Description

Combines multiple `criterion()` objects into a criteria set for use with `measure_assess()`.

Usage

```
measure_criteria(..., .list = NULL)
```

Arguments

`...`	`criterion()` objects or named arguments that will be converted to criteria. Named arguments use the format `name = list(operator, threshold)` or `name = threshold` (assumes `"<="`).
`.list`	Optional list of criterion objects.

Value

A `measure_criteria` object (list of `measure_criterion` objects).

See Also

`criterion()` for creating individual criteria, `measure_assess()` for evaluating criteria.

Examples

```
# Using criterion() objects
measure_criteria(
  criterion("cv_qc", "<", 15),
  criterion("r_squared", ">=", 0.99),
  criterion("recovery", "between", c(80, 120))
)

# Using shorthand notation
measure_criteria(
  cv_qc = list("<", 15),
  r_squared = list(">=", 0.99),
  bias = list("between", c(-10, 10))
)

# Simple threshold (assumes "<=")
measure_criteria(
  cv = 15,       # cv <= 15
  rsd = 20       # rsd <= 20
)
```

---

measure_deming_regression

Deming Regression for Method Comparison

Description

Performs Deming regression to compare two measurement methods when both have measurement error. This is preferred over ordinary least squares when both methods have non-negligible error.

Usage

```
measure_deming_regression(
  data,
  method1_col,
  method2_col,
  error_ratio = NULL,
  method1_sd = NULL,
  method2_sd = NULL,
```

```
  bootstrap = FALSE,
  bootstrap_n = 1000,
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing paired measurements.
`method1_col`	Name of column for method 1 (typically reference/comparator).
`method2_col`	Name of column for method 2 (typically test method).
`error_ratio`	Ratio of error variances (var_method2 / var_method1). Default is 1 (equal variances). Can be estimated from replicate data.
`method1_sd`	Optional known SD of method 1. Used to calculate error_ratio.
`method2_sd`	Optional known SD of method 2. Used to calculate error_ratio.
`bootstrap`	Use bootstrap for confidence intervals? Default is FALSE.
`bootstrap_n`	Number of bootstrap samples. Default is 1000.
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Error Ratio:
The error ratio (lambda) represents the ratio of error variances: `lambda = var(method2) / var(method1)`
Common approaches:

- `lambda = 1`: Assume equal error variances
- Estimate from replicates: Use SDs from replicate measurements
- Estimate from calibration: Use known method precision data

Interpretation:
For equivalent methods:

- Slope should be close to 1 (proportional agreement)
- Intercept should be close to 0 (no constant bias)

If 95% CI for slope includes 1 and CI for intercept includes 0, methods are considered equivalent.

Implementation:
If the `mcr` package is available, it is used for fitting. Otherwise, a manual implementation is used with optional bootstrap CIs.

Value

A `measure_deming_regression` object containing:

- `coefficients`: Tibble with intercept and slope estimates and CIs
- `statistics`: List of diagnostic statistics (RMSE, R-squared)
- `data_summary`: Summary of input data
- `bootstrap`: Bootstrap results if requested

See Also

measure_bland_altman(), measure_passing_bablok()

Other method-comparison: measure_bland_altman(), measure_passing_bablok(), measure_proficiency_scor

Examples

```
# Method comparison data
data <- data.frame(
  reference = c(5.2, 10.5, 15.8, 25.3, 50.1, 75.4, 100.2),
  new_method = c(5.1, 10.8, 16.2, 25.9, 49.8, 76.1, 101.3)
)

# Deming regression with bootstrap CIs
result <- measure_deming_regression(
  data,
  method1_col = "reference",
  method2_col = "new_method",
  bootstrap = TRUE,
  bootstrap_n = 500
)

print(result)
tidy(result)
```

---

measure_detect_drift *Detect Drift in Analytical Data*

Description

Detects significant drift in feature responses across run order using trend tests and/or slope analysis.

Usage

```
measure_detect_drift(
  data,
  features,
  run_order_col = "run_order",
  sample_type_col = "sample_type",
  qc_type = NULL,
  method = c("slope", "mann_kendall", "both")
)
```

Arguments

data	A data frame containing the measurement data.
features	Character vector of feature column names to analyze.
run_order_col	Name of the run order column.

`sample_type_col`
: Name of the sample type column.

`qc_type`
: Value(s) identifying QC samples. If provided, analysis is restricted to QC samples.

`method`
: Detection method:
  - `"slope"` (default): Linear regression slope test
  - `"mann_kendall"`: Mann-Kendall trend test
  - `"both"`: Both methods

Value

A tibble with drift statistics for each feature:

- `feature`: Feature name
- `slope`: Regression slope (change per run)
- `slope_pvalue`: P-value for slope != 0
- `percent_change`: Total percent change over run
- `significant`: Logical, TRUE if drift is statistically significant

Examples

```
# Create data with drift
data <- data.frame(
  sample_type = rep("qc", 20),
  run_order = 1:20,
  feature1 = 100 + (1:20) * 0.5 + rnorm(20, sd = 2),
  feature2 = 50 + rnorm(20, sd = 1)  # No drift
)

measure_detect_drift(data, c("feature1", "feature2"))
```

`measure_dim_names` *Get dimension names of an n-dimensional measurement*

Description

Returns the semantic names for each dimension (e.g., ”wavelength”, ”retention_time”).

Usage

```
measure_dim_names(x)
```

Arguments

`x`
: A `measure_nd_tbl` or `measure_nd_list` object.

Value

Character vector of dimension names, or `NULL` if not set.

Examples

```
m2d <- new_measure_nd_tbl(
  location_1 = 1:10,
  location_2 = rep(1:2, each = 5),
  value = rnorm(10),
  dim_names = c("retention_time", "wavelength")
)
measure_dim_names(m2d)
```

---

measure_dim_units *Get dimension units of an n-dimensional measurement*

Description

Returns the units for each dimension (e.g., "nm", "min").

Usage

```
measure_dim_units(x)
```

Arguments

`x`	A `measure_nd_tbl` or `measure_nd_list` object.

Value

Character vector of dimension units, or `NULL` if not set.

Examples

```
m2d <- new_measure_nd_tbl(
  location_1 = 1:10,
  location_2 = rep(1:2, each = 5),
  value = rnorm(10),
  dim_units = c("min", "nm")
)
measure_dim_units(m2d)
```

`measure_fold` *Fold 1D measurement back to n-dimensional*

Description

Reconstructs an n-dimensional measurement from a 1D vector that was created by `measure_unfold()`. Requires the fold metadata attribute.

Usage

```
measure_fold(x)
```

Arguments

`x`	A `measure_tbl` or `measure_list` with `"fold_info"` attribute.

Value

A `measure_nd_tbl` or `measure_nd_list` with the original dimensional structure restored.

See Also

`measure_unfold()` to create foldable 1D data

Examples

```
# Create, unfold, then fold back
m2d <- new_measure_nd_tbl(
  location_1 = rep(1:3, each = 4),
  location_2 = rep(1:4, times = 3),
  value = 1:12
)

m1d <- measure_unfold(m2d)
m2d_restored <- measure_fold(m1d)

# Values are preserved
all.equal(m2d$value, m2d_restored$value)
```

`measure_gage_rr` *Gage R&R (Measurement System Analysis)*

Description

Performs a Gage Repeatability and Reproducibility study to assess measurement system variation.

Usage

```
measure_gage_rr(
  data,
  response_col,
  part_col,
  operator_col,
  tolerance = NULL,
  conf_level = 0.95,
  k = 5.15
)
```

Arguments

`data`	A data frame containing Gage R&R study data.
`response_col`	Name of the column containing the measurements.
`part_col`	Name of the column identifying parts/samples.
`operator_col`	Name of the column identifying operators/analysts.
`tolerance`	Optional specification tolerance for calculating %Study variation and %Tolerance.
`conf_level`	Confidence level. Default is 0.95.
`k`	Multiplier for study variation calculation. Default is 5.15 (99%).

Details

Gage R&R decomposes total measurement variation into:

- **Repeatability (EV)**: Equipment variation - variability from repeated measurements by the same operator on the same part
- **Reproducibility (AV)**: Appraiser variation - variability between operators measuring the same parts
- **Part-to-Part (PV)**: True variation between parts

Acceptance Criteria (typical guidelines):

- %R&R < 10%: Measurement system acceptable
- %R&R 10-30%: Measurement system may be acceptable depending on application
- %R&R > 30%: Measurement system needs improvement

The number of distinct categories (ndc) should be >= 5 for a capable measurement system.

Value

A `measure_gage_rr` object containing:

- Variance components (Repeatability, Reproducibility, Part-to-Part)
- %Contribution of each component
- %Study Variation (using k * sigma)
- %Tolerance (if tolerance provided)
- Number of distinct categories (ndc)

See Also

`measure_repeatability()`, `measure_intermediate_precision()`

Other precision: `measure_intermediate_precision()`, `measure_repeatability()`, `measure_reproducibility(`

Examples

```
# Gage R&R study with 10 parts, 3 operators, 2 replicates each
set.seed(123)
data <- expand.grid(
  part = 1:10,
  operator = c("A", "B", "C"),
  replicate = 1:2
)
data$measurement <- 50 +
  (data$part - 5) * 2 +  # Part-to-part variation
  ifelse(data$operator == "A", 0.5,
         ifelse(data$operator == "B", -0.3, 0)) +  # Operator effect
  rnorm(nrow(data), 0, 0.5)  # Repeatability

result <- measure_gage_rr(
  data,
  response_col = "measurement",
  part_col = "part",
  operator_col = "operator",
  tolerance = 20
)
print(result)
```

---

measure_grid_info *Get grid information for an n-dimensional measurement*

Description

Returns detailed information about the coordinate grid, including unique values per dimension, grid shape, and regularity status.

Usage

```
measure_grid_info(x)
```

Arguments

x	A measure_nd_tbl object.

Value

A list with components:

- ndim: Number of dimensions
- dim_names: Semantic dimension names (if set)
- dim_units: Dimension units (if set)
- unique_values: List of unique coordinate values per dimension
- shape: Integer vector of unique value counts per dimension
- n_points: Total number of data points
- is_regular: Whether the grid is regular
- has_na: Whether any values are NA

Examples

```
m2d <- new_measure_nd_tbl(
  location_1 = rep(seq(0, 10, by = 2), each = 4),
  location_2 = rep(c(254, 280, 320, 350), times = 6),
  value = rnorm(24),
  dim_names = c("time", "wavelength"),
  dim_units = c("min", "nm")
)
measure_grid_info(m2d)
```

---

measure_identify_columns

Identify Column Types in Analytical Data

Description

Automatically detects column types in a data frame based on naming conventions common in analytical chemistry. This helps set up recipes with appropriate roles for different column types.

Usage

```
measure_identify_columns(data, patterns = measure_column_patterns)
```

Arguments

`data`	A data frame to analyze.
`patterns`	Named list of regex patterns for column detection. Defaults to `measure_column_patterns`. Custom patterns can be provided as a named list where names become the detected type.

Details

Column type detection uses the following naming conventions:

Prefix	Type	Suggested Role	Use Case
`wn_*`	wavenumber	predictor	IR spectroscopy (cm^-1)
`nm_*`	wavelength	predictor	UV-Vis, NIR spectroscopy
`rt_*`	retention_time	predictor	Chromatography
`mz_*`	mz	predictor	Mass spectrometry
`ppm_*`	ppm	predictor	NMR spectroscopy
`ch_*`	channel	predictor	Generic channel data
`x_*`	generic	predictor	Generic measurements

Columns not matching any pattern are classified as "other" and suggested as either "outcome" (if numeric), "id" (if character/factor with unique values), or "predictor".

Value

A tibble with columns:

column Column name

type Detected type (from pattern names, or "other" if no match)

suggested_role Suggested recipe role based on type

n_values Number of non-NA values

class R class of the column

Examples

```
# Wide format spectral data
df <- data.frame(
  sample_id = 1:5,
  outcome = rnorm(5),
  wn_1000 = rnorm(5),
  wn_1001 = rnorm(5),
  wn_1002 = rnorm(5)
)
measure_identify_columns(df)

# Chromatography data
df2 <- data.frame(
  id = letters[1:3],
  concentration = c(1.2, 2.3, 3.4),
```

```
  rt_0.5 = rnorm(3),
  rt_1.0 = rnorm(3),
  rt_1.5 = rnorm(3)
)
measure_identify_columns(df2)
```

`measure_intermediate_precision` *Intermediate Precision (Between-Run Precision)*

Description

Calculates intermediate precision statistics for measurements performed under varying conditions (different days, analysts, or instruments).

Usage

```
measure_intermediate_precision(
  data,
  response_col,
  factors,
  group_col = NULL,
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing measurements with factor columns.
`response_col`	Name of the column containing the response values.
`factors`	Character vector of factor column names (e.g., `c("day", "analyst")`).
`group_col`	Optional grouping column (e.g., concentration level).
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Intermediate precision quantifies the variability due to different conditions within the same laboratory. This typically includes:

- Different days
- Different analysts
- Different equipment (of the same type)

The function uses a one-way or nested ANOVA approach to estimate variance components. For more complex designs, consider using mixed effects models with the `lme4` package.

Value

A `measure_precision` object containing variance components and precision estimates:

- `component`: Name of the variance component
- `variance`: Estimated variance
- `percent_variance`: Percentage of total variance
- `sd`: Standard deviation (square root of variance)
- `cv`: Coefficient of variation (%) for that component

See Also

`measure_repeatability()`, `measure_reproducibility()`

Other precision: `measure_gage_rr()`, `measure_repeatability()`, `measure_reproducibility()`

Examples

```
# Intermediate precision across days
set.seed(123)
data <- data.frame(
  day = rep(1:5, each = 6),
  concentration = rnorm(30, mean = 100, sd = 3) +
    rep(rnorm(5, 0, 2), each = 6)  # Day effect
)
measure_intermediate_precision(data, "concentration", factors = "day")
```

`measure_is_regular` *Check if an n-dimensional measurement has a regular grid*

Description

A regular grid means all combinations of unique coordinate values exist exactly once (i.e., it forms a complete rectangular grid).

Usage

```
measure_is_regular(x)
```

Arguments

`x`	A `measure_nd_tbl` object.

Value

Logical indicating if the measurement has a regular grid.

Examples

```
# Regular grid (all combinations present)
regular <- new_measure_nd_tbl(
  location_1 = rep(1:3, each = 2),
  location_2 = rep(1:2, times = 3),
  value = rnorm(6)
)
measure_is_regular(regular)  # TRUE

# Irregular grid (missing combinations)
irregular <- new_measure_nd_tbl(
  location_1 = c(1, 1, 2, 3),
  location_2 = c(1, 2, 1, 2),
  value = rnorm(4)
)
measure_is_regular(irregular)  # FALSE
```

---

measure_linearity *Linearity Assessment*

Description

Assesses linearity of a method by evaluating the relationship between response and concentration across the specified range.

Usage

```
measure_linearity(
  data,
  conc_col,
  response_col,
  method = c("regression", "residual"),
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing concentration and response data.
`conc_col`	Name of the column containing concentrations.
`response_col`	Name of the column containing responses.
`method`	Linearity assessment method: • `"regression"` (default): Linear regression with diagnostics • `"residual"`: Residual analysis and lack-of-fit test
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Linearity demonstrates that the method produces results that are directly proportional to analyte concentration within a given range.

Assessment Criteria:

- R-squared >= 0.99 (typical for many applications)
- Residuals randomly distributed around zero
- No systematic pattern in residual plots
- Lack-of-fit test not significant ($p > 0.05$)

ICH Q2 Requirements:

Linearity should be evaluated across the range with at least 5 concentration levels. Report the regression equation, correlation coefficient, and visual inspection of residual plots.

Value

A `measure_linearity` object containing:

- `r_squared`: Coefficient of determination
- `adj_r_squared`: Adjusted R-squared
- `slope`: Regression slope with CI
- `intercept`: Regression intercept with CI
- `residual_sd`: Residual standard deviation
- `lack_of_fit`: Lack-of-fit test results (if replicates exist)
- `range`: Concentration range evaluated

See Also

`measure_accuracy()`, `measure_calibration_fit()`

Other accuracy: `measure_accuracy()`, `measure_carryover()`

Examples

```
# Linearity assessment
set.seed(123)
data <- data.frame(
  concentration = rep(c(10, 25, 50, 75, 100), each = 3),
  response = rep(c(10, 25, 50, 75, 100), each = 3) * 1.5 + rnorm(15, 0, 2)
)

result <- measure_linearity(data, "concentration", "response")
print(result)
```

`measure_lod` *Calculate Limit of Detection (LOD)*

Description

Calculates the limit of detection using one of several accepted methods. The method used is explicitly documented in the output.

Usage

```
measure_lod(
  data,
  response_col,
  method = c("blank_sd", "calibration", "sn", "precision"),
  conc_col = "nominal_conc",
  sample_type_col = "sample_type",
  calibration = NULL,
  k = 3,
  sn_col = NULL,
  noise = NULL,
  sn_threshold = 3,
  ...
)
```

Arguments

`data`	A data frame containing the measurement data.
`response_col`	Name of the response column.
`method`	Method for LOD calculation: • `"blank_sd"`: 3 * SD of blank samples (requires `sample_type == "blank"`) • `"calibration"`: 3.3 * sigma / slope from calibration curve • `"sn"`: Signal-to-noise ratio method (requires `sn_col` or noise estimate) • `"precision"`: Based on acceptable precision at low concentrations
`conc_col`	Name of concentration column (for calibration method).
`sample_type_col`	Name of sample type column. Default is `"sample_type"`.
`calibration`	Optional measure_calibration object for calibration method.
`k`	Multiplier for SD. Default is 3 for LOD.
`sn_col`	Column containing S/N ratios (for `"sn"` method).
`noise`	Noise estimate for S/N calculation (alternative to `sn_col`).
`sn_threshold`	S/N threshold for LOD (default 3).
`...`	Additional arguments passed to method-specific calculations.

Details

Blank SD Method:
LOD = mean(blank) + k * SD(blank)
Where k is typically 3. This is a simple but widely accepted approach.

Calibration Method:
LOD = k * sigma / slope
Where sigma is the residual standard error of the calibration curve and slope is the calibration slope. k is typically 3.3 for LOD.

Signal-to-Noise Method:
LOD is the concentration where S/N = threshold (typically 3:1).

Precision-Based Method:
LOD is the lowest concentration where precision (CV) meets a specified criterion.

Value

A `measure_lod` object containing:

- `value`: The LOD value
- `method`: Method used
- `parameters`: Method-specific parameters
- `uncertainty`: Uncertainty estimate (when available)

See Also

`measure_loq()` for limit of quantitation, `measure_lod_loq()` for calculating both together.

Examples

```
# Create sample data with blanks
data <- data.frame(
  sample_type = c(rep("blank", 10), rep("standard", 5)),
  response = c(rnorm(10, mean = 0.5, sd = 0.1),
               c(5, 15, 35, 70, 150)),
  nominal_conc = c(rep(0, 10), c(10, 25, 50, 100, 200))
)

# LOD from blank SD
measure_lod(data, "response", method = "blank_sd")

# LOD from calibration curve
cal <- measure_calibration_fit(
  data[data$sample_type == "standard", ],
  response ~ nominal_conc
)
measure_lod(data, "response", method = "calibration", calibration = cal)
```

`measure_lod_loq` *Calculate LOD and LOQ Together*

Description

Convenience function to calculate both LOD and LOQ using the same method.

Usage

```
measure_lod_loq(
  data,
  response_col,
  method = c("blank_sd", "calibration", "sn", "precision"),
  conc_col = "nominal_conc",
  sample_type_col = "sample_type",
  calibration = NULL,
  k_lod = NULL,
  k_loq = 10,
  ...
)
```

Arguments

`data`	A data frame containing the measurement data.
`response_col`	Name of the response column.
`method`	Method for LOD calculation: • `"blank_sd"`: 3 * SD of blank samples (requires `sample_type == "blank"`) • `"calibration"`: 3.3 * sigma / slope from calibration curve • `"sn"`: Signal-to-noise ratio method (requires `sn_col` or noise estimate) • `"precision"`: Based on acceptable precision at low concentrations
`conc_col`	Name of concentration column (for calibration method).
`sample_type_col`	Name of sample type column. Default is `"sample_type"`.
`calibration`	Optional measure_calibration object for calibration method.
`k_lod`	Multiplier for LOD (default 3 or 3.3 for calibration).
`k_loq`	Multiplier for LOQ (default 10).
`...`	Additional arguments passed to method-specific calculations.

Value

A list with components `lod` and `loq`, each being the respective limit object.

See Also

measure_lod(), measure_loq().

Examples

```
data <- data.frame(
  sample_type = c(rep("blank", 10), rep("standard", 5)),
  response = c(rnorm(10, mean = 0.5, sd = 0.1),
               c(5, 15, 35, 70, 150)),
  nominal_conc = c(rep(0, 10), c(10, 25, 50, 100, 200))
)

limits <- measure_lod_loq(data, "response", method = "blank_sd")
limits$lod
limits$loq
```

---

measure_loq *Calculate Limit of Quantitation (LOQ)*

Description

Calculates the limit of quantitation using one of several accepted methods. The method used is explicitly documented in the output.

Usage

```
measure_loq(
  data,
  response_col,
  method = c("blank_sd", "calibration", "sn", "precision"),
  conc_col = "nominal_conc",
  sample_type_col = "sample_type",
  calibration = NULL,
  k = 10,
  sn_col = NULL,
  noise = NULL,
  sn_threshold = 10,
  precision_cv = 20,
  ...
)
```

Arguments

data	A data frame containing the measurement data.
response_col	Name of the response column.
method	Method for LOD calculation:

- `"blank_sd"`: 3 * SD of blank samples (requires `sample_type == "blank"`)
- `"calibration"`: 3.3 * sigma / slope from calibration curve
- `"sn"`: Signal-to-noise ratio method (requires `sn_col` or noise estimate)
- `"precision"`: Based on acceptable precision at low concentrations

`conc_col` Name of concentration column (for calibration method).

`sample_type_col` Name of sample type column. Default is `"sample_type"`.

`calibration` Optional measure_calibration object for calibration method.

`k` Multiplier for SD. Default is 10 for LOQ.

`sn_col` Column containing S/N ratios (for `"sn"` method).

`noise` Noise estimate for S/N calculation (alternative to `sn_col`).

`sn_threshold` S/N threshold for LOQ (default 10).

`precision_cv` Maximum allowable CV for LOQ (default 20%).

`...` Additional arguments passed to method-specific calculations.

Details

Blank SD Method:

LOQ = mean(blank) + k * SD(blank)

Where k is typically 10. This is a simple but widely accepted approach.

Calibration Method:

LOQ = k * sigma / slope

Where sigma is the residual standard error of the calibration curve and slope is the calibration slope. k is typically 10 for LOQ.

Signal-to-Noise Method:

LOQ is the concentration where S/N = threshold (typically 10:1).

Precision-Based Method:

LOQ is the lowest concentration where precision (CV) is <= the specified criterion (typically 20% for bioanalytical methods).

Value

A `measure_loq` object containing:

- `value`: The LOQ value
- `method`: Method used
- `parameters`: Method-specific parameters
- `uncertainty`: Uncertainty estimate (when available)

See Also

measure_lod() for limit of detection, measure_lod_loq() for calculating both together.

Examples

```
# Create sample data with blanks
data <- data.frame(
  sample_type = c(rep("blank", 10), rep("standard", 5)),
  response = c(rnorm(10, mean = 0.5, sd = 0.1),
               c(5, 15, 35, 70, 150)),
  nominal_conc = c(rep(0, 10), c(10, 25, 50, 100, 200))
)

# LOQ from blank SD
measure_loq(data, "response", method = "blank_sd")
```

---

measure_map *Apply a Function to Each Sample's Measurements*

Description

measure_map() applies a function to each sample's measurement data. This function is intended for **exploration and prototyping**, not for production pipelines. For reproducible preprocessing, use step_measure_map() instead.

Usage

```
measure_map(
  .data,
  .f,
  .cols = NULL,
  ...,
  verbosity = 1L,
  .error_call = rlang::caller_env()
)
```

Arguments

.data	A data frame containing one or more measure_list columns.
.f	A function or formula to apply to each sample's measurement tibble. • If a **function**, it is used as-is. • If a **formula** (e.g., ~ { .x$value <- log(.x$value); .x }), it is converted to a function using rlang::as_function().
.cols	<tidy-select> Columns to apply the transformation to. Defaults to all measure_list columns.
...	Additional arguments passed to .f.

`verbosity` An integer controlling output verbosity:

- `0`: Silent - suppress all messages and output from `.f`
- `1`: Normal (default) - show output from `.f`

`.error_call` The execution environment for error reporting.

Details

Intended Use: Exploration, Not Production:

This function is designed for interactive exploration and debugging:

```
# Good: Prototyping a new transformation
baked_data |>
  measure_map(~ { .x$value <- my_experimental_fn(.x$value); .x })

# Better: Once it works, put it in a recipe step
recipe(...) |>
  step_measure_map(my_experimental_fn) |>
  prep()
```

Unlike recipe steps, transformations applied with `measure_map()` are NOT:

- Automatically applied to new data
- Bundled into workflows
- Reproducible across sessions

Function Requirements:

The function `.f` must:

- Accept a tibble with `location` and `value` columns
- Return a tibble with `location` and `value` columns
- Not change the number of rows

Value

A data frame with the specified measure columns transformed.

See Also

- `step_measure_map()` for production use in recipe pipelines
- `measure_map_safely()` for fault-tolerant exploration
- `measure_summarize()` for computing summary statistics

Examples

```
library(recipes)

# First, get data in internal format
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
```

```
  prep()

baked_data <- bake(rec, new_data = NULL)

# Explore a custom transformation
result <- measure_map(baked_data, ~ {
  # Subtract the minimum value from each spectrum
  .x$value <- .x$value - min(.x$value)
  .x
})

# Once you're happy with it, use step_measure_map() in your recipe:
# recipe(...) |>
#   step_measure_map(~ { .x$value <- .x$value - min(.x$value); .x })
```

`measure_map_safely` *Apply a Function Safely to Each Sample's Measurements*

Description

`measure_map_safely()` is a fault-tolerant version of `measure_map()` that captures errors instead of stopping execution. This is useful when exploring data that may have problematic samples.

Usage

```
measure_map_safely(
  .data,
  .f,
  .cols = NULL,
  ...,
  .otherwise = NULL,
  .error_call = rlang::caller_env()
)
```

Arguments

`.data`	A data frame containing one or more `measure_list` columns.
`.f`	A function or formula to apply to each sample's measurement tibble. • If a **function**, it is used as-is. • If a **formula** (e.g., `~ { .x$value <- log(.x$value); .x }`), it is converted to a function using `rlang::as_function()`.
`.cols`	<`tidy-select`> Columns to apply the transformation to. Defaults to all `measure_list` columns.
`...`	Additional arguments passed to `.f`.
`.otherwise`	Value to use when `.f` fails for a sample. Default is `NULL`, which keeps the original (untransformed) measurement.
`.error_call`	The execution environment for error reporting.

Value

A list with two elements:

- `result`: A data frame with transformations applied where successful
- `errors`: A tibble with columns `column`, `sample`, and `error`

See Also

`measure_map()` for standard (fail-fast) mapping

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

baked_data <- bake(rec, new_data = NULL)

# A function that might fail for some samples
risky_transform <- function(x) {
  if (any(x$value < 0)) stop("Negative values not allowed")
  x$value <- log(x$value)
  x
}

# Errors are captured, not thrown
result <- measure_map_safely(baked_data, risky_transform)

# Check which samples failed
if (nrow(result$errors) > 0) {
  print(result$errors)
}
```

---

measure_matrix_effect

Matrix Effect Analysis

Description

Quantifies matrix effects (ion suppression/enhancement) by comparing analyte response in matrix versus neat solution. This is essential for validating LC-MS/MS and other analytical methods where matrix interference is a concern.

Usage

```
measure_matrix_effect(
  data,
  response_col,
  sample_type_col,
  matrix_level,
  neat_level,
  concentration_col = NULL,
  analyte_col = NULL,
  group_cols = NULL,
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing response data.
`response_col`	Name of the column containing analyte responses.
`sample_type_col`	Name of the column indicating sample type (matrix vs neat/standard).
`matrix_level`	Value in `sample_type_col` indicating matrix samples.
`neat_level`	Value in `sample_type_col` indicating neat/standard samples.
`concentration_col`	Optional column for concentration levels. If provided, matrix effects are calculated per concentration.
`analyte_col`	Optional column for analyte names. If provided, matrix effects are calculated per analyte.
`group_cols`	Additional grouping columns (e.g., batch, matrix source).
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Matrix Effect Calculation:

Matrix effect (ME%) is calculated as: `ME% = (response_in_matrix / response_in_neat) * 100`

Or equivalently: `ME% = 100 + ((response_in_matrix - response_in_neat) / response_in_neat) * 100`

Interpretation:

- **ME = 100%**: No matrix effect
- **ME > 100%**: Ion enhancement
- **ME < 100%**: Ion suppression

Acceptance Criteria (typical):

According to ICH M10 and FDA guidance:

- ME should be between 80-120% (±20%)
- CV of ME should be 15%

Experimental Design:

To assess matrix effects:

1. Prepare blank matrix (e.g., plasma) from multiple sources
2. Spike analyte post-extraction at known concentration
3. Compare to analyte in neat solvent at same concentration

Value

A `measure_matrix_effect` object containing:

- `results`: Tibble with matrix effect percentages per group
- `statistics`: Overall summary statistics
- `raw_data`: Data used for calculations

See Also

`step_measure_standard_addition()`, `measure_accuracy()`

Other calibration: `step_measure_dilution_correct()`, `step_measure_standard_addition()`, `step_measure_surrogate_recovery()`

Examples

```
# Matrix effect study data
me_data <- data.frame(
  sample_type = rep(c("matrix", "neat"), each = 6),
  matrix_lot = rep(c("Lot1", "Lot2", "Lot3", "Lot1", "Lot2", "Lot3"), 2),
  concentration = rep(c("low", "high"), each = 3, times = 2),
  response = c(
    # Matrix samples (some suppression)
    9500, 9800, 9200, 48000, 49500, 47000,
    # Neat samples
    10000, 10000, 10000, 50000, 50000, 50000
  )
)

me <- measure_matrix_effect(
  me_data,
  response_col = "response",
  sample_type_col = "sample_type",
  matrix_level = "matrix",
  neat_level = "neat",
  concentration_col = "concentration"
)

print(me)
tidy(me)
```

measure_ndim *Get the number of dimensions of a measurement*

Description

Returns the dimensionality of a measurement object. For 1D measurements (`measure_tbl`), returns 1. For n-dimensional measurements (`measure_nd_tbl`), returns the number of location dimensions.

Usage

```
measure_ndim(x)
```

Arguments

x	A `measure_tbl`, `measure_nd_tbl`, `measure_list`, or `measure_nd_list` object.

Value

Integer indicating the number of dimensions.

Examples

```
# 1D measurement
m1d <- new_measure_tbl(location = 1:10, value = rnorm(10))
measure_ndim(m1d)  # 1

# 2D measurement
m2d <- new_measure_nd_tbl(
  location_1 = rep(1:5, each = 3),
  location_2 = rep(1:3, times = 5),
  value = rnorm(15)
)
measure_ndim(m2d)  # 2
```

measure_packs *List Registered Technique Packs*

Description

Returns a tibble of all registered technique packs, including the core `measure` package.

Usage

```
measure_packs()
```

Value

A tibble with columns:

- `name`: Package name
- `technique`: Technique category (e.g., "general", "SEC/GPC")
- `version`: Package version
- `description`: Brief description

See Also

`measure_steps()`, `register_measure_pack()`

Examples

```
measure_packs()
```

---

measure_passing_bablok

Passing-Bablok Regression for Method Comparison

Description

Performs Passing-Bablok regression, a non-parametric method for comparing two analytical methods. This is robust to outliers and does not require normal distribution of residuals.

Usage

```
measure_passing_bablok(
  data,
  method1_col,
  method2_col,
  conf_level = 0.95,
  alpha = 0.05
)
```

Arguments

`data`	A data frame containing paired measurements.
`method1_col`	Name of column for method 1 (reference/comparator).
`method2_col`	Name of column for method 2 (test method).
`conf_level`	Confidence level for intervals. Default is 0.95.
`alpha`	Significance level for CUSUM linearity test. Default is 0.05.

Details

Method:

Passing-Bablok regression:

1. Calculates slopes between all pairs of points
2. Uses median slope as the estimate (robust to outliers)
3. Calculates intercept from median slope
4. Uses non-parametric confidence intervals

CUSUM Linearity Test:

Tests the assumption of linear relationship. If significant ($p < alpha$), the linear model may not be appropriate.

Interpretation:

For equivalent methods:

- 95% CI for slope includes 1
- 95% CI for intercept includes 0

Requirements:

This function requires the `mcr` package. Install with: `install.packages("mcr")`

Value

A `measure_passing_bablok` object containing:

- `coefficients`: Tibble with intercept and slope estimates and CIs
- `linearity`: CUSUM test results for linearity assumption
- `statistics`: Summary statistics

See Also

`measure_bland_altman()`, `measure_deming_regression()`

Other method-comparison: `measure_bland_altman()`, `measure_deming_regression()`, `measure_proficiency_score()`

Examples

```
## Not run:
# Requires mcr package
data <- data.frame(
  reference = c(5.2, 10.5, 15.8, 25.3, 50.1, 75.4, 100.2),
  new_method = c(5.1, 10.8, 16.2, 25.9, 49.8, 76.1, 101.3)
)

result <- measure_passing_bablok(
  data,
  method1_col = "reference",
  method2_col = "new_method"
)
```

```
print(result)

## End(Not run)
```

`measure_plot_summary` *Plot Summary Statistics for Measure Data*

Description

Create a summary plot showing mean +/- standard deviation across all samples at each measurement location.

Usage

```
measure_plot_summary(data, measure_col = NULL, show_range = FALSE)
```

Arguments

`data`	A data frame with a measure column (`.measures`).
`measure_col`	Name of the measure column. If NULL, auto-detected.
`show_range`	Logical. If TRUE, also show min/max range. Default FALSE.

Value

A ggplot2 object.

Examples

```
## Not run:
rec <- recipe(water ~ ., data = meats_long) |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_snv() |>
  prep()

baked <- bake(rec, new_data = NULL)
measure_plot_summary(baked)

## End(Not run)
```

`measure_proficiency_score` *Proficiency Testing Scores*

Description

Calculates proficiency testing scores (z-scores, En scores, or zeta scores) for evaluating laboratory performance in interlaboratory comparisons.

Usage

```
measure_proficiency_score(
  data,
  measured_col,
  reference_col,
  uncertainty_col = NULL,
  reference_uncertainty_col = NULL,
  score_type = c("z_score", "en_score", "zeta_score"),
  sigma = NULL,
  group_col = NULL
)
```

Arguments

`data`	A data frame containing measurement data.
`measured_col`	Name of column with measured/reported values.
`reference_col`	Name of column with reference/assigned values.
`uncertainty_col`	Name of column with measurement uncertainties. Required for En and zeta scores.
`reference_uncertainty_col`	Name of column with reference value uncertainties. Optional for En/zeta scores.
`score_type`	Type of score to calculate: • `"z_score"` (default): (measured - reference) / sigma • `"en_score"`: (measured - reference) / sqrt(U_meas^2 + U_ref^2) • `"zeta_score"`: Similar to En, for correlated uncertainties
`sigma`	Standard deviation for z-score calculation. If NULL, estimated from the data.
`group_col`	Optional grouping column for separate assessments.

Details

Score Interpretation:

| |Score| | Status | Action | |———|————|——| | <= 2 | Satisfactory | None | | 2-3 | Questionable | Review | | > 3 | Unsatisfactory| Investigate |

Score Types:

z-score: Uses a fixed standard deviation (sigma), typically derived from historical data or consensus of participants.

En score: Uses expanded uncertainties of both the lab result and reference value. Appropriate when uncertainties are well-characterized.

zeta score: Similar to En, but accounts for potential correlation between lab and reference uncertainties.

Value

A `measure_proficiency_score` object containing:

- `scores`: Tibble with individual scores and flags
- `statistics`: Summary statistics and counts

See Also

`measure_accuracy()`, `criteria_proficiency_testing()`

Other method-comparison: `measure_bland_altman()`, `measure_deming_regression()`, `measure_passing_bablok()`

Examples

```
# Proficiency testing results from multiple labs
pt_data <- data.frame(
  lab_id = paste0("Lab_", 1:10),
  measured = c(99.2, 100.5, 98.8, 101.2, 97.5, 100.1, 99.8, 102.3, 100.6, 94.0),
  assigned = rep(100, 10),
  uncertainty = c(1.5, 2.0, 1.8, 1.6, 2.2, 1.9, 1.7, 2.1, 1.5, 2.0)
)

# z-scores with known sigma
z_result <- measure_proficiency_score(
  pt_data,
  measured_col = "measured",
  reference_col = "assigned",
  score_type = "z_score",
  sigma = 2.5
)

print(z_result)

# En scores using uncertainties
en_result <- measure_proficiency_score(
  pt_data,
```

```
  measured_col = "measured",
  reference_col = "assigned",
  uncertainty_col = "uncertainty",
  score_type = "en_score"
)

print(en_result)
```

`measure_project` *Project n-dimensional measurement by aggregating across dimensions*

Description

Reduces dimensionality by applying an aggregation function across one or more dimensions.

Usage

```
measure_project(x, along, fn = mean, na_rm = TRUE, ...)
```

Arguments

`x`	A `measure_nd_tbl` or `measure_nd_list` object.
`along`	Integer or character specifying which dimension(s) to aggregate across. Can use dimension numbers or names.
`fn`	Aggregation function. Default is `mean`.
`na_rm`	Logical. Remove NA values before aggregation? Default `TRUE`.
`...`	Additional arguments passed to `fn`.

Value

A `measure_tbl`, `measure_nd_tbl`, `measure_list`, or `measure_nd_list` with reduced dimensionality.

Examples

```
# Create 2D measurement (time x wavelength)
m2d <- new_measure_nd_tbl(
  location_1 = rep(1:5, each = 3),
  location_2 = rep(c(254, 280, 320), times = 5),
  value = rnorm(15, mean = 100),
  dim_names = c("time", "wavelength")
)

# Project across wavelength (average spectrum at each time)
time_trace <- measure_project(m2d, along = 2)

# Project across time (average time profile at each wavelength)
```

```
wavelength_profile <- measure_project(m2d, along = 1)

# Use sum instead of mean
total <- measure_project(m2d, along = 2, fn = sum)
```

`measure_quality_summary`
Summarize measure data quality

Description

Provides a comprehensive quality summary for measure data, including axis information and validation results.

Usage

```
measure_quality_summary(x, verbose = TRUE)
```

Arguments

`x`	A `measure_tbl`, `measure_list`, or data frame with measure column.
`verbose`	Logical; if TRUE, prints summary to console. Default is TRUE.

Value

Invisibly returns a list containing axis info and validation results.

Examples

```
specs <- new_measure_list(list(
  new_measure_tbl(location = seq(1000, 2500, by = 2), value = rnorm(751)),
  new_measure_tbl(location = seq(1000, 2500, by = 2), value = rnorm(751))
))
measure_quality_summary(specs)
```

`measure_repeatability`
Repeatability (Within-Run Precision)

Description

Calculates repeatability statistics for replicate measurements performed under identical conditions (same operator, instrument, short time interval).

Usage

```
measure_repeatability(data, response_col, group_col = NULL, conf_level = 0.95)
```

Arguments

`data`	A data frame containing replicate measurements.
`response_col`	Name of the column containing the response values.
`group_col`	Optional name of a grouping column (e.g., concentration level). If provided, repeatability is calculated within each group.
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Repeatability represents the precision of a method under constant conditions over a short time interval. It is typically assessed using at least 6 replicates of a sample at each concentration level of interest.

The coefficient of variation (CV) is reported as a percentage: `CV = 100 * SD / mean`

Value

A `measure_precision` object containing:

- `mean`: Mean of the replicates
- `sd`: Standard deviation
- `cv`: Coefficient of variation (%)
- `n`: Number of replicates
- `se`: Standard error
- `ci_lower`, `ci_upper`: Confidence interval for the mean

See Also

`measure_intermediate_precision()`, `measure_reproducibility()`

Other precision: `measure_gage_rr()`, `measure_intermediate_precision()`, `measure_reproducibility()`

Examples

```
# Simple repeatability from replicate measurements
data <- data.frame(
  sample_id = rep("QC1", 10),
  concentration = rnorm(10, mean = 100, sd = 2)
)
measure_repeatability(data, "concentration")

# Repeatability at multiple concentration levels
data <- data.frame(
  level = rep(c("low", "mid", "high"), each = 6),
  concentration = c(
    rnorm(6, 10, 0.5),
```

```
    rnorm(6, 50, 2),
    rnorm(6, 100, 4)
  )
)
measure_repeatability(data, "concentration", group_col = "level")
```

`measure_reproducibility` *Reproducibility (Between-Lab Precision)*

Description

Calculates reproducibility statistics for measurements performed at different laboratories.

Usage

```
measure_reproducibility(
  data,
  response_col,
  lab_col,
  group_col = NULL,
  conf_level = 0.95
)
```

Arguments

`data`	A data frame containing measurements from multiple labs.
`response_col`	Name of the column containing the response values.
`lab_col`	Name of the column identifying the laboratory.
`group_col`	Optional grouping column (e.g., concentration level).
`conf_level`	Confidence level for intervals. Default is 0.95.

Details

Reproducibility represents the precision of a method when performed at different laboratories. It includes both within-lab (repeatability) and between-lab variance components.

Value

A `measure_precision` object containing:

- Within-lab variance (repeatability)
- Between-lab variance
- Total reproducibility variance
- Corresponding CV estimates

See Also

measure_repeatability(), measure_intermediate_precision()

Other precision: measure_gage_rr(), measure_intermediate_precision(), measure_repeatability()

Examples

```
# Reproducibility across laboratories
set.seed(123)
data <- data.frame(
  lab_id = rep(c("Lab_A", "Lab_B", "Lab_C"), each = 10),
  concentration = rnorm(30, mean = 100, sd = 2) +
    rep(c(0, 3, -2), each = 10)  # Lab bias
)
measure_reproducibility(data, "concentration", lab_col = "lab_id")
```

---

measure_sample_types *Canonical Sample Types*

Description

The allowed values for the sample_type column in analytical workflows.

Usage

```
measure_sample_types
```

Format

An object of class character of length 5.

---

measure_slice *Extract slices from n-dimensional measurement*

Description

Fixes one or more dimensions at specific coordinate values or ranges, returning a lower-dimensional result.

Usage

```
measure_slice(x, ..., drop = TRUE)
```

Arguments

`x`	A `measure_nd_tbl` or `measure_nd_list` object.
`...`	Named arguments specifying slice conditions. Names should be dimension numbers (e.g., `dim_1 = 5`) or dimension names if set (e.g., `time = 5`). Values can be: • A single value: exact match • A numeric vector: match any of these values • A function: applied to coordinates, should return logical
`drop`	Logical. If `TRUE` (default), dimensions with a single value are dropped from the result. If `FALSE`, they are retained.

Value

A `measure_tbl`, `measure_nd_tbl`, `measure_list`, or `measure_nd_list` depending on the number of remaining dimensions.

Examples

```
# Create a 3D measurement (2 x 3 x 4)
m3d <- new_measure_nd_tbl(
  location_1 = rep(1:2, each = 12),
  location_2 = rep(rep(1:3, each = 4), 2),
  location_3 = rep(1:4, 6),
  value = 1:24,
  dim_names = c("sample", "time", "wavelength")
)

# Extract slice at sample = 1
slice_2d <- measure_slice(m3d, dim_1 = 1)
measure_ndim(slice_2d)  # 2D

# Extract at specific time points
slice_subset <- measure_slice(m3d, dim_2 = c(1, 3))

# Use dimension names
slice_wl <- measure_slice(m3d, wavelength = 2)
```

`measure_standardize_sample_type` *Standardize Sample Type Values*

Description

Converts non-standard sample type values to canonical form using a user-specified mapping. This is useful when data uses different naming conventions (e.g., ”QC”, ”quality_control”, ”pooled_qc”).

Usage

```
measure_standardize_sample_type(
  data,
  col = "sample_type",
  mapping = NULL,
  unknown_action = c("error", "warn", "keep", "unknown")
)
```

Arguments

`data`	A data frame containing a sample type column.
`col`	Name of the sample type column. Default is `"sample_type"`.
`mapping`	A named list mapping canonical types to vectors of aliases. For example: `list(qc = c("QC", "quality_control", "pooled_qc"))`. If NULL, uses default case-insensitive matching.
`unknown_action`	What to do with values that don't match any mapping: • `"error"` (default): Stop with error • `"warn"`: Warn and keep original value • `"keep"`: Silently keep original value • `"unknown"`: Convert to "unknown"

Value

The data frame with standardized sample_type values.

Examples

```
# Data with non-standard sample types
data <- data.frame(
  sample_id = 1:5,
  sample_type = c("QC", "STD", "BLK", "UNK", "REF")
)

# Standardize with custom mapping
measure_standardize_sample_type(
  data,
  mapping = list(
    qc = c("QC", "qc", "quality_control"),
    standard = c("STD", "std", "cal"),
    blank = c("BLK", "blk", "blank"),
    unknown = c("UNK", "unk", "sample"),
    reference = c("REF", "ref")
  )
)
```

`measure_steps` *List Available Steps*

Description

Returns a tibble of all registered recipe steps from measure and any loaded technique packs. Results can be filtered by pack, category, or technique.

Usage

```
measure_steps(packs = NULL, categories = NULL, techniques = NULL)
```

Arguments

`packs`	Character vector of pack names to include. If `NULL`, includes all packs.
`categories`	Character vector of step categories to include. If `NULL`, includes all categories.
`techniques`	Character vector of techniques to include. If `NULL`, includes all techniques.

Value

A tibble with columns:

- `step_name`: Function name (e.g., "step_measure_baseline_als")
- `pack_name`: Source package name
- `category`: Step category (e.g., "baseline", "smoothing")
- `description`: Brief description
- `technique`: Technique (e.g., "general", "SEC/GPC")

See Also

`measure_packs()`, `register_measure_step()`

Examples

```
# List all steps
measure_steps()

# List only baseline correction steps
measure_steps(categories = "baseline")

# List steps from a specific technique pack
measure_steps(techniques = "SEC/GPC")
```

`measure_summarize` *Summarize Measurements Across Samples*

Description

`measure_summarize()` computes summary statistics for each measurement location across all samples. This is useful for understanding your data, computing reference spectra, or identifying outliers.

Usage

```
measure_summarize(
  .data,
  .cols = NULL,
  .fns = list(mean = mean, sd = stats::sd),
  na.rm = TRUE
)
```

Arguments

`.data`	A data frame containing one or more `measure_list` columns.
`.cols`	<tidy-select> Columns to summarize. Defaults to all `measure_list` columns.
`.fns`	A named list of summary functions. Each function should accept a numeric vector and return a single value. Default is `list(mean = mean, sd = sd)`.
`na.rm`	Logical. Should NA values be removed? Default is `TRUE`.

Details

This function does NOT transform data; it summarizes it. Common uses:

- **Mean spectrum**: The average spectrum across all samples
- **Reference spectrum**: For MSC-style corrections
- **Variability**: Standard deviation at each wavelength
- **Quality control**: Identify problematic wavelength regions

Value

A tibble with one row per measurement location and columns for each summary statistic.

Examples

```
library(recipes)
library(ggplot2)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

baked_data <- bake(rec, new_data = NULL)

# Compute mean and SD at each wavelength
summary_stats <- measure_summarize(baked_data)
summary_stats

# Visualize mean spectrum with confidence band
ggplot(summary_stats, aes(x = location)) +
  geom_ribbon(aes(ymin = mean - sd, ymax = mean + sd), alpha = 0.3) +
  geom_line(aes(y = mean)) +
  labs(x = "Channel", y = "Transmittance", title = "Mean Spectrum +/- 1 SD")

# Custom summary functions
measure_summarize(
  baked_data,
  .fns = list(
    median = median,
    q25 = function(x) quantile(x, 0.25),
    q75 = function(x) quantile(x, 0.75)
  )
)
```

---

measure_system_suitability

System Suitability Check

Description

Performs system suitability tests on QC or reference samples to verify instrument performance meets requirements.

Usage

```
measure_system_suitability(
  data,
  metrics,
  sample_type_col = NULL,
  sst_type = "sst"
)
```

Arguments

`data`	A data frame containing system suitability data.
`metrics`	Named list of columns and their acceptance criteria. Each element should be a list with `col`, `min`, and/or `max`.
`sample_type_col`	Optional column identifying sample types.
`sst_type`	Value in sample_type_col that identifies SST samples.

Details

System suitability testing (SST) verifies that the analytical system is performing adequately before, during, or after a run. Common metrics include:

- Peak resolution
- Retention time reproducibility
- Peak symmetry/tailing factor
- Signal-to-noise ratio
- Plate count

Value

A `measure_sst` object containing:

- `results`: Pass/fail status for each metric
- `summary`: Overall pass/fail and summary statistics
- `details`: Individual sample results

See Also

Other control-charts: `measure_control_chart()`, `measure_control_limits()`

Examples

```
# System suitability check
sst_data <- data.frame(
  sample_id = paste0("SST_", 1:5),
  resolution = c(2.1, 2.3, 2.2, 2.0, 2.1),
  tailing = c(1.1, 1.0, 1.2, 1.1, 1.0),
  plates = c(5200, 5100, 5300, 5000, 5150)
)

result <- measure_system_suitability(
  sst_data,
  metrics = list(
    resolution = list(col = "resolution", min = 2.0),
    tailing = list(col = "tailing", max = 1.5),
    plates = list(col = "plates", min = 5000)
  )
)
print(result)
```

`measure_uncertainty` *Quick Uncertainty Calculation*

Description

A convenience function that returns just the key uncertainty values without the full budget object.

Usage

```
measure_uncertainty(..., .list = NULL, k = 2)
```

Arguments

`...`	`uncertainty_component()` objects to include in the budget.
`.list`	Optional list of uncertainty components.
`k`	Coverage factor for expanded uncertainty. Default is 2 (approximately 95% coverage for normal distribution).

Value

A named list with:

- `combined_u`: Combined standard uncertainty
- `expanded_U`: Expanded uncertainty
- `effective_df`: Effective degrees of freedom
- `coverage_factor`: Coverage factor used

Examples

```
u1 <- uncertainty_component("A", 0.05, type = "A", df = 9)
u2 <- uncertainty_component("B", 0.03, type = "B")

measure_uncertainty(u1, u2)
```

`measure_uncertainty_budget` *Create an Uncertainty Budget*

Description

Combines multiple uncertainty components into a complete uncertainty budget following ISO GUM methodology. Calculates combined standard uncertainty, effective degrees of freedom (Welch-Satterthwaite), and expanded uncertainty.

Usage

```
measure_uncertainty_budget(..., .list = NULL, k = 2, result_value = NULL)
```

Arguments

`...`	`uncertainty_component()` objects to include in the budget.
`.list`	Optional list of uncertainty components.
`k`	Coverage factor for expanded uncertainty. Default is 2 (approximately 95% coverage for normal distribution).
`result_value`	Optional. The measurement result value, used for calculating relative uncertainty.

Details

Combined Standard Uncertainty:

Calculated as the root sum of squares of contributions:

$$u_c = \sqrt{\sum_i (c_i \cdot u_i)^2}$$

Welch-Satterthwaite Effective Degrees of Freedom:

$$\nu_{eff} = \frac{u_c^4}{\sum_i \frac{(c_i \cdot u_i)^4}{\nu_i}}$$

This is used to determine the appropriate coverage factor for a given confidence level.

Expanded Uncertainty:

$$U = k \cdot u_c$$

With k=2, this provides approximately 95% coverage.

Value

A `measure_uncertainty_budget` object containing:

- `components`: List of input uncertainty components
- `combined_u`: Combined standard uncertainty
- `effective_df`: Effective degrees of freedom (Welch-Satterthwaite)
- `coverage_factor`: The k value used
- `expanded_U`: Expanded uncertainty (k * combined_u)
- `result_value`: The measurement result (if provided)
- `relative_u`: Relative standard uncertainty (if result provided)

See Also

`uncertainty_component()` for creating components, `tidy.measure_uncertainty_budget()` for extracting results, `autoplot.measure_uncertainty_budget()` for visualization.

Examples

```
# Create components
u_repeat <- uncertainty_component("Repeatability", 0.05, type = "A", df = 9)
u_cal <- uncertainty_component("Calibrator", 0.02, type = "B", df = 50)
u_temp <- uncertainty_component("Temperature", 0.03, type = "B")

# Create budget
budget <- measure_uncertainty_budget(u_repeat, u_cal, u_temp, k = 2)
print(budget)

# With result value for relative uncertainty
budget <- measure_uncertainty_budget(
  u_repeat, u_cal, u_temp,
  result_value = 10.5
)
```

`measure_unfold` *Unfold n-dimensional measurement to 1D*

Description

Converts an n-dimensional measurement to a 1D vector by flattening according to a specified dimension order. Stores metadata needed to reconstruct the original nD structure via `measure_fold()`.

Usage

```
measure_unfold(x, order = NULL)
```

Arguments

`x`	A `measure_nd_tbl` or `measure_nd_list` object.
`order`	Integer vector specifying the order of dimensions for unfolding. Default is `NULL`, which uses the natural order (1, 2, ..., n). The first dimension varies fastest.

Details

Unfolding is useful for:

- Applying 1D modeling techniques (PCA, PLS) to nD data
- Exporting to formats that expect 1D vectors
- Visualization as a single trace

The fold metadata includes:

- `ndim`: Original number of dimensions
- `dim_names`, `dim_units`: Original dimension metadata
- `coordinates`: The original coordinate values for each dimension
- `order`: The unfolding order used

Value

A `measure_tbl` or `measure_list` with an attribute `"fold_info"` containing the metadata needed to reconstruct the nD structure.

See Also

`measure_fold()` to reconstruct the nD structure

Examples

```
# Create a 2D measurement (3 x 4 grid)
m2d <- new_measure_nd_tbl(
  location_1 = rep(1:3, each = 4),
  location_2 = rep(1:4, times = 3),
  value = 1:12,
  dim_names = c("time", "wavelength")
)

# Unfold to 1D
m1d <- measure_unfold(m2d)
m1d

# Reconstruct
m2d_restored <- measure_fold(m1d)
```

`measure_validate_metadata` *Validate Analytical Metadata*

Description

Validates that a data frame contains the required metadata columns for analytical workflows. This function checks for column presence, correct data types, and valid values (e.g., sample_type levels).

Usage

```
measure_validate_metadata(
  data,
  require = NULL,
  sample_types = measure_sample_types,
  action = c("error", "warn", "message")
)
```

Arguments

`data`	A data frame to validate.
`require`	Character vector of required columns. Common columns include: • `"sample_type"`: Sample classification (qc, standard, blank, unknown, reference) • `"run_order"`: Injection/measurement sequence (integer) • `"batch_id"`: Batch identifier (character/factor) • `"nominal_conc"`: Known concentration for standards (numeric) • `"sample_id"`: Unique sample identifier • `"analyst_id"`, `"day"`, `"instrument_id"`: Precision study factors
`sample_types`	Allowed values for `sample_type` column. Default is measure_sample_types: "qc", "standard", "blank", "unknown", "reference".
`action`	What to do when validation fails: • `"error"` (default): Stop with an informative error • `"warn"`: Issue warnings but continue • `"message"`: Issue messages but continue

Details

Canonical Columns:

Milestone 2 functions expect specific column names with specific types:

Column	Type	Description
`sample_type`	character/factor	Sample classification

run_order	integer	Injection sequence within batch
batch_id	character/factor	Batch identifier
nominal_conc	numeric	Known concentration (standards)
sample_id	character/factor	Unique sample identifier
analyst_id	character/factor	Analyst performing measurement
day	character/Date	Day of measurement
instrument_id	character/factor	Instrument identifier
dilution_factor	numeric	Sample dilution factor

Sample Type Values:

The `sample_type` column must contain only values from measure_sample_types:

- `"qc"`: Quality control sample (pooled QC, system suitability)
- `"standard"`: Calibration standard with known concentration
- `"blank"`: Blank sample (solvent, matrix blank)
- `"unknown"`: Sample with unknown concentration
- `"reference"`: Reference material for batch correction

Value

Invisibly returns a list with validation results:

- `valid`: Logical, TRUE if all checks passed
- `checks`: List of individual check results
- `data`: The original data (unchanged)

See Also

`measure_standardize_sample_type()` for converting non-standard sample type values to canonical form.

Examples

```
# Create sample analytical data
data <- data.frame(
  sample_id = paste0("S", 1:10),
  sample_type = c("qc", "standard", "standard", "unknown", "unknown",
                  "unknown", "qc", "blank", "unknown", "qc"),
  run_order = 1:10,
  batch_id = "B001",
  nominal_conc = c(NA, 10, 50, NA, NA, NA, NA, 0, NA, NA),
  response = rnorm(10, mean = 100)
)

# Validate required columns
measure_validate_metadata(data, require = c("sample_type", "run_order"))

# Validate for calibration workflow
measure_validate_metadata(
```

```
  data,
  require = c("sample_type", "nominal_conc")
)

# More lenient validation (warnings only)
measure_validate_metadata(
  data,
  require = c("sample_type", "run_order", "missing_col"),
  action = "warn"
)
```

`measure_validation_report` *Create an Analytical Method Validation Report*

Description

Creates a structured validation report object that collects results from various validation studies (calibration, precision, accuracy, etc.) and can be rendered to HTML, PDF, or Word formats using standardized templates.

This function supports two major validation frameworks:

- **ICH Q2(R2)**: International harmonized guidelines for analytical validation
- **USP <1225>**: United States Pharmacopeia compendial validation procedures

Usage

```
measure_validation_report(
  title = "Analytical Method Validation Report",
  method_name = NULL,
  method_description = NULL,
  analyst = NULL,
  reviewer = NULL,
  lab = NULL,
  date = Sys.Date(),
  instrument = NULL,
  software = NULL,
  calibration = NULL,
  lod_loq = NULL,
  accuracy = NULL,
  precision = NULL,
  linearity = NULL,
  range = NULL,
  specificity = NULL,
  robustness = NULL,
  carryover = NULL,
  system_suitability = NULL,
  uncertainty = NULL,
```

```
  method_comparison = NULL,
  stability = NULL,
  criteria = NULL,
  conclusions = NULL,
  references = NULL,
  appendices = NULL,
  ...
)
```

Arguments

`title`	Report title. Default: "Analytical Method Validation Report"
`method_name`	Name of the analytical method being validated.
`method_description`	Brief description of the method (technique, analyte, matrix).
`analyst`	Name of the analyst(s) performing validation.
`reviewer`	Name of the reviewer (optional).
`lab`	Laboratory name or identifier.
`date`	Date of the validation study. Default: current date.
`instrument`	Instrument details (name, model, serial number).
`software`	Software used for data acquisition/processing.
`calibration`	A `measure_calibration` object from `measure_calibration_fit()`.
`lod_loq`	LOD/LOQ results from `measure_lod()`, `measure_loq()`, or `measure_lod_loq()`. Can be a single object or a list.
`accuracy`	Accuracy results from `measure_accuracy()`.
`precision`	A list containing precision study results: • `repeatability`: from `measure_repeatability()` • `intermediate`: from `measure_intermediate_precision()` • `reproducibility`: from `measure_reproducibility()` (optional)
`linearity`	Linearity results from `measure_linearity()`.
`range`	A list with `lower` and `upper` validated range limits, or results supporting range determination.
`specificity`	User-provided specificity/selectivity assessment. Can be text, a data frame of interference results, or a list.
`robustness`	User-provided robustness study results. Can be text, a data frame, or structured results.
`carryover`	Carryover results from `measure_carryover()`.
`system_suitability`	System suitability results from `measure_system_suitability()`.
`uncertainty`	Uncertainty budget from `measure_uncertainty_budget()`.
`method_comparison`	Method comparison results (Bland-Altman, Deming, Passing-Bablok) from the corresponding functions.

`stability`	User-provided stability data (solution stability, freeze-thaw, etc.).
`criteria`	A `measure_criteria` object defining acceptance criteria, or a named list of criteria objects for different sections.
`conclusions`	User-provided conclusions text or a list with `summary` and `recommendations`.
`references`	Character vector of references cited.
`appendices`	Named list of additional content to include as appendices.
`...`	Additional metadata to include in the report.

Details

Workflow:

1. Run individual validation studies using measure functions
2. Collect results into a validation report object
3. Render to desired format using `render_validation_report()`

Supported Validation Characteristics (ICH Q2):

- **Specificity/Selectivity**: Ability to assess analyte in presence of interferences
- **Linearity**: Proportional response over concentration range
- **Range**: Validated concentration interval
- **Accuracy**: Closeness to true value (trueness)
- **Precision**: Repeatability, intermediate precision, reproducibility
- **Detection Limit (LOD)**: Lowest detectable amount
- **Quantitation Limit (LOQ)**: Lowest quantifiable amount with acceptable precision/accuracy
- **Robustness**: Capacity to remain unaffected by small method variations

Data Provenance:

The report automatically captures:

- R version and package versions
- Date/time of report generation
- Function calls used to generate each section

Value

A `measure_validation_report` object containing:

- `metadata`: Report metadata (title, analyst, date, etc.)
- `sections`: Named list of validation results by section
- `criteria`: Acceptance criteria used
- `provenance`: Data provenance and computational environment info
- `call`: The function call

See Also

render_validation_report() to generate the final report document.

Related validation functions:

- measure_calibration_fit(), measure_calibration_predict()
- measure_lod(), measure_loq(), measure_lod_loq()
- measure_accuracy(), measure_linearity(), measure_carryover()
- measure_repeatability(), measure_intermediate_precision()
- measure_uncertainty_budget()

Examples

```
# Create sample validation data
set.seed(123)
cal_data <- data.frame(
  nominal_conc = rep(c(1, 5, 10, 25, 50, 100), each = 3),
  response = c(1, 5, 10, 25, 50, 100) * 1000 +
    rnorm(18, sd = 50),
  sample_type = "standard"
)

# Fit calibration
cal_fit <- measure_calibration_fit(
  cal_data,
  formula = response ~ nominal_conc,
  weights = "1/x"
)

# Calculate LOD/LOQ (requires sample_type column)
blank_data <- data.frame(
  response = rnorm(10, mean = 50, sd = 15),
  sample_type = "blank"
)
lod_result <- measure_lod(blank_data, response_col = "response")

# Create precision data
precision_data <- data.frame(
  concentration = rep(c(10, 50, 100), each = 6),
  replicate = rep(1:6, 3),
  response = c(
    rnorm(6, 10000, 200),
    rnorm(6, 50000, 800),
    rnorm(6, 100000, 1500)
  )
)
repeatability <- measure_repeatability(
  precision_data,
  response_col = "response",
  group_col = "concentration"
)
```

```
# Create validation report
report <- measure_validation_report(
  title = "Validation of HPLC Method for Compound X",
  method_name = "HPLC-UV Assay",
  method_description = "Reversed-phase HPLC with UV detection at 254 nm",
  analyst = "J. Smith",
  lab = "Analytical Development Lab",
  calibration = cal_fit,
  lod_loq = lod_result,
  precision = list(repeatability = repeatability),
  conclusions = "Method meets all acceptance criteria for intended use."
)

print(report)
```

`meats_long` *Fat, water and protein content of meat samples*

Description

"These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 - 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents.

Details

If results from these data are used in a publication we want you to mention the instrument and company name (Tecator) in the publication. In addition, please send a preprint of your article to

Karin Thente, Tecator AB, Box 70, S-263 21 Hoganas, Sweden

The data are available in the public domain with no responsibility from the original data source. The data can be redistributed as long as this permission note is attached."

"For each meat sample the data consists of a 100 channel spectrum of absorbances and the contents of moisture (water), fat and protein. The absorbance is -log10 of the transmittance measured by the spectrometer. The three contents, measured in percent, are determined by analytic chemistry."

Included here are the meats data transformed to a long format with

```
modeldata::meats |>
  rowid_to_column(var = "id") |>
  pivot_longer(cols = starts_with("x_"),
               names_to = "channel",
               values_to = "transmittance") |>
  mutate(channel = str_extract(channel, "[:digit:]+") |> as.integer())
```

Value

`meats_long` a tibble

Examples

```
data(meats_long)
str(meats_long)
```

---

new_measure_list *Create a new measure list*

Description

Constructor for creating a collection of measurements suitable for use as a list column in a data frame. Each element should be a `measure_tbl` or tibble with `location` and `value` columns.

Usage

```
new_measure_list(x = list())
```

Arguments

`x`	A list of `measure_tbl` objects or tibbles with `location` and `value` columns.

Value

A list with class `measure_list`.

See Also

`new_measure_tbl()` for creating individual measurements, `is_measure_list()` for checking object class.

Examples

```
# Create individual spectra
spec1 <- new_measure_tbl(location = 1:10, value = rnorm(10))
spec2 <- new_measure_tbl(location = 1:10, value = rnorm(10))

# Combine into a measure_list
specs <- new_measure_list(list(spec1, spec2))
specs
```

`new_measure_nd_list` *Create a new n-dimensional measure list*

Description

Constructor for creating a collection of n-dimensional measurements suitable for use as a list column in a data frame. Each element should be a `measure_nd_tbl` or tibble with `location_*` and `value` columns.

Usage

```
new_measure_nd_list(x = list())
```

Arguments

`x`	A list of `measure_nd_tbl` objects or tibbles with `location_*` and `value` columns.

Value

A list with class `measure_nd_list`.

See Also

`new_measure_nd_tbl()` for creating individual nD measurements, `is_measure_nd_list()` for checking object class.

Examples

```
# Create individual 2D measurements
meas1 <- new_measure_nd_tbl(
  location_1 = rep(1:5, each = 3),
  location_2 = rep(1:3, times = 5),
  value = rnorm(15)
)
meas2 <- new_measure_nd_tbl(
  location_1 = rep(1:5, each = 3),
  location_2 = rep(1:3, times = 5),
  value = rnorm(15)
)

# Combine into a measure_nd_list
meas_list <- new_measure_nd_list(list(meas1, meas2))
meas_list
```

`new_measure_nd_tbl` *Create a new n-dimensional measure tibble*

Description

Constructor for creating a single n-dimensional measurement object containing location coordinates (e.g., wavelength, retention time) and values.

Usage

```
new_measure_nd_tbl(..., value = double(), dim_names = NULL, dim_units = NULL)
```

Arguments

`...`	Named location vectors. Names should follow the pattern `location_1`, `location_2`, etc. Each must be a numeric vector of the same length.
`value`	Numeric vector of measurement values (e.g., absorbance, intensity, signal). Must have the same length as location vectors.
`dim_names`	Optional character vector of semantic dimension names (e.g., `c("wavelength", "retention_time")`).
`dim_units`	Optional character vector of dimension units (e.g., `c("nm", "min")`).

Value

A tibble with class `measure_nd_tbl` containing `location_1`, `location_2`, ..., `location_n`, and `value` columns. Attributes include `ndim`, `dim_names`, `dim_units`, and `dim_order`.

See Also

`new_measure_nd_list()` for creating collections of nD measurements, `is_measure_nd_tbl()` for checking object class, `measure_ndim()` for getting dimensionality.

Examples

```
# Create a 2D measurement (e.g., LC-UV: retention time x wavelength)
meas_2d <- new_measure_nd_tbl(
  location_1 = rep(seq(0, 10, length.out = 5), each = 3),
  location_2 = rep(c(254, 280, 320), times = 5),
  value = rnorm(15),
  dim_names = c("retention_time", "wavelength"),
  dim_units = c("min", "nm")
)
meas_2d
```

new_measure_tbl *Create a new measure tibble*

Description

Constructor for creating a single measurement object containing location (e.g., wavelength, retention time) and value pairs.

Usage

```
new_measure_tbl(location = double(), value = double())
```

Arguments

location	Numeric vector of measurement locations (e.g., wavelengths, wavenumbers, retention times).
value	Numeric vector of measurement values (e.g., absorbance, intensity, signal).

Value

A tibble with class `measure_tbl` containing `location` and `value` columns.

See Also

`new_measure_list()` for creating collections of measurements, `is_measure_tbl()` for checking object class.

Examples

```
# Create a simple spectrum
spec <- new_measure_tbl(
  location = seq(1000, 1100, by = 10),
  value = sin(seq(1000, 1100, by = 10) / 50)
)
spec
```

new_peak_model *Create a Peak Model Object*

Description

Creates a new peak model S3 object. This is the base constructor for all peak shape models used in deconvolution.

Usage

```
new_peak_model(
  name,
  n_params,
  param_names,
  description = "",
  technique = NULL,
  ...
)
```

Arguments

`name`	Character name of the model (e.g., "gaussian", "emg").
`n_params`	Number of parameters in the model.
`param_names`	Character vector of parameter names.
`description`	Brief description of the model.
`technique`	Optional technique name (e.g., "SEC/GPC"). If `NULL`, model is general-purpose.
`...`	Additional model-specific attributes.

Value

A `peak_model` S3 object with subclass `{name}_peak_model`.

See Also

`peak_model_value()`, `peak_model_gradient()`, `peak_model_bounds()`

Examples

```
# Create a simple Gaussian model
model <- new_peak_model(
  name = "gaussian",
  n_params = 3,
  param_names = c("height", "center", "width"),
  description = "Symmetric Gaussian peak"
)
print(model)
```

`optimize_deconvolution` *Optimize Peak Deconvolution*

Description

Finds optimal parameters for a set of peak models by minimizing the sum of squared residuals between the observed and fitted values.

Usage

```
optimize_deconvolution(
  x,
  y,
  models,
  init_params,
  optimizer = "auto",
  max_iter = 1000L,
  tol = 1e-06,
  constrain_positions = TRUE,
  ...
)
```

Arguments

`x`	Numeric vector of x-axis values (e.g., retention time, wavelength).
`y`	Numeric vector of observed y-axis values.
`models`	List of `peak_model` objects, one per peak.
`init_params`	List of initial parameter lists, one per peak.
`optimizer`	Optimization method: `"auto"`, `"lbfgsb"`, `"multistart"`, or `"nelder_mead"`.
`max_iter`	Maximum number of iterations.
`tol`	Convergence tolerance.
`constrain_positions`	Logical. If `TRUE`, enforce that peak centers maintain their relative ordering.
`...`	Additional arguments passed to specific optimizers.

Value

A list containing:

- `parameters`: List of optimized parameter lists
- `fitted_values`: Numeric vector of fitted y values
- `residuals`: Numeric vector of residuals

- `convergence`: Logical indicating convergence
- `n_iterations`: Number of iterations used
- `final_value`: Final objective function value (SSE)
- `optimizer`: Name of optimizer used
- `elapsed_time`: Optimization time in seconds

See Also

Other peak-deconvolution: `add_param_jitter()`, `assess_deconv_quality()`, `check_quality_gates()`, `initialize_peak_params()`

Examples

```
# Create synthetic data with two overlapping Gaussian peaks
x <- seq(0, 20, by = 0.1)
true_y <- 1.5 * exp(-0.5 * ((x - 8) / 1)^2) +
  0.8 * exp(-0.5 * ((x - 12) / 1.5)^2)
y <- true_y + rnorm(length(x), sd = 0.05)

# Set up models and initial guesses
models <- list(gaussian_peak_model(), gaussian_peak_model())
init_params <- list(
  list(height = 1.2, center = 7.5, width = 1.2),
  list(height = 0.6, center = 12.5, width = 1.8)
)

# Optimize
result <- optimize_deconvolution(x, y, models, init_params)
print(result$parameters)
```

---

outlier_threshold *Parameters for quality control steps*

Description

`outlier_threshold()` controls the threshold for outlier detection (in standard deviation or Mahalanobis distance units).

Usage

```
outlier_threshold(range = c(2, 5), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
outlier_threshold()
```

---

peak_algorithms *List Available Peak Detection Algorithms*

Description

Returns a tibble of all registered peak detection algorithms.

Usage

```
peak_algorithms(packs = NULL, techniques = NULL)
```

Arguments

`packs`	Character vector of pack names to include. If `NULL`, includes all packs.
`techniques`	Character vector of techniques to include. If `NULL`, includes all techniques (including general-purpose algorithms).

Value

A tibble with columns:

- `name`: Algorithm name (e.g., "prominence", "derivative")
- `pack_name`: Source package name
- `description`: Brief description
- `technique`: Technique (or `NA` for general-purpose)
- `default_params`: List column of default parameter values

See Also

`register_peak_algorithm()`, `get_peak_algorithm()`

Examples

```
# List all algorithms
peak_algorithms()

# List only algorithms from a specific pack
peak_algorithms(packs = "measure")
```

---

peak_location_min *Parameters for peak normalization*

Description

`peak_location_min()` and `peak_location_max()` define the bounds for the reference region in peak normalization. These should be specified in the same units as the location values in your measurement data.

Usage

```
peak_location_min(range = c(0, 100), trans = NULL)

peak_location_max(range = c(0, 100), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
peak_location_min()
peak_location_max()
```

`peak_model_area` *Calculate Peak Area*

Description

Integrates the peak model over a given range to calculate the area.

Usage

```
peak_model_area(model, params, x_range = NULL)
```

Arguments

`model`	A `peak_model` object.
`params`	Named list of model parameters.
`x_range`	Numeric vector of length 2 giving the integration range. If `NULL`, integrates over the full domain (may require analytical solution).

Details

For models with analytical integrals (e.g., Gaussian), this can return an exact value. Otherwise, numerical integration is used.

Value

Numeric scalar giving the peak area.

See Also

`peak_model_value()`

Examples

```
model <- create_peak_model("gaussian")
params <- list(height = 1, center = 5, width = 1)
area <- peak_model_area(model, params, c(0, 10))
area
```

`peak_model_bounds` *Get Parameter Bounds for Optimization*

Description

Returns lower and upper bounds for each parameter, used to constrain optimization during deconvolution.

Usage

```
peak_model_bounds(model, x_range, y_range)
```

Arguments

`model`	A `peak_model` object.
`x_range`	Numeric vector of length 2 giving the x-axis range (min, max).
`y_range`	Numeric vector of length 2 giving the y-axis range (min, max).

Value

A list with two components:

- `lower`: Named numeric vector of lower bounds
- `upper`: Named numeric vector of upper bounds

See Also

`peak_model_initial_guess()`

Examples

```
model <- create_peak_model("gaussian")
bounds <- peak_model_bounds(model, c(0, 20), c(0, 100))
bounds$lower
bounds$upper
```

peak_model_gradient *Calculate Peak Model Gradient*

Description

Calculates partial derivatives of the model with respect to each parameter. Used by optimization algorithms for gradient-based fitting.

Usage

```
peak_model_gradient(model, x, params)
```

Arguments

model	A peak_model object.
x	Numeric vector of x values.
params	Named list of model parameters.

Details

If no analytical gradient is available, a numerical gradient can be computed using finite differences. See peak_model_gradient_numerical().

Value

Matrix of partial derivatives with dimensions (length(x), n_params). Column names correspond to parameter names.

See Also

peak_model_value(), peak_model_gradient_numerical()

peak_model_gradient_numerical *Numerical Gradient for Peak Model*

Description

Computes the gradient numerically using finite differences. This is used as a fallback when no analytical gradient is defined.

Usage

```
peak_model_gradient_numerical(model, x, params, eps = 1e-08)
```

Arguments

`model`	A `peak_model` object.
`x`	Numeric vector of x values.
`params`	Named list of model parameters.
`eps`	Step size for finite differences. Default is `1e-8`.

Value

Matrix of partial derivatives with dimensions `(length(x), n_params)`.

`peak_model_initial_guess`

Generate Initial Parameter Guess

Description

Estimates initial parameter values from the data, providing a starting point for optimization.

Usage

```
peak_model_initial_guess(model, x, y, peak_idx)
```

Arguments

`model`	A `peak_model` object.
`x`	Numeric vector of x values.
`y`	Numeric vector of y values (signal intensity).
`peak_idx`	Integer index of the peak maximum in `x` and `y`.

Details

A good initial guess is crucial for successful optimization. The method should estimate parameters from local features of the data (peak height, width at half maximum, asymmetry, etc.).

Value

Named list of initial parameter values.

See Also

`peak_model_bounds()`

Examples

```
model <- create_peak_model("gaussian")
x <- seq(0, 10, by = 0.1)
y <- dnorm(x, mean = 5, sd = 1)
peak_idx <- which.max(y)
initial <- peak_model_initial_guess(model, x, y, peak_idx)
initial
```

---

peak_model_param_names *Get Parameter Names from Peak Model*

Description

Get Parameter Names from Peak Model

Usage

```
peak_model_param_names(model)
```

Arguments

model	A peak_model object.

Value

Character vector of parameter names.

---

peak_model_value *Evaluate Peak Model*

Description

Evaluates the peak model at given x values with specified parameters.

Usage

```
peak_model_value(model, x, params)
```

Arguments

model	A peak_model object.
x	Numeric vector of x values (e.g., retention time, wavelength).
params	Named list of model parameters.

Value

Numeric vector of y values (same length as `x`).

See Also

`peak_model_gradient()`, `peak_model_area()`

Examples

```
# Using a registered Gaussian model
model <- create_peak_model("gaussian")
x <- seq(0, 10, by = 0.1)
params <- list(height = 1, center = 5, width = 1)
y <- peak_model_value(model, x, params)
plot(x, y, type = "l")
```

---

peak_models *List Available Peak Models*

Description

Returns a tibble of all registered peak models.

Usage

```
peak_models(packs = NULL, techniques = NULL)
```

Arguments

`packs`	Character vector of pack names to filter by. If `NULL`, includes all packs.
`techniques`	Character vector of techniques to filter by. If `NULL`, includes all (including general-purpose models).

Value

A tibble with columns: name, pack_name, description, technique.

See Also

`register_peak_model()`, `create_peak_model()`

Examples

```
peak_models()
```

`plot_measure_comparison` *Compare Multiple Preprocessing Recipes*

Description

Visualize the effect of different preprocessing recipes side-by-side. Useful for comparing different parameter settings or preprocessing strategies.

Usage

```
plot_measure_comparison(..., data = NULL, n_samples = 5, summary_only = FALSE)
```

Arguments

`...`	Named recipe objects to compare. Each must be a prepped recipe.
`data`	Data to apply recipes to. If NULL, uses the training data from the first recipe.
`n_samples`	Number of samples to show. Default 5.
`summary_only`	If TRUE, only show summary statistics (mean +/- SD). Default FALSE shows individual spectra.

Value

A ggplot2 object with faceted comparison.

Examples

```
## Not run:
library(recipes)
library(ggplot2)

# Compare SNV vs MSC preprocessing
base_rec <- recipe(water ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel))

snv_rec <- base_rec |>
  step_measure_snv() |>
  prep()

msc_rec <- base_rec |>
  step_measure_msc() |>
  prep()

plot_measure_comparison(
  "SNV" = snv_rec,
  "MSC" = msc_rec,
```

```
  n_samples = 10
)

## End(Not run)
```

`print.measure_validation_report` *Print a Validation Report*

Description

Displays a formatted summary of a validation report object, including metadata, section status, conclusions, and provenance information.

Usage

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

Arguments

`x`	A `measure_validation_report` object.
`...`	Additional arguments (currently ignored).

Value

Invisibly returns the input object.

Examples

```
report <- measure_validation_report(
  title = "Test Report",
  method_name = "HPLC Assay",
  analyst = "J. Smith"
)
print(report)
```

`register_measure_pack` *Register a Technique Pack*

Description

Registers an external technique pack with the measure package. This function should be called from the `.onLoad()` function of technique pack packages.

Usage

```
register_measure_pack(pack_name, technique, version = NULL, description = NULL)
```

Arguments

`pack_name`	Package name (e.g., `"measure.sec"`). Use `pkgname` from `.onLoad()` for portability.
`technique`	Technique name (e.g., `"SEC/GPC"`, `"FTIR"`, `"Raman"`).
`version`	Package version. If `NULL`, attempts to retrieve from installed package.
`description`	Brief description of the technique pack.

Value

Invisible `TRUE`.

See Also

`register_measure_step()`, `measure_packs()`

Examples

```
## Not run:
# In a technique pack's R/zzz.R file:
.onLoad <- function(libname, pkgname) {
  if (requireNamespace("measure", quietly = TRUE)) {
    measure::register_measure_pack(
      pack_name = pkgname,
      technique = "SEC/GPC",
      description = "Size Exclusion Chromatography"
    )
  }
}

## End(Not run)
```

`register_measure_step` *Register a Step from a Technique Pack*

Description

Registers a recipe step with the measure package. This function should be called from the `.onLoad()` function of technique pack packages after registering the pack with `register_measure_pack()`.

Usage

```
register_measure_step(
  step_name,
  pack_name,
  category = "processing",
  description = "",
  technique = NULL
)
```

Arguments

`step_name`	Full step function name (e.g., `"step_sec_mw_averages"`).
`pack_name`	Source package name. Use `pkgname` from `.onLoad()`.
`category`	Step category (e.g., `"preprocessing"`, `"calculation"`).
`description`	Brief description of what the step does.
`technique`	Technique name. If `NULL`, inherits from the registered pack.

Details

Registration is idempotent: calling this function multiple times with the same `pack_name` and `step_name` will update rather than duplicate the entry.

Value

Invisible `TRUE`.

See Also

`register_measure_pack()`, `measure_steps()`

Examples

```
## Not run:
# In a technique pack's R/zzz.R file:
measure::register_measure_step(
  step_name = "step_sec_mw_averages",
  pack_name = pkgname,
```

```
  category = "calculation",
  description = "Calculate Mn, Mw, Mz, dispersity"
)

## End(Not run)
```

`register_peak_algorithm` *Register a Peak Detection Algorithm*

Description

Registers a peak detection algorithm with the measure package. This function can be called from technique pack packages to add specialized algorithms.

Usage

```
register_peak_algorithm(
  name,
  algorithm_fn,
  pack_name,
  description = "",
  default_params = list(),
  param_info = list(),
  technique = NULL
)
```

Arguments

`name`	Algorithm name (e.g., `"cwt"`, `"finderskeepers"`). Must be unique.
`algorithm_fn`	The algorithm function. Must accept `location`, `value`, and return a `peaks_tbl` object. Additional parameters are passed via `...`.
`pack_name`	Source package name. Use `pkgname` from `.onLoad()`.
`description`	Brief description of the algorithm.
`default_params`	Named list of default parameter values.
`param_info`	Named list of parameter descriptions (for documentation).
`technique`	Optional technique name (e.g., `"SEC/GPC"`). If `NULL`, algorithm is considered general-purpose.

Value

Invisible `TRUE`.

See Also

peak_algorithms(), get_peak_algorithm()

Examples

```
## Not run:
# In a technique pack's R/zzz.R file:
.onLoad <- function(libname, pkgname) {
  if (requireNamespace("measure", quietly = TRUE)) {
    measure::register_peak_algorithm(
      name = "sec_loess_ist",
      algorithm_fn = .detect_peaks_sec_loess_ist,
      pack_name = pkgname,
      description = "LOESS smoothing with iterative soft thresholding",
      default_params = list(loess_span = 0.01, ist_points = 50),
      technique = "SEC/GPC"
    )
  }
}

## End(Not run)
```

---

register_peak_model *Register a Peak Model*

Description

Registers a peak model constructor with the measure package. Technique packs can use this to add custom peak shapes.

Usage

```
register_peak_model(
  name,
  constructor,
  pack_name,
  description = "",
  technique = NULL
)
```

Arguments

name	Model name (e.g., "gaussian", "emg", "fraser_suzuki").
constructor	Function that creates the peak model object.
pack_name	Source package name.
description	Brief description of the model.
technique	Optional technique name (e.g., "SEC/GPC").

Value

Invisible `TRUE`.

See Also

`peak_models()`, `create_peak_model()`

Examples

```
## Not run:
# In a technique pack's R/zzz.R:
register_peak_model(
  name = "fraser_suzuki",
  constructor = fraser_suzuki_model,
  pack_name = pkgname,
  description = "Fraser-Suzuki asymmetric peak",
  technique = "SEC/GPC"
)

## End(Not run)
```

---

render_validation_report

Render a Validation Report to Document Format

Description

Renders a `measure_validation_report` object to HTML, PDF, or Word format using standardized Quarto templates. Templates follow either ICH Q2(R2) or USP <1225> validation report structures.

Usage

```
render_validation_report(
  report,
  output_file = NULL,
  output_format = c("html", "pdf", "docx"),
  template = c("ich_q2", "usp_1225"),
  output_dir = ".",
  include_plots = TRUE,
  include_raw_data = FALSE,
  open = interactive(),
  quiet = FALSE,
  ...
)
```

Arguments

report	A measure_validation_report object created by measure_validation_report().
output_file	Output file path. If NULL, uses the report title with appropriate extension.
output_format	Output format: "html" (default), "pdf", or "docx". PDF requires a LaTeX installation (e.g., TinyTeX).
template	Template style: "ich_q2" (default) for ICH Q2(R2) layout, or "usp_1225" for USP <1225> compendial layout.
output_dir	Directory for output file. Default: current directory.
include_plots	Logical; include diagnostic plots? Default: TRUE.
include_raw_data	Logical; include raw data tables in appendix? Default: FALSE.
open	Logical; open the rendered document? Default: TRUE in interactive sessions.
quiet	Logical; suppress Quarto rendering messages? Default: FALSE.
...	Additional arguments passed to quarto::quarto_render().

Details

Template Styles:

ICH Q2(R2) Template (template = "ich_q2"):

- Organized by validation characteristic (specificity, linearity, etc.)
- Includes performance-based lifecycle considerations
- Structured for regulatory submission

USP <1225> Template (template = "usp_1225"):

- Compendial validation structure
- Category-based organization (I, II, III, IV)
- Emphasis on system suitability

Requirements:

- **HTML output**: Requires quarto package
- **PDF output**: Requires quarto package and LaTeX (TinyTeX recommended)
- **DOCX output**: Requires quarto package

Install Quarto from https://quarto.org/docs/get-started/. Install TinyTeX with quarto::quarto_install_tinytex().

Value

Invisibly returns the path to the rendered document.

See Also

measure_validation_report() to create the report object.

Examples

```
## Not run:
# Create a validation report (see measure_validation_report examples)
report <- measure_validation_report(
  title = "Method Validation Report",
  method_name = "HPLC Assay",
  analyst = "J. Smith"
)

# Render to HTML with ICH Q2 template
render_validation_report(report, output_format = "html")

# Render to PDF with USP template
render_validation_report(
  report,
  output_format = "pdf",
  template = "usp_1225",
  output_file = "validation_report.pdf"
)

# Render to Word for editing
render_validation_report(report, output_format = "docx")

## End(Not run)
```

---

sec_calibration *SEC/GPC Calibration Standards Summary*

Description

Summary information for the polystyrene calibration standards used with sec_chromatograms. Contains the known molecular weights and peak retention times needed to construct a calibration curve.

Format

A tibble with 5 observations and 3 variables:

standard Standard name (e.g., "PS_1k")

mw Known molecular weight in g/mol

peak_time Peak elution time in minutes

Details

The calibration curve for SEC/GPC relates log(MW) to retention time. For this simulated data: log10(MW) = 9.5 - 0.35 * time

Source

Simulated data generated for the measure package. See `data-raw/generate_datasets.R` for the generation script.

See Also

`sec_chromatograms` for the full chromatogram data

Examples

```
data(sec_calibration)

# View calibration data
sec_calibration

# Create calibration curve (if ggplot2 available)
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(sec_calibration, aes(x = peak_time, y = log10(mw))) +
    geom_point(size = 3) +
    geom_smooth(method = "lm", se = FALSE) +
    labs(x = "Peak Retention Time (min)", y = "log10(MW)",
         title = "SEC Calibration Curve")
}
```

---

sec_chromatograms *Simulated SEC/GPC Chromatography Data*

Description

Simulated Size Exclusion Chromatography (SEC) / Gel Permeation Chromatography (GPC) data for demonstration of molecular weight analysis. The dataset includes both narrow polystyrene calibration standards and polymer samples with broad molecular weight distributions.

Format

A tibble with 7,510 observations and 6 variables:

sample_id Sample identifier (standard or polymer name)

sample_type Either "standard" or "sample"

elution_time Elution/retention time in minutes

ri_signal Refractive index detector signal (arbitrary units)

known_mw Known weight-average molecular weight (g/mol)

known_dispersity Known dispersity (Mw/Mn); ~1.05 for standards

Details

SEC/GPC separates molecules by hydrodynamic size, with larger molecules eluting before smaller ones. This allows determination of molecular weight distributions and averages (Mn, Mw, Mz, dispersity).

The dataset is useful for demonstrating:

- Baseline correction for chromatography
- Calibration curve construction using standards
- Molecular weight calculations (step_measure_mw_averages)
- Molecular weight distribution analysis

The dataset contains:

Calibration Standards (narrow dispersity polystyrene):

- PS_1k: 1,000 g/mol
- PS_5k: 5,000 g/mol
- PS_20k: 20,000 g/mol
- PS_100k: 100,000 g/mol
- PS_500k: 500,000 g/mol

Polymer Samples (broad distribution):

- Polymer_A through Polymer_E with varying Mw and dispersity

The calibration relationship follows: log10(MW) = 9.5 - 0.35 * time

Source

Simulated data generated for the measure package. See `data-raw/generate_datasets.R` for the generation script.

See Also

`sec_calibration` for the calibration standards summary `hplc_chromatograms` for HPLC chromatography data `step_measure_mw_averages` for molecular weight calculations

Examples

```
data(sec_chromatograms)

# View structure
str(sec_chromatograms)

# Separate standards and samples
library(dplyr)
standards <- sec_chromatograms |> filter(sample_type == "standard")
samples <- sec_chromatograms |> filter(sample_type == "sample")

# Plot standards (if ggplot2 available)
```

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot(standards, aes(x = elution_time, y = ri_signal, color = sample_id)) +
    geom_line() +
    labs(x = "Elution Time (min)", y = "RI Signal",
         title = "SEC Calibration Standards",
         color = "Standard")
}
```

`set_measure_roles` *Set Measure Roles in a Recipe*

Description

Batch assign roles to columns based on their detected types or explicit patterns. This is a convenience wrapper around `recipes::update_role()` for common analytical data patterns.

Usage

```
set_measure_roles(
  recipe,
  id_cols = NULL,
  blank_cols = NULL,
  qc_cols = NULL,
  standard_cols = NULL,
  metadata_cols = NULL,
  measure_cols = NULL
)
```

Arguments

`recipe`	A recipe object.
`id_cols`	Column(s) to assign ”id” role. Accepts tidyselect.
`blank_cols`	Column(s) to assign ”blank” role. Accepts tidyselect.
`qc_cols`	Column(s) to assign ”qc” role. Accepts tidyselect.
`standard_cols`	Column(s) to assign ”standard” role. Accepts tidyselect.
`metadata_cols`	Column(s) to assign ”metadata” role. Accepts tidyselect.
`measure_cols`	Column(s) to assign ”measure” role. Accepts tidyselect.

Details

Common roles for analytical chemistry workflows:

Role	Purpose
id	Sample identifiers (not used in modeling)
blank	Blank/background samples for subtraction
qc	Quality control samples
standard	Calibration standards
metadata	Sample metadata (not used in modeling)
measure	Measurement columns for input steps
predictor	Columns used as model predictors
outcome	Target variable(s) for modeling

Value

Updated recipe object with roles assigned.

Examples

```
## Not run:
library(recipes)

# Basic role assignment
rec <- recipe(outcome ~ ., data = my_data) |>
  set_measure_roles(
    id_cols = sample_id,
    metadata_cols = c(batch, operator)
  )

# With QC and blank identification by column name patterns
rec <- recipe(outcome ~ ., data = my_data) |>
  set_measure_roles(
    id_cols = sample_id,
    blank_cols = starts_with("blank_"),
    qc_cols = starts_with("qc_")
  )

## End(Not run)
```

`smooth_window` *Parameters for smoothing steps*

Description

`smooth_window()` controls the window size for moving average and median smoothing. `smooth_sigma()` controls the standard deviation for Gaussian smoothing. `fourier_cutoff()` controls the frequency cutoff for Fourier filtering.

Usage

```
smooth_window(range = c(3L, 21L), trans = NULL)

smooth_sigma(range = c(0.5, 5), trans = NULL)

fourier_cutoff(range = c(0.01, 0.5), trans = NULL)

despike_threshold(range = c(2, 10), trans = NULL)
```

Arguments

range	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
trans	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
smooth_window()
smooth_sigma()
fourier_cutoff()
```

---

step_measure_absorbance

Convert Transmittance to Absorbance

Description

`step_measure_absorbance()` creates a *specification* of a recipe step that converts transmittance values to absorbance using the Beer-Lambert law.

Usage

```
step_measure_absorbance(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_absorbance")
)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
measures	An optional character vector of measure column names to process. If NULL (the default), all measure columns will be processed.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the step has been trained.
skip	A logical. Should the step be skipped when baking?
id	A character string that is unique to this step.

Details

This step applies the Beer-Lambert law transformation:

$$A = -\log_{10}(T)$$

where T is transmittance and A is absorbance.

Important: Transmittance values should be in the range (0, 1] or (0, 100]. Zero or negative values will produce -Inf or NaN with a warning.

The measurement locations are preserved unchanged.

Value

An updated version of recipe with the new step added.

See Also

step_measure_transmittance() for the inverse transformation

Other measure-preprocessing: step_measure_calibrate_x(), step_measure_calibrate_y(), step_measure_derivative(), step_measure_derivative_gap(), step_measure_emsc(), step_measure_kubelka_munk(), step_measure_log(), step_measure_map(), step_measure_msc(), step_measure_normalize_istd(), step_measure_osc(), step_measure_ratio_reference(), step_measure_snv(), step_measure_subtract_blank(), step_measure_subtract_reference(), step_measure_transmittance()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_absorbance() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_align_cow` *Correlation Optimized Warping Alignment*

Description

`step_measure_align_cow()` creates a *specification* of a recipe step that aligns spectra using Correlation Optimized Warping (COW). This method uses piecewise linear warping to correct for non-linear shifts.

Usage

```
step_measure_align_cow(
  recipe,
  measures = NULL,
  reference = c("mean", "median", "first"),
  segment_length = 30L,
  slack = 1L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_align_cow")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`reference`	How to determine the reference: `"mean"` (default, mean spectrum from training), `"median"` (median spectrum from training), or `"first"` (first sample).
`segment_length`	Length of each segment for warping. Default is 30. Tunable via `align_segment_length()`.
`slack`	Maximum compression/expansion per segment in points. Default is 1. A slack of 1 means each segment can shrink or expand by 1 point.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Correlation Optimized Warping (COW) divides signals into segments and uses dynamic programming to find the optimal piecewise linear warping that maximizes correlation with the reference spectrum.

Key parameters:

- `segment_length`: Controls the resolution of warping. Smaller segments allow more local corrections but increase computation.
- `slack`: Controls how much each segment can stretch or compress. Larger values allow more flexibility but may introduce artifacts.

This is a pure R implementation based on Nielsen et al. (1998).

Value

An updated recipe with the new step added.

References

Nielsen, N.P.V., Carstensen, J.M., and Smedsgaard, J. (1998). Aligning of single and multiple wavelength chromatographic profiles for chemometric data analysis using correlation optimised warping. Journal of Chromatography A, 805, 17-35.

See Also

Other measure-align: `step_measure_align_dtw()`, `step_measure_align_ptw()`, `step_measure_align_referenc`
`step_measure_align_shift()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_align_cow(segment_length = 20, slack = 2) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_align_dtw

Dynamic Time Warping Alignment

Description

`step_measure_align_dtw()` creates a *specification* of a recipe step that aligns spectra using Dynamic Time Warping (DTW). This method can handle non-linear distortions in the x-axis.

Usage

```
step_measure_align_dtw(
  recipe,
  measures = NULL,
  reference = c("mean", "median", "first"),
  window_type = c("none", "sakoechiba", "slantedband"),
  window_size = 10L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_align_dtw")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`reference`	How to determine the reference: • `"mean"` (default): Use the mean spectrum from training • `"median"`: Use the median spectrum from training • `"first"`: Use the first sample
`window_type`	Windowing constraint for DTW. One of `"none"` (default), `"sakoechiba"`, or `"slantedband"`.
`window_size`	Window size for constrained DTW. Default is 10.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

DTW finds the optimal non-linear alignment between two sequences by minimizing a distance measure while allowing warping of the time/x-axis.

This is useful for:

- Chromatographic peak alignment
- Correcting non-linear retention time shifts
- Aligning spectra with complex distortions

Requires the `dtw` package to be installed.

Value

An updated recipe with the new step added.

See Also

Other measure-align: step_measure_align_cow(), step_measure_align_ptw(), step_measure_align_referenc
step_measure_align_shift()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_align_dtw() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_align_ptw

Parametric Time Warping Alignment

Description

step_measure_align_ptw() creates a *specification* of a recipe step that aligns spectra using Parametric Time Warping (PTW). This method uses polynomial warping functions to correct for shifts and distortions.

Usage

```
step_measure_align_ptw(
  recipe,
  measures = NULL,
  reference = c("mean", "median", "first"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_align_ptw")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
reference	How to determine the reference: "mean" (default, mean spectrum from training), "median" (median spectrum from training), or "first" (first sample).

`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Parametric Time Warping optimizes polynomial warping coefficients to maximize the correlation between each sample and the reference spectrum. This corrects for smooth, continuous distortions in the x-axis.

Requires the `ptw` package to be installed.

Value

An updated recipe with the new step added.

References

Eilers, P.H.C. (2004). Parametric Time Warping. Analytical Chemistry, 76(2), 404-411.

See Also

Other measure-align: `step_measure_align_cow()`, `step_measure_align_dtw()`, `step_measure_align_referenc`
`step_measure_align_shift()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_align_ptw() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_align_reference

Align to Reference Spectrum

Description

`step_measure_align_reference()` creates a *specification* of a recipe step that aligns spectra to a user-provided reference spectrum using cross-correlation.

Usage

```
step_measure_align_reference(
  recipe,
  measures = NULL,
  ref_spectrum,
  max_shift = 10L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_align_reference")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
ref_spectrum	A numeric vector containing the reference spectrum. Must have the same length as the measurement spectra.
max_shift	Maximum shift (in points) to consider. Default is 10.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

Similar to step_measure_align_shift(), but uses an externally provided reference spectrum instead of computing one from training data. This is useful when you have a known standard or calibration spectrum.

Value

An updated recipe with the new step added.

See Also

Other measure-align: step_measure_align_cow(), step_measure_align_dtw(), step_measure_align_ptw(), step_measure_align_shift()

Examples

```
library(recipes)

# Create a reference spectrum (in practice, this would be from calibration)
ref <- rep(1, 100)  # placeholder

# Note: This example would need matching spectrum lengths to work
## Not run:
```

```
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_align_reference(ref_spectrum = ref) |>
  prep()

## End(Not run)
```

`step_measure_align_shift` *Shift Alignment via Cross-Correlation*

Description

`step_measure_align_shift()` creates a *specification* of a recipe step that aligns spectra by finding the optimal shift using cross-correlation.

Usage

```
step_measure_align_shift(
  recipe,
  measures = NULL,
  max_shift = 10L,
  reference = c("mean", "median", "first"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_align_shift")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`max_shift`	Maximum shift (in points) to consider. Default is 10. Tunable via `align_max_shift()`.
`reference`	How to determine the reference: • `"mean"` (default): Use the mean spectrum from training • `"median"`: Use the median spectrum from training • `"first"`: Use the first sample
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step corrects for small linear shifts between spectra, which can occur due to:

- Wavelength calibration drift
- Sample positioning differences
- Temperature effects on instrument

The optimal shift is found by maximizing the cross-correlation between each spectrum and the reference. After shifting, edge values are filled by constant extrapolation.

Value

An updated recipe with the new step added.

See Also

Other measure-align: `step_measure_align_cow()`, `step_measure_align_dtw()`, `step_measure_align_ptw()`, `step_measure_align_reference()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_align_shift(max_shift = 5) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_augment_noise

Add Random Noise to Measurements

Description

`step_measure_augment_noise()` creates a *specification* of a recipe step that adds controlled random noise to spectral data for data augmentation.

Usage

```
step_measure_augment_noise(
  recipe,
  sd = 0.01,
  distribution = c("gaussian", "uniform"),
  relative = TRUE,
  measures = NULL,
```

```
  role = NA,
  trained = FALSE,
  skip = TRUE,
  id = recipes::rand_id("measure_augment_noise")
)
```

Arguments

`recipe`	A recipe object.
`sd`	Standard deviation of noise. If `relative = TRUE` (default), this is relative to the signal range (0.01 = 1% of range). If `relative = FALSE`, this is the absolute noise level.
`distribution`	Noise distribution: `"gaussian"` (default) or `"uniform"`.
`relative`	Logical. If `TRUE` (default), `sd` is relative to signal range.
`measures`	An optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking? Default is `TRUE`, meaning augmentation only applies during training.
`id`	Unique step identifier.

Details

Data augmentation adds variability to training data to improve model robustness. Adding noise simulates measurement uncertainty and helps models generalize better.

Default behavior (`skip = TRUE`): The augmentation is only applied during `prep()` on training data. When `bake()` is called on new data, the step is skipped.

Reproducibility: The noise is deterministic based on the row content, so the same input always produces the same augmented output within a session.

Value

An updated recipe with the new step added.

See Also

Other measure-augmentation: `step_measure_augment_scale()`, `step_measure_augment_shift()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_augment_noise(sd = 0.02) |>
  prep()
```

```
# Noise only applied to training data
bake(rec, new_data = NULL)
```

`step_measure_augment_scale` *Random Intensity Scaling*

Description

`step_measure_augment_scale()` creates a *specification* of a recipe step that applies random intensity scaling for scale invariance training.

Usage

```
step_measure_augment_scale(
  recipe,
  range = c(0.9, 1.1),
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = TRUE,
  id = recipes::rand_id("measure_augment_scale")
)
```

Arguments

`recipe`	A recipe object.
`range`	A numeric vector of length 2 specifying the range of scaling factors. Default is `c(0.9, 1.1)`, meaning 90%-110% of original.
`measures`	An optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking? Default is `TRUE`.
`id`	Unique step identifier.

Details

This step multiplies spectrum values by a random scaling factor sampled uniformly from the specified range. This helps models become robust to variations in signal intensity.

Common use cases:

- Simulating concentration variations
- Compensating for detector sensitivity differences
- Making models robust to sample preparation variability

Default behavior (`skip = TRUE`): The scaling is only applied during training. When predicting on new data, the step is skipped.

Value

An updated recipe with the new step added.

See Also

Other measure-augmentation: `step_measure_augment_noise()`, `step_measure_augment_shift()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_augment_scale(range = c(0.8, 1.2)) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_augment_shift` *Add Random X-axis Shifts*

Description

`step_measure_augment_shift()` creates a *specification* of a recipe step that applies random shifts along the x-axis for shift invariance training.

Usage

```
step_measure_augment_shift(
  recipe,
  max_shift = 1,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = TRUE,
  id = recipes::rand_id("measure_augment_shift")
)
```

Arguments

`recipe`	A recipe object.
`max_shift`	Maximum shift amount in location units. The actual shift is uniformly sampled from `[-max_shift, max_shift]`.
`measures`	An optional character vector of measure column names.
`role`	Not used.

`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking? Default is `TRUE`.
`id`	Unique step identifier.

Details

This step adds random x-axis shifts to help models become invariant to small retention time or wavelength shifts. This is particularly useful for chromatographic data where peak positions may vary slightly.

The spectrum is interpolated to the shifted positions using linear interpolation. Values outside the original range use boundary values.

Default behavior (`skip = TRUE`): The shift is only applied during training. When predicting on new data, the step is skipped.

Value

An updated recipe with the new step added.

See Also

Other measure-augmentation: `step_measure_augment_noise()`, `step_measure_augment_scale()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_augment_shift(max_shift = 2) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_baseline_airpls` *Adaptive Iteratively Reweighted Penalized Least Squares Baseline*

Description

`step_measure_baseline_airpls()` creates a *specification* of a recipe step that applies airPLS baseline correction. This method automatically adjusts weights based on the difference between the signal and fitted baseline.

Usage

```
step_measure_baseline_airpls(
  recipe,
  measures = NULL,
  lambda = 1e+05,
  max_iter = 50L,
  tol = 0.001,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_airpls")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`lambda`	Smoothness parameter. Higher values produce smoother baselines. Default is 1e5. Tunable via `baseline_lambda()`.
`max_iter`	Maximum number of iterations. Default is 50.
`tol`	Convergence tolerance for weight changes. Default is 1e-3.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

airPLS (Adaptive Iteratively Reweighted Penalized Least Squares) is an improvement over standard ALS that automatically adapts the asymmetry parameter based on the residuals. Key features:

- No need to manually set asymmetry parameter
- Good for signals with varying baseline curvature
- Robust to different peak heights

Value

An updated recipe with the new step added.

References

Zhang, Z.M., Chen, S., & Liang, Y.Z. (2010). Baseline correction using adaptive iteratively reweighted penalized least squares. Analyst, 135, 1138-1146.

See Also

Other measure-baseline: step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morph(), step_measure_baseline_morphologica step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_airpls(lambda = 1e5) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_als

Asymmetric Least Squares (ALS) Baseline Correction

Description

step_measure_baseline_als() creates a *specification* of a recipe step that applies Asymmetric Least Squares baseline correction to measurement data. ALS iteratively fits a smooth baseline giving less weight to points above the baseline (peaks).

Usage

```
step_measure_baseline_als(
  recipe,
  measures = NULL,
  lambda = 1e+06,
  p = 0.01,
  max_iter = 20L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_als")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`lambda`	Smoothness parameter (2nd derivative constraint). Higher values produce smoother baselines. Default is `1e6`. Typical range is 1e3 to 1e9. Tunable via `baseline_lambda()`.
`p`	Asymmetry parameter controlling weight for positive residuals. Values near 0 (e.g., 0.001-0.05) work well for spectra with peaks above baseline. Default is `0.01`. Tunable via `baseline_asymmetry()`.
`max_iter`	Maximum number of iterations. Default is `20`.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

Asymmetric Least Squares (ALS) baseline correction uses a Whittaker smoother with asymmetric weights to fit a baseline that follows the lower envelope of the spectrum. The algorithm iteratively:

1

. Fits a smooth baseline using penalized least squares 2. Calculates residuals (spectrum - baseline) 3. Assigns weights: `p` for positive residuals (peaks), `1-p` for negative 4. Repeats until convergence or max iterations

The smoothness is controlled by `lambda`, which penalizes the second derivative of the baseline. Larger `lambda` produces smoother baselines.

ALS is particularly effective for:

- NIR/IR spectroscopy with broad baseline drift
- Raman spectroscopy with fluorescence background
- UV-Vis spectroscopy with scattering effects

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `lambda`, `p`, and `id` is returned.

Tuning

This step has parameters that can be tuned:

- `lambda`: Use `baseline_lambda()` (log10 scale recommended)
- `p`: Use `baseline_asymmetry()`

References

Eilers, P.H.C. and Boelens, H.F.M. (2005). Baseline Correction with Asymmetric Least Squares Smoothing. Leiden University Medical Centre report.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_als(lambda = 1e6, p = 0.01) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_arpls

Asymmetrically Reweighted Penalized Least Squares Baseline Correction

---

Description

`step_measure_baseline_arpls()` creates a *specification* of a recipe step that applies arPLS baseline correction using asymmetric weighting.

Usage

```
step_measure_baseline_arpls(
  recipe,
  measures = NULL,
  lambda = 1e+05,
  ratio = 0.001,
  max_iter = 50L,
  tol = 0.001,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_arpls")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`lambda`	Smoothing parameter. Larger values produce smoother baselines. Default is 1e5.
`ratio`	Asymmetric weighting ratio. Default is 0.001.
`max_iter`	Maximum number of iterations. Default is 50.
`tol`	Convergence tolerance. Default is 1e-3.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

The arPLS algorithm uses asymmetric least squares with a ratio-based weighting scheme. It is robust to peak interference and works well for signals with varying baseline curvature.

Reference: Baek et al. (2015), Analyst 140, 250-257

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_arpls(lambda = 1e5) |>
  prep()
```

---

step_measure_baseline_aspls

Adaptive Smoothness Penalized Least Squares Baseline

Description

`step_measure_baseline_aspls()` creates a *specification* of a recipe step that applies Adaptive Smoothness Penalized Least Squares baseline correction.

Usage

```
step_measure_baseline_aspls(
  recipe,
  measures = NULL,
  lambda = 1e+06,
  alpha = 0.5,
  max_iter = 50L,
  tol = 1e-05,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_aspls")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`lambda`	Base smoothness parameter. Default is 1e6.
`alpha`	Adaptive weight parameter controlling smoothness adaptation (0 = no adaptation, 1 = maximum adaptation). Higher values cause regions with larger residuals to receive higher smoothness penalties. Note that adaptation is applied globally via an averaged lambda. Default is 0.5. Tunable via `baseline_alpha()`.
`max_iter`	Maximum number of iterations. Default is 50.

`tol`	Convergence tolerance. Default is 1e-5.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

aspls adapts the smoothness parameter based on the signal properties. The algorithm computes a local smoothness weight based on residual magnitude, then uses the global average as the effective lambda. This provides some adaptation to peak intensity while maintaining computational efficiency.

This method is particularly effective for:

- Signals with varying peak widths
- Data with both sharp peaks and gradual baseline changes
- Chromatography with complex baselines

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_aspls(lambda = 1e6, alpha = 0.5) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_baseline_auto` *Automatic Baseline Correction Method Selection*

Description

`step_measure_baseline_auto()` creates a *specification* of a recipe step that automatically selects and applies the best baseline correction method based on signal characteristics.

Usage

```
step_measure_baseline_auto(
  recipe,
  measures = NULL,
  methods = c("rolling", "airpls", "snip", "tophat", "minima"),
  role = NA,
  trained = FALSE,
  selected_method = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_auto")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`methods`	Character vector of methods to consider. Default includes all available methods.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`selected_method`	The method selected during training (internal).
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step analyzes the signal characteristics (noise level, baseline curvature, peak density) during training and selects an appropriate baseline correction method. The selected method is then applied consistently during baking.

Method selection heuristics:

- High noise, smooth baseline: rolling ball
- Complex curvature: airPLS or arPLS
- Sharp peaks: SNIP or top-hat
- Simple baseline: polynomial or minima

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morph(), step_measure_baseline_morphologica step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_auto() |>
  prep()
```

---

step_measure_baseline_custom

Custom Baseline Correction with User-Provided Function

Description

`step_measure_baseline_custom()` creates a *specification* of a recipe step that applies a user-provided function for baseline correction. This allows for flexible, custom baseline estimation algorithms.

Usage

```
step_measure_baseline_custom(
  recipe,
  .fn,
  ...,
  subtract = TRUE,
  measures = NULL,
  tunable = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_custom")
)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
.fn	A function or formula for baseline estimation. The function should accept a measure_tbl (tibble with location and value columns) and return a numeric vector of baseline values with the same length as the input. Formulas are converted to functions via rlang::as_function(), where .x represents the measure_tbl.
...	Additional arguments passed to .fn. These are captured as quosures and evaluated at bake time.
subtract	If TRUE (default), the baseline is subtracted from the signal. If FALSE, the baseline values replace the original values (useful for extracting baselines).
measures	An optional character vector of measure column names to process. If NULL (the default), all measure columns (columns with class measure_list) will be processed.
tunable	An optional named list specifying which arguments in ... are tunable. Each element should be a list with pkg, fun, and optionally range. See Details.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
skip	A logical. Should the step be skipped when the recipe is baked?
id	A character string that is unique to this step to identify it.

Details

This step allows you to use any baseline estimation algorithm by providing a custom function. The function receives a measure_tbl object (a tibble with location and value columns) and should return a numeric vector of the estimated baseline values.

Function Contract:

Your function should:

- Accept a measure_tbl as its first argument
- Return a numeric vector of the same length as nrow(measure_tbl)
- Handle NA values appropriately

Formula Interface:

You can use a formula instead of a function. The formula is converted to a function where .x represents the measure_tbl:

```
# These are equivalent:
step_measure_baseline_custom(.fn = function(x) mean(x$value))
step_measure_baseline_custom(.fn = ~ mean(.x$value))
```

Tunability:

To make parameters tunable with dials, provide a tunable argument:

```
step_measure_baseline_custom(
  .fn = ~ stats::loess(.x$value ~ .x$location, span = span)$fitted,
  span = 0.5,
  tunable = list(
    span = list(pkg = "dials", fun = "degree", range = c(0.1, 0.9))
  )
)
```

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `subtract`, and `id` is returned.

See Also

`step_measure_baseline_als()`, `step_measure_baseline_poly()` for built-in baseline correction methods.

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)

# Simple polynomial baseline using a function
poly_baseline <- function(x) {
  fit <- lm(x$value ~ poly(x$location, 2))
  predict(fit)
}

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_custom(.fn = poly_baseline) |>
  prep()

bake(rec, new_data = NULL)

# Using formula interface with additional parameters
```

```
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_custom(
    .fn = ~ stats::loess(.x$value ~ .x$location, span = span)$fitted,
    span = 0.5
  ) |>
  prep()
```

`step_measure_baseline_fastchrom` *Fast Chromatography Baseline Correction*

Description

`step_measure_baseline_fastchrom()` creates a *specification* of a recipe step that applies fast baseline correction optimized for chromatography data.

Usage

```
step_measure_baseline_fastchrom(
  recipe,
  measures = NULL,
  lambda = 1e+06,
  window = 50L,
  max_iter = 10L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_fastchrom")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`lambda`	Smoothness parameter. Default is 1e6.
`window`	Window size for local minima detection. Default is 50.
`max_iter`	Maximum number of refinement iterations. Default is 10.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This algorithm combines morphological operations with penalized least squares for fast and robust baseline estimation:

1. Finds local minima using a rolling window
2. Smooths the minima to get initial baseline estimate
3. Iteratively refines using weighted PLS

Particularly effective for SEC/GPC chromatography and other analytical techniques with well-defined peaks.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_fastchrom(lambda = 1e6, window = 50) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_gpc

GPC/SEC Baseline Correction

Description

[Superseded]

`step_measure_baseline_gpc()` creates a *specification* of a recipe step that applies baseline correction optimized for Gel Permeation Chromatography (GPC) or Size Exclusion Chromatography (SEC) data. This method estimates the baseline by interpolating between baseline regions at the start and end of the chromatogram.

This step has been superseded by `measure.sec::step_sec_baseline()`. For new code, we recommend using the `measure.sec` package which provides more complete SEC/GPC analysis functionality.

Usage

```
step_measure_baseline_gpc(
  recipe,
  measures = NULL,
  left_frac = 0.05,
  right_frac = 0.05,
  method = "linear",
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_gpc")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`left_frac`	Fraction of points from the beginning to use as the left baseline region. Default is `0.05` (first 5% of data points).
`right_frac`	Fraction of points from the end to use as the right baseline region. Default is `0.05` (last 5% of data points).
`method`	Method for baseline estimation. One of: • `"linear"` (default): Linear interpolation between left and right means • `"median"`: Uses median of baseline regions (more robust to outliers) • `"spline"`: Smooth spline through baseline regions
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

GPC/SEC chromatograms typically have distinct baseline regions at the beginning and end where no polymer elutes. This step leverages this characteristic by:

1 2. Computing a representative baseline value for each region (mean or median) 3. Interpolating between these values to estimate the full baseline 4. Subtracting the estimated baseline from the signal

The `left_frac` and `right_frac` parameters control how much of the chromatogram is considered "baseline". Choose values that:

- Include only the flat, signal-free regions
- Exclude any polymer peaks or system peaks
- Are large enough to average out noise

Unlike general-purpose baseline methods like ALS or polynomial fitting, this approach is specifically designed for the characteristic shape of GPC/SEC chromatograms and is computationally very fast.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `left_frac`, `right_frac`, `method`, and `id` is returned.

See Also

`step_measure_baseline_als()` for general-purpose baseline correction, `step_measure_detrend()` for simple trend removal.

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`, `step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)

# Using meats_long as example (works on any measurement data)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
```

```
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_gpc(left_frac = 0.1, right_frac = 0.1) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_baseline_iarpls` *Improved arPLS Baseline Correction (Two-Stage)*

Description

`step_measure_baseline_iarpls()` creates a *specification* of a recipe step that applies Improved arPLS baseline correction using a two-stage approach.

Usage

```
step_measure_baseline_iarpls(
  recipe,
  measures = NULL,
  lambda = 1e+06,
  lambda_1 = 10000,
  max_iter = 10L,
  tol = 1e-05,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_iarpls")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`lambda`	Final smoothness parameter. Default is 1e6.
`lambda_1`	First stage (coarse) smoothness parameter. Default is 1e4.
`max_iter`	Maximum number of iterations. Default is 10.
`tol`	Convergence tolerance. Default is 1e-5.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

iarpls uses a two-stage approach:

1. First stage with smaller lambda for coarse baseline estimation
2. Second stage with larger lambda for refined baseline using weights derived from the first stage

This approach often provides better results than single-stage arPLS for signals with complex baseline patterns.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphologica`
`step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_iarpls(lambda = 1e6, lambda_1 = 1e4) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_minima

Local Minima Interpolation Baseline Correction

Description

`step_measure_baseline_minima()` creates a *specification* of a recipe step that estimates baseline by interpolating between local minima.

Usage

```
step_measure_baseline_minima(
  recipe,
  measures = NULL,
  window_size = 50L,
  method = c("spline", "linear"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_minima")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
window_size	Window size for finding local minima. Default is 50.
method	Interpolation method: "linear" or "spline". Default is "spline".
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

This method finds local minima within specified windows, then interpolates between them to create a baseline estimate. This is intuitive and works well when baseline points are clearly identifiable as local minima.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_morph(), step_measure_baseline_morphologica step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_minima(window_size = 30, method = "spline") |>
  prep()
```

---

step_measure_baseline_morph

Iterative Morphological Baseline Correction

Description

`step_measure_baseline_morph()` creates a *specification* of a recipe step that applies iterative morphological baseline correction using erosion and dilation operations.

Usage

```
step_measure_baseline_morph(
  recipe,
  measures = NULL,
  half_window = 50L,
  iterations = 10L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_morph")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`half_window`	Half-window size for the structuring element. Default is 50.
`iterations`	Number of erosion-dilation iterations. Default is 10.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This method applies iterative morphological operations (erosion followed by dilation) to estimate the baseline. Multiple iterations can help refine the baseline estimate for complex signals.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morphologic step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_morph(half_window = 30, iterations = 5) |>
  prep()
```

---

step_measure_baseline_morphological

Morphological Baseline Correction (Erosion/Dilation)

Description

`step_measure_baseline_morphological()` creates a *specification* of a recipe step that applies morphological erosion followed by dilation for baseline estimation.

Usage

```
step_measure_baseline_morphological(
  recipe,
  measures = NULL,
  window_size = 50L,
  iterations = 1L,
  role = NA,
```

```
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_morphological")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
window_size	Size of the structuring element. Default is 50.
iterations	Number of erosion iterations. Default is 1.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

This morphological approach uses erosion (local minimum) to push the baseline down below peaks, followed by dilation (local maximum) to smooth the result.

Multiple erosion iterations can be used for signals with tall peaks that require more aggressive baseline estimation.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morph(), step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_morphological(window_size = 50) |>
  prep()
```

```
bake(rec, new_data = NULL)
```

`step_measure_baseline_poly` *Polynomial Baseline Correction*

Description

`step_measure_baseline_poly()` creates a *specification* of a recipe step that applies polynomial baseline correction to measurement data. The method fits a polynomial to the spectrum, optionally with iterative peak exclusion.

Usage

```
step_measure_baseline_poly(
  recipe,
  measures = NULL,
  degree = 2L,
  max_iter = 0L,
  threshold = 1.5,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_poly")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`degree`	Polynomial degree for baseline fitting. Default is `2` (quadratic). Higher degrees fit more complex baselines but risk overfitting. Tunable via `baseline_degree()`.
`max_iter`	Maximum number of iterations for peak exclusion. Default is `0` (no iteration, fit polynomial to all points). Set to a positive integer to iteratively exclude points above the fitted baseline.
`threshold`	Number of standard deviations above baseline for a point to be excluded in iterative fitting. Default is `1.5`. Only used when `max_iter > 0`.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

Polynomial baseline correction fits a polynomial function to the spectrum and subtracts it. This is effective for removing smooth, curved baselines caused by instrumental drift, scattering, or other slowly varying effects.

When `max_iter > 0`, the algorithm uses iterative peak exclusion:

1. Fit polynomial to all points
2. Calculate residuals (spectrum - baseline)
3. Exclude points where residual > threshold * SD(residuals)
4. Refit polynomial to remaining points
5. Repeat until convergence or max_iter reached

This iterative approach prevents peaks from pulling up the baseline estimate.

Degree selection:

- `degree = 1`: Linear baseline (for simple drift)
- `degree = 2`: Quadratic (most common, handles gentle curvature)
- `degree = 3-5`: Higher-order (for complex baselines, use cautiously)

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `degree`, and `id` is returned.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphological()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)

# Simple polynomial baseline (no iteration)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
```

```
  step_measure_baseline_poly(degree = 2) |>
  prep()

bake(rec, new_data = NULL)

# With iterative peak exclusion
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_poly(degree = 3, max_iter = 5, threshold = 2) |>
  prep()
```

`step_measure_baseline_py` *Python-Based Baseline Correction via pybaselines*

Description

`step_measure_baseline_py()` creates a *specification* of a recipe step that applies baseline correction using the Python pybaselines library, which provides 50+ baseline correction algorithms.

Usage

```
step_measure_baseline_py(
  recipe,
  method = "asls",
  ...,
  subtract = TRUE,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_py")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`method`	The pybaselines method to use. Common methods include: • Whittaker methods: `"asls"`, `"iasls"`, `"airpls"`, `"arpls"`, `"drpls"`, `"psalsa"` • Polynomial methods: `"poly"`, `"modpoly"`, `"imodpoly"`, `"loess"`, `"quant_reg"` • Morphological: `"mor"`, `"imor"`, `"rolling_ball"`, `"tophat"` • Spline: `"pspline_asls"`, `"pspline_airpls"`, `"mixture_model"`

- Smooth: `"snip"`, `"swima"`, `"noise_median"`
- Classification: `"dietrich"`, `"golotvin"`, `"fastchrom"`
- See pybaselines documentation for the full list.

`...` Additional arguments passed to the pybaselines method. Common parameters include:

- `lam`: Smoothness parameter for Whittaker methods (default varies by method)
- `p`: Asymmetry parameter for ALS methods (default ~0.01)
- `poly_order`: Polynomial degree for polynomial methods
- `half_window`: Window size for morphological methods
- `max_half_window`: Maximum window for SNIP method

`subtract` If `TRUE` (default), the baseline is subtracted from the signal. If `FALSE`, the baseline values replace the original values (useful for extracting baselines).

`measures` An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.

`role` Not used by this step since no new variables are created.

`trained` A logical to indicate if the quantities for preprocessing have been estimated.

`skip` A logical. Should the step be skipped when the recipe is baked?

`id` A character string that is unique to this step to identify it.

Details

This step provides access to the comprehensive pybaselines Python library, which implements over 50 baseline correction algorithms across several categories:

Whittaker Methods:

Based on penalized least squares with asymmetric weights:

- `asls`: Asymmetric Least Squares (good general-purpose method)
- `iasls`: Improved ALS with automatic smoothness selection
- `airpls`: Adaptive iteratively reweighted penalized least squares
- `arpls`: Asymmetrically reweighted penalized least squares
- `psalsa`: Peaked Signal's Asymmetric Least Squares Algorithm

Polynomial Methods:

Fit polynomials to baseline regions:

- `poly`: Simple polynomial fitting
- `modpoly`: Modified polynomial (iterative)
- `imodpoly`: Improved modified polynomial
- `loess`: Local regression (LOESS)

Morphological Methods:

Based on mathematical morphology:

- `mor`: Morphological opening
- `imor`: Improved morphological
- `rolling_ball`: Rolling ball algorithm
- `tophat`: Top-hat transform

Requirements:
This step requires the `reticulate` package and Python with pybaselines installed. Install pybaselines with:

```
reticulate::py_require("pybaselines")
```

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `method`, `subtract`, and `id` is returned.

See Also

`step_measure_baseline_als()`, `step_measure_baseline_custom()` for R-based alternatives.

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphological()`, `step_measure_baseline_poly()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


# Asymmetric Least Squares baseline correction
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_py(method = "asls", lam = 1e6, p = 0.01) |>
  prep()

bake(rec, new_data = NULL)
```

```
# Using SNIP algorithm
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_py(method = "snip", max_half_window = 40) |>
  prep()
```

`step_measure_baseline_rf`
Robust Fitting Baseline Correction

Description

`step_measure_baseline_rf()` creates a *specification* of a recipe step that applies robust fitting baseline correction to measurement data. This method uses local regression with iterative reweighting to fit a baseline that is resistant to peaks.

Usage

```
step_measure_baseline_rf(
  recipe,
  measures = NULL,
  span = 2/3,
  maxit = c(5L, 5L),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_rf")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`span`	Controls the amount of smoothing. This is the fraction of data used in computing each fitted value. Default is `2/3`. Smaller values produce less smooth baselines that follow local features more closely.
`maxit`	A length-2 integer vector specifying the number of iterations for the robust fit. The first value is for the asymmetric weighting function, the second for symmetric weighting. Default is `c(5, 5)`.
`role`	Not used by this step since no new variables are created.

`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

Robust fitting baseline correction uses local polynomial regression (LOESS/LOWESS) with iterative reweighting to estimate the baseline. The algorithm uses asymmetric weights in initial iterations to down-weight peaks, then symmetric weights for final smoothing.

This method is particularly effective for:

- Spectra with peaks of varying widths
- Data where the baseline shape is not well-described by a polynomial
- Situations where peaks should not influence the baseline estimate

The `span` parameter controls the trade-off between smoothness and local adaptation:

- Larger span (e.g., 0.8): Smoother baseline, may miss local variations
- Smaller span (e.g., 0.3): More local adaptation, may overfit

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `span`, and `id` is returned.

See Also

`subtract_rf_baseline()` for the standalone function this step wraps.

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphological()`, `step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_rf(span = 0.5) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_rolling

Rolling Ball Baseline Correction

Description

`step_measure_baseline_rolling()` creates a *specification* of a recipe step that applies rolling ball baseline correction. This morphological approach "rolls" a ball of specified radius along the underside of the spectrum.

Usage

```
step_measure_baseline_rolling(
  recipe,
  measures = NULL,
  window_size = 100,
  smoothing = 50,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_rolling")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`window_size`	The diameter of the rolling ball in number of points. Default is 100.
`smoothing`	Additional smoothing window applied to the baseline. Default is 50.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

The rolling ball algorithm simulates rolling a ball of specified radius along the underside of the spectrum. Points where the ball touches become the baseline. This is effective for:

- Chromatographic baselines
- Spectra with gradual drift
- Data where peaks are narrower than baseline features

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphological()`, `step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`, `step_measure_detrend()`

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_rolling(window_size = 50) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_snip

SNIP Baseline Correction

Description

`step_measure_baseline_snip()` creates a *specification* of a recipe step that applies SNIP (Statistics-sensitive Non-linear Iterative Peak-clipping) baseline correction.

Usage

```
step_measure_baseline_snip(
  recipe,
  measures = NULL,
  iterations = 40L,
  decreasing = TRUE,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_snip")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`iterations`	Number of clipping iterations. More iterations produce lower baselines. Default is 40.
`decreasing`	Logical. If `TRUE` (default), iterations decrease from `iterations` to 1. If `FALSE`, uses fixed window size.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

SNIP is a robust baseline estimation algorithm originally developed for gamma-ray spectroscopy. It works by iteratively replacing each point with the minimum of itself and the average of its neighbors at increasing distances.

The algorithm is particularly effective for:

- Spectra with sharp peaks on slowly varying baseline
- X-ray fluorescence and diffraction
- Mass spectrometry

Value

An updated recipe with the new step added.

References

Ryan, C.G., et al. (1988). SNIP, a statistics-sensitive background treatment for the quantitative analysis of PIXE spectra in geoscience applications. Nuclear Instruments and Methods in Physics Research B, 34, 396-402.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morph(), step_measure_baseline_morphological(), step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_tophat(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_snip(iterations = 30) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_baseline_tophat

Top-Hat Morphological Baseline Correction

Description

step_measure_baseline_tophat() creates a *specification* of a recipe step that applies top-hat morphological baseline correction.

Usage

```
step_measure_baseline_tophat(
  recipe,
  measures = NULL,
  half_window = 50L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_baseline_tophat")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
half_window	Half-window size for the structuring element. Default is 50.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

The top-hat transform is a morphological operation that extracts bright features (peaks) from a dark background. It is computed as the difference between the original signal and its morphological opening.

This is effective for chromatography with sharp, well-defined peaks on a smooth baseline.

Value

An updated recipe with the new step added.

See Also

Other measure-baseline: step_measure_baseline_airpls(), step_measure_baseline_als(), step_measure_baseline_arpls(), step_measure_baseline_aspls(), step_measure_baseline_auto(), step_measure_baseline_custom(), step_measure_baseline_fastchrom(), step_measure_baseline_gpc(), step_measure_baseline_iarpls(), step_measure_baseline_minima(), step_measure_baseline_morph(), step_measure_baseline_morphological(), step_measure_baseline_poly(), step_measure_baseline_py(), step_measure_baseline_rf(), step_measure_baseline_rolling(), step_measure_baseline_snip(), step_measure_detrend()

Examples

```
library(recipes)


rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_baseline_tophat(half_window = 30) |>
  prep()
```

`step_measure_batch_reference` *Reference-Based Batch Correction*

Description

`step_measure_batch_reference()` creates a *specification* of a recipe step that corrects for batch effects using reference samples. This is a simpler alternative to ComBat-style correction that doesn't require heavy dependencies.

Usage

```
step_measure_batch_reference(
  recipe,
  ...,
  batch_col = "batch_id",
  sample_type_col = "sample_type",
  reference_type = "reference",
  method = c("median_ratio", "mean_ratio", "median_center", "mean_center"),
  target_batch = NULL,
  min_ref = 2,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_batch_reference")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns.
`batch_col`	Name of the column containing batch identifiers.
`sample_type_col`	Name of the column containing sample type.
`reference_type`	Value(s) in `sample_type_col` that identify reference samples to use for batch correction. Default is `"reference"`.
`method`	Correction method: • `"median_ratio"` (default): Scale by ratio of reference medians • `"mean_ratio"`: Scale by ratio of reference means • `"median_center"`: Center batches to common median • `"mean_center"`: Center batches to common mean
`target_batch`	Which batch to use as reference. Default is the first batch (alphabetically). Can also be `"global"` to use global reference median/mean.
`min_ref`	Minimum number of reference samples per batch. Default is 2.

`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Correction Methods:

Median/Mean Ratio: Multiplies all samples in a batch by: `target_reference / batch_reference`

This preserves relative differences within batches while aligning batch centers.

Median/Mean Center: Subtracts the difference between batch reference and target reference. This is appropriate for log-transformed data.

Reference Samples:

Reference samples should be identical samples run in each batch (e.g., pooled QC, reference material). The step will error if any batch lacks sufficient reference samples.

Value

An updated recipe with the new step added.

See Also

`step_measure_drift_qc_loess()` for within-batch drift correction.

Examples

```
library(recipes)

# Data with batch effects
data <- data.frame(
  sample_id = paste0("S", 1:20),
  sample_type = rep(c("reference", "unknown", "unknown", "unknown", "reference"), 4),
  batch_id = rep(c("B1", "B1", "B2", "B2"), 5),
  feature1 = c(rep(100, 10), rep(120, 10)) + rnorm(20, sd = 5),  # Batch effect
  feature2 = c(rep(50, 10), rep(45, 10)) + rnorm(20, sd = 2)
)

rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_batch_reference(feature1, feature2, batch_col = "batch_id") |>
  prep()

corrected <- bake(rec, new_data = NULL)
```

`step_measure_bin` *Spectral Binning*

Description

`step_measure_bin()` creates a *specification* of a recipe step that reduces a spectrum to fewer points by averaging within bins.

Usage

```
step_measure_bin(
  recipe,
  n_bins = NULL,
  bin_width = NULL,
  method = c("mean", "sum", "median", "max"),
  measures = NULL,
  role = NA,
  trained = FALSE,
  bin_breaks = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_bin")
)
```

Arguments

`recipe`	A recipe object.
`n_bins`	Number of bins (mutually exclusive with `bin_width`).
`bin_width`	Width of each bin in location units (mutually exclusive with `n_bins`).
`method`	Aggregation method: `"mean"` (default), `"sum"`, `"median"`, or `"max"`.
`measures`	An optional character vector of measure column names.
`role`	Not used (modifies existing data).
`trained`	Logical indicating if the step has been trained.
`bin_breaks`	The computed bin breaks (after training).
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step reduces the number of points in each spectrum by dividing the x-axis into bins and aggregating values within each bin. The result replaces the `.measures` column with the binned data.

This is useful for:

- Reducing data dimensionality

- Decreasing noise through averaging
- Speeding up downstream processing
- Aligning data from different resolutions

The bin boundaries are determined during `prep()` from the training data and stored for consistent application to new data.

Value

An updated recipe with the new step added.

See Also

Other measure-features: `step_measure_integrals()`, `step_measure_moments()`, `step_measure_ratios()`

Examples

```
library(recipes)

# Bin to 20 points
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_bin(n_bins = 20) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_calibrate_x

Apply X-Axis Calibration

Description

`step_measure_calibrate_x()` creates a *specification* of a recipe step that transforms the x-axis (location) values using a calibration function or calibration data.

Usage

```
step_measure_calibrate_x(
  recipe,
  calibration,
  from = "x",
  to = "y",
  method = "spline",
  extrapolate = FALSE,
  measures = NULL,
  role = NA,
```

```
  trained = FALSE,
  cal_fn = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_calibrate_x")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`calibration`	The calibration to apply. Can be: • A data.frame with columns specified by `from` and `to` • A function that takes location values and returns calibrated values
`from`	Column name in calibration data.frame containing original x values. Default is `"x"`.
`to`	Column name in calibration data.frame containing calibrated values. Default is `"y"`.
`method`	Interpolation method when using calibration data.frame: • `"linear"`: Linear interpolation • `"spline"` (default): Cubic spline interpolation
`extrapolate`	Logical. If `TRUE`, allow extrapolation outside the calibration range. If `FALSE` (default), values outside the range will return `NA` for linear interpolation or use spline extrapolation.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`cal_fn`	The calibration function created during training.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

X-axis calibration is commonly used to convert raw measurement units to physically meaningful values. Common examples include:

- **GPC/SEC**: Convert retention time to molecular weight (via log MW)
- **Mass spectrometry**: Apply m/z calibration corrections
- **Spectroscopy**: Convert pixel or channel numbers to wavelength/wavenumber

The calibration can be provided as either:

1. **Calibration data**: A data.frame with known x→y mappings. The step will build an interpolation function during `prep()`.
2. **Calibration function**: A function that directly transforms x values.

Warning: This step modifies the `location` column. Subsequent steps will see the calibrated values. Make sure your calibration is appropriate for your data range.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `method`, `extrapolate`, and `id` is returned.

See Also

`step_measure_calibrate_y()` for y-axis calibration

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Example: GPC molecular weight calibration
# Calibration standards: retention_time -> log(MW)
gpc_cal <- data.frame(
  retention_time = c(10, 12, 14, 16, 18),
  log_mw = c(6.5, 5.8, 5.0, 4.2, 3.5)
)

# Note: meats_long doesn't have retention time, this is illustrative
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_calibrate_x(
    calibration = function(x) log10(x + 1),  # Example transformation
    method = "spline"
  )

# With calibration data
# rec <- recipe(...) |>
#   step_measure_calibrate_x(
#     calibration = gpc_cal,
#     from = "retention_time",
#     to = "log_mw",
#     method = "spline"
```

```
#   )
```

`step_measure_calibrate_y` *Apply Y-Axis Calibration (Response Factor)*

Description

`step_measure_calibrate_y()` creates a *specification* of a recipe step that applies a response factor or calibration function to y-axis (value) values.

Usage

```
step_measure_calibrate_y(
  recipe,
  response_factor = 1,
  calibration = NULL,
  measures = NULL,
  role = NA,
  trained = FALSE,
  cal_fn = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_calibrate_y")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`response_factor`	A numeric value to multiply all values by. Default is `1.0` (no change). This is a simple scalar calibration.
`calibration`	An optional calibration function that takes value(s) and returns calibrated value(s). If provided, this takes precedence over `response_factor`.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`cal_fn`	The calibration function to apply (built during prep).
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

Y-axis calibration is used to convert raw signal intensities to quantitative values. Common examples include:

- **Chromatography**: Apply detector response factors
- **Spectroscopy**: Apply molar absorptivity corrections
- **Mass spectrometry**: Apply ionization efficiency corrections

Simple mode: Use `response_factor` to multiply all values by a constant.

Complex mode: Use `calibration` to provide a function for non-linear calibration curves (e.g., from fitting standards).

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `response_factor`, `has_calibration`, and `id` is returned.

See Also

`step_measure_calibrate_x()` for x-axis calibration

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Simple response factor
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_calibrate_y(response_factor = 2.5)

# With calibration function (e.g., log transform)
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_calibrate_y(calibration = function(x) log10(x + 0.001))
```

`step_measure_center` *Mean Centering*

Description

`step_measure_center()` creates a *specification* of a recipe step that subtracts the mean at each measurement location (column-wise centering). The means are computed from the training data and applied to new data.

Usage

```
step_measure_center(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_params = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_center")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_params`	A named list containing learned means and locations for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Mean centering is a fundamental preprocessing step for multivariate analysis methods like PCA and PLS. It removes the average signal at each measurement location.

For a data matrix X with samples as rows and measurement locations as columns, the transformation is:

$$X_{centered} = X - \bar{X}$$

where $\bar{X}$ is the column-wise mean computed from the training data.

The means are learned during `prep()` from the training data and stored for use when applying the transformation to new data during `bake()`.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step after training, a tibble with the learned means at each location is returned.

See Also

`step_measure_scale_auto()`, `step_measure_scale_pareto()`

Other measure-scaling: `step_measure_scale_auto()`, `step_measure_scale_pareto()`, `step_measure_scale_range()`, `step_measure_scale_vast()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_center() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_channel_align` *Align Multiple Channels to a Common Grid*

Description

`step_measure_channel_align()` creates a *specification* of a recipe step that aligns multiple measurement channels to a common location grid.

Usage

```
step_measure_channel_align(
  recipe,
  ...,
  method = c("union", "intersection", "reference"),
  reference = 1L,
  interpolation = c("linear", "spline", "constant"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_channel_align")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose measure columns. If empty, all measure columns are used.
`method`	How to determine the common grid: • `"union"` (default): Use all unique locations from all channels • `"intersection"`: Use only locations present in all channels • `"reference"`: Use the grid from the reference channel
`reference`	For `method = "reference"`, which channel to use as reference. Can be a column name (character) or column index (integer). Default is 1 (first channel).
`interpolation`	Interpolation method for missing values: • `"linear"` (default): Linear interpolation • `"spline"`: Cubic spline interpolation • `"constant"`: Nearest neighbor (constant)
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Multi-channel analytical instruments (e.g., LC-DAD, SEC with multiple detectors) often produce measurements at slightly different location grids for each channel. This step aligns all channels to a common grid, enabling:

- Direct comparison between channels
- Channel combination or ratio calculations
- Modeling with consistent feature dimensions

Grid Methods:

- **Union**: Creates a grid containing all unique locations from all channels. Values are interpolated where channels don't have data.
- **Intersection**: Uses only locations where all channels have data. No interpolation needed but may lose data at edges.
- **Reference**: Uses one channel's grid as the target. Other channels are interpolated to match.

Value

An updated recipe with the new step added.

See Also

Other measure-channel: `step_measure_channel_combine()`, `step_measure_channel_ratio()`

Examples

```
library(recipes)
library(tibble)

# Create sample multi-channel data
df <- tibble(
  id = rep(1:3, each = 10),
  time_uv = rep(seq(0, 9, by = 1), 3),
  absorbance_uv = rnorm(30, 100, 10),
  time_ri = rep(seq(0.5, 9.5, by = 1), 3),
  absorbance_ri = rnorm(30, 50, 5),
  concentration = rep(c(10, 25, 50), each = 10)
)

# Ingest as separate channels, then align
rec <- recipe(concentration ~ ., data = df) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(absorbance_uv, location = vars(time_uv)) |>
  step_measure_input_long(absorbance_ri, location = vars(time_ri)) |>
  step_measure_channel_align(method = "union")
```

`step_measure_channel_combine` *Combine Multiple Channels*

Description

`step_measure_channel_combine()` creates a *specification* of a recipe step that combines multiple measurement channels into a single representation.

Usage

```
step_measure_channel_combine(
  recipe,
  ...,
  strategy = c("stack", "concat", "weighted_sum", "mean"),
  weights = NULL,
  output_col = ".measures",
  remove_original = TRUE,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_channel_combine")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose measure columns. If empty, all measure columns are used.
`strategy`	How to combine channels: • `"stack"`: Stack channels into an nD measurement with channel as a dimension • `"concat"`: Concatenate channels into a single 1D measurement • `"weighted_sum"`: Compute weighted sum across channels • `"mean"`: Average across channels (equal weights)
`weights`	For `strategy = "weighted_sum"`, a numeric vector of weights. Must have same length as number of channels. Default is equal weights.
`output_col`	Name of the output measure column. Default is `".measures"`.
`remove_original`	Logical. Should original channel columns be removed? Default is `TRUE`.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

After aligning multiple channels to a common grid with `step_measure_channel_align()`, this step combines them for downstream analysis. The choice of strategy depends on the analysis goal:

Strategies:

- **stack**: Creates an n-dimensional measurement where channel becomes a dimension. Useful for multi-way analysis (PARAFAC, Tucker).
- **concat**: Concatenates all channels end-to-end into a single long vector. Useful for PLS or other models that expect 1D input.
- **weighted_sum**: Computes a weighted combination of channel values at each location. Useful when channels should be fused into a single signal.
- **mean**: Simple average across channels (special case of weighted_sum).

Value

An updated recipe with the new step added.

Note

Channels must be aligned to the same grid before combining. Use `step_measure_channel_align()` first if grids differ.

See Also

Other measure-channel: `step_measure_channel_align()`, `step_measure_channel_ratio()`

Examples

```
library(recipes)
library(tibble)

# Create sample multi-channel data (already aligned)
df <- tibble(
  id = rep(1:3, each = 10),
  time = rep(seq(0, 9, by = 1), 3),
  uv = rnorm(30, 100, 10),
  ri = rnorm(30, 50, 5),
  concentration = rep(c(10, 25, 50), each = 10)
)

# Ingest and combine with stacking
rec <- recipe(concentration ~ ., data = df) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(uv, location = vars(time)) |>
  step_measure_input_long(ri, location = vars(time)) |>
  step_measure_channel_combine(strategy = "stack")
```

`step_measure_channel_ratio` *Compute Ratios Between Channels*

Description

`step_measure_channel_ratio()` creates a *specification* of a recipe step that computes ratios between pairs of measurement channels.

Usage

```
step_measure_channel_ratio(
  recipe,
  numerator,
  denominator,
  output_prefix = "ratio_",
  epsilon = 1e-10,
  log_transform = FALSE,
  remove_original = FALSE,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_channel_ratio")
)
```

Arguments

`recipe`	A recipe object.
`numerator`	Column name(s) for the numerator channel(s).
`denominator`	Column name(s) for the denominator channel(s). Must have same length as `numerator` (paired ratios).
`output_prefix`	Prefix for output column names. Default is `"ratio_"`.
`epsilon`	Small value added to denominator to avoid division by zero. Default is `1e-10`.
`log_transform`	Logical. Should the ratio be log-transformed? Default is `FALSE`.
`remove_original`	Logical. Should original channel columns be removed? Default is `FALSE`.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Channel ratios are useful in analytical chemistry for:

- **Normalization**: UV/RI ratios normalize for concentration variations
- **Identification**: Characteristic ratios help identify compounds
- **Quality control**: Ratio stability indicates system performance

Output Columns:
For each numerator/denominator pair, creates a new measure column named `{output_prefix}{numerator}_{de` (e.g., `"ratio_uv_ri"`).

Log Transform:
When `log_transform = TRUE`, computes `log(numerator / denominator)` which can be useful for:

- Normalizing skewed distributions
- Converting multiplicative relationships to additive
- Working with absorbance ratios

Value

An updated recipe with the new step added.

Note

Channels must be aligned to the same grid before computing ratios. Use `step_measure_channel_align()` first if grids differ.

See Also

Other measure-channel: `step_measure_channel_align()`, `step_measure_channel_combine()`

Examples

```
library(recipes)
library(tibble)

# Create sample multi-channel data
df <- tibble(
  id = rep(1:3, each = 10),
  time = rep(seq(0, 9, by = 1), 3),
  uv = rnorm(30, 100, 10),
  ri = rnorm(30, 50, 5),
  concentration = rep(c(10, 25, 50), each = 10)
)

# Compute UV/RI ratio
rec <- recipe(concentration ~ ., data = df) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(uv, location = vars(time)) |>
  step_measure_input_long(ri, location = vars(time)) |>
  step_measure_channel_ratio(numerator = "uv", denominator = "ri")
```

`step_measure_derivative` *Simple Finite Difference Derivatives*

Description

`step_measure_derivative()` creates a *specification* of a recipe step that computes derivatives using simple finite differences.

Usage

```
step_measure_derivative(
  recipe,
  order = 1L,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_derivative")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`order`	The order of the derivative (1 or 2). Default is `1` (first derivative).
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

This step computes derivatives using forward finite differences:

$$\frac{dy}{dx} \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

For each derivative order, the spectrum length is reduced by 1.

- First derivative: n-1 points
- Second derivative: n-2 points

The location values are updated to the left point of each difference.

Note: For smoothed derivatives, consider using `step_measure_savitzky_golay()` with `differentiation_order > 0` instead.

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_derivative_gap()` for gap derivatives, `step_measure_savitzky_golay()` for smoothed derivatives

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# First derivative
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_derivative(order = 1) |>
  prep()

# Second derivative
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_derivative(order = 2) |>
  prep()
```

---

step_measure_derivative_gap

Gap (Norris-Williams) Derivatives

Description

`step_measure_derivative_gap()` creates a *specification* of a recipe step that computes gap derivatives using the Norris-Williams method.

Usage

```
step_measure_derivative_gap(
  recipe,
  gap = 2L,
  segment = 1L,
  measures = NULL,
```

```
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_derivative_gap")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`gap`	The gap size (number of points to skip on each side). Default is `2`. The derivative at point i is computed from points i-gap and i+gap.
`segment`	The segment size for averaging. Default is `1` (no averaging). When greater than 1, multiple points are averaged on each side before computing the difference.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

Gap derivatives compute the difference between points separated by a gap:

$$\frac{dy}{dx} \approx \frac{y_{i+g} - y_{i-g}}{x_{i+g} - x_{i-g}}$$

where g is the gap size.

When `segment > 1`, the Norris-Williams method is used, which averages `segment` points on each side before computing the difference.

The spectrum length is reduced by `2 * gap` points.

Gap derivatives are often used in NIR chemometrics as an alternative to Savitzky-Golay derivatives when less smoothing is desired.

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_derivative()` for simple finite differences, `step_measure_savitzky_golay()` for smoothed derivatives

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_emsc()`, `step_measure_kubelka_m`

step_measure_log(), step_measure_map(), step_measure_msc(), step_measure_normalize_istd(), step_measure_osc(), step_measure_ratio_reference(), step_measure_snv(), step_measure_subtract_blan step_measure_subtract_reference(), step_measure_transmittance()

Examples

```
library(recipes)

# Gap derivative with gap=2
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_derivative_gap(gap = 2) |>
  prep()

# Norris-Williams with gap=3, segment=2
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_derivative_gap(gap = 3, segment = 2) |>
  prep()
```

---

step_measure_despike *Remove Spikes and Outliers from Measurements*

Description

`step_measure_despike()` creates a *specification* of a recipe step that detects and removes spikes (sudden, brief outliers) from measurement data. Spikes are common artifacts in spectroscopy (cosmic rays in Raman, detector glitches) and chromatography (electrical noise).

Usage

```
step_measure_despike(
  recipe,
  measures = NULL,
  window = 5L,
  threshold = 5,
  method = c("interpolate", "median", "mean"),
  max_width = 3L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_despike")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
window	The window size for local statistics. Must be an odd integer of at least 3. Default is 5. Tunable via smooth_window().
threshold	The threshold multiplier for spike detection. Points deviating more than threshold * MAD from the local median are flagged. Default is 5. Tunable via despike_threshold().
method	How to replace detected spikes. One of "interpolate" (default, linear interpolation from neighbors), "median" (replace with local median), or "mean" (replace with local mean).
max_width	Maximum width (in points) of a spike. Consecutive outliers wider than this are not considered spikes. Default is 3.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

Spike detection uses a robust local statistic approach:

1. For each point, calculate the local median and MAD (Median Absolute Deviation) within a sliding window
2. Flag points where |value - local_median| > threshold * MAD
3. Group consecutive flagged points into spike regions
4. If a spike region is narrower than max_width, replace with the specified method

MAD is scaled by 1.4826 to be consistent with standard deviation for normally distributed data.

This approach is robust because:

- Median and MAD are not affected by the spikes themselves
- The threshold adapts to local noise levels
- The max_width parameter prevents removing genuine peaks

Value

An updated recipe with the new step added.

See Also

Other measure-smoothing: step_measure_filter_fourier(), step_measure_savitzky_golay(), step_measure_smooth_gaussian(), step_measure_smooth_ma(), step_measure_smooth_median(), step_measure_smooth_wavelet()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_despike(threshold = 5) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_detrend *Remove Trend from Measurements*

Description

`step_measure_detrend()` creates a *specification* of a recipe step that removes a polynomial trend from measurement data. This is useful for removing drift, offset, or slowly varying background effects.

Usage

```
step_measure_detrend(
  recipe,
  measures = NULL,
  degree = 1L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_detrend")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`degree`	Polynomial degree for trend fitting. Default is `1` (linear detrending). Use `0` to remove only the mean (centering).
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

Detrending removes a polynomial trend from each spectrum. This is simpler than baseline correction methods like ALS or robust fitting, but effective for:

- **Linear drift** (`degree = 1`): Instrumental drift, temperature effects
- **Offset removal** (`degree = 0`): Centers each spectrum at zero mean
- **Curved trends** (`degree = 2+`): Gradual curvature from scattering

Unlike `step_measure_baseline_poly()`, detrending fits the polynomial to ALL points without iterative peak exclusion. This makes it faster and appropriate when:

- The trend is the dominant feature (not peaks)
- You want to preserve peak structure while removing background
- Processing time-series or process data with drift

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `degree`, and `id` is returned.

See Also

`step_measure_baseline_poly()` for baseline correction with peak exclusion.

Other measure-baseline: `step_measure_baseline_airpls()`, `step_measure_baseline_als()`, `step_measure_baseline_arpls()`, `step_measure_baseline_aspls()`, `step_measure_baseline_auto()`, `step_measure_baseline_custom()`, `step_measure_baseline_fastchrom()`, `step_measure_baseline_gpc()`, `step_measure_baseline_iarpls()`, `step_measure_baseline_minima()`, `step_measure_baseline_morph()`, `step_measure_baseline_morphological()`, `step_measure_baseline_poly()`, `step_measure_baseline_py()`, `step_measure_baseline_rf()`, `step_measure_baseline_rolling()`, `step_measure_baseline_snip()`, `step_measure_baseline_tophat()`

Examples

```
library(recipes)

# Linear detrending
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_detrend(degree = 1) |>
  prep()

bake(rec, new_data = NULL)
```

```
# Mean centering only (degree = 0)
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_detrend(degree = 0) |>
  prep()
```

`step_measure_dilution_correct` *Dilution Factor Correction*

Description

`step_measure_dilution_correct()` creates a *specification* of a recipe step that corrects concentration values by applying dilution factors. This is essential when samples are diluted during preparation and need to be back-calculated to original concentrations.

Usage

```
step_measure_dilution_correct(
  recipe,
  ...,
  dilution_col = "dilution_factor",
  operation = c("multiply", "divide"),
  handle_zero = c("error", "warn", "skip"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_dilution_correct")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns (concentration values) to correct. If empty, all numeric columns (excluding metadata columns) will be selected.
`dilution_col`	Name of the column containing dilution factors. Default is `"dilution_factor"`.
`operation`	How to apply the dilution factor: • `"multiply"` (default): `concentration * dilution_factor` (back-calculate from diluted to original concentration) • `"divide"`: `concentration / dilution_factor` (apply dilution)
`handle_zero`	How to handle zero dilution factors: • `"error"` (default): Stop with an error • `"warn"`: Warn and set result to NA

- `"skip"`: Silently set result to NA

`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Dilution Factor Interpretation:

The dilution factor represents how much the sample was diluted:

- A factor of 1 means no dilution (undiluted)
- A factor of 2 means 1:2 dilution (1 part sample + 1 part diluent)
- A factor of 10 means 1:10 dilution

Back-Calculation:

When using `operation = "multiply"` (the default): `original_concentration = measured_concentration * dilution_factor`

This corrects for the dilution to get the true concentration in the original sample.

When to Use:

Use this step after quantitation (calibration) when samples were diluted to bring concentrations within the calibration range.

Value

An updated recipe with the new step added.

See Also

`step_measure_surrogate_recovery()`, `measure_calibration_predict()`

Other calibration: `measure_matrix_effect()`, `step_measure_standard_addition()`, `step_measure_surrogate`

Examples

```
library(recipes)

# Example: samples diluted to fit calibration range
data <- data.frame(
  sample_id = paste0("S", 1:6),
  dilution_factor = c(1, 2, 5, 10, 1, 1),
  analyte = c(50, 45, 42, 48, 51, 49)  # Measured after dilution
)

rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_dilution_correct(
    analyte,
    dilution_col = "dilution_factor",
```

```
    operation = "multiply"
  ) |>
  prep()

# Back-calculated concentrations
bake(rec, new_data = NULL)
# S1: 50*1=50, S2: 45*2=90, S3: 42*5=210, S4: 48*10=480
```

`step_measure_drift_linear` *Linear Drift Correction*

Description

`step_measure_drift_linear()` creates a *specification* of a recipe step that corrects for linear signal drift across run order using QC or reference samples. This is a simpler alternative to LOESS when drift is approximately linear.

Usage

```
step_measure_drift_linear(
  recipe,
  ...,
  run_order_col = "run_order",
  sample_type_col = "sample_type",
  qc_type = "qc",
  apply_to = c("all", "unknown"),
  min_qc = 3,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_drift_linear")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns. For feature-level data, select the numeric response columns. For curve-level data with `.measures`, leave empty to apply to all locations.
`run_order_col`	Name of the column containing run order (injection sequence). Must be numeric/integer.
`sample_type_col`	Name of the column containing sample type.
`qc_type`	Value(s) in `sample_type_col` that identify QC samples to use for drift modeling. Default is `"qc"`.
`apply_to`	Which samples to apply correction to:

- `"all"` (default): Correct all samples
- `"unknown"`: Only correct unknown samples

`min_qc`	Minimum number of QC samples required. Default is 5.
`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

How It Works:

1. During `prep()`: A linear regression is fit to QC sample responses vs run order for each feature.
2. During `bake()`: Correction factors are calculated as: `correction = median(QC_responses) / predicted_value`
   Each sample's response is multiplied by the correction factor.

When to Use:

Use linear drift correction when:

- Drift is approximately linear over the run
- You have fewer QC samples (requires at least 3)
- You want a more conservative correction

For non-linear drift patterns, use `step_measure_drift_qc_loess()` or `step_measure_drift_spline()`.

Value

An updated recipe with the new step added.

See Also

`step_measure_drift_qc_loess()` for LOESS-based correction, `step_measure_drift_spline()` for spline-based correction.

Other drift-correction: `step_measure_drift_qc_loess()`, `step_measure_drift_spline()`, `step_measure_qc_bracket()`

Examples

```
library(recipes)

# Data with linear drift
data <- data.frame(
  sample_id = paste0("S", 1:20),
  sample_type = rep(c("qc", "unknown", "unknown", "unknown", "qc"), 4),
  run_order = 1:20,
  feature1 = 100 + (1:20) * 0.5 + rnorm(20, sd = 2)
)
```

```
rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_drift_linear(feature1) |>
  prep()

corrected <- bake(rec, new_data = NULL)
```

`step_measure_drift_qc_loess` *QC-Based Drift Correction Using LOESS*

Description

`step_measure_drift_qc_loess()` creates a *specification* of a recipe step that corrects for signal drift across run order using QC (or reference) samples. This implements the QC-RLSC (robust LOESS signal correction) method.

Usage

```
step_measure_drift_qc_loess(
  recipe,
  ...,
  run_order_col = "run_order",
  sample_type_col = "sample_type",
  qc_type = "qc",
  apply_to = c("all", "unknown"),
  span = 0.75,
  degree = 2,
  robust = TRUE,
  min_qc = 5,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_drift_qc_loess")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns. For feature-level data, select the numeric response columns. For curve-level data with `.measures`, leave empty to apply to all locations.
`run_order_col`	Name of the column containing run order (injection sequence). Must be numeric/integer.
`sample_type_col`	Name of the column containing sample type.

`qc_type`	Value(s) in `sample_type_col` that identify QC samples to use for drift modeling. Default is `"qc"`.
`apply_to`	Which samples to apply correction to: • `"all"` (default): Correct all samples • `"unknown"`: Only correct unknown samples
`span`	LOESS span parameter controlling smoothness. Default is 0.75. Smaller values = more flexible fit.
`degree`	Polynomial degree for LOESS (1 or 2). Default is 2.
`robust`	Logical. Use robust LOESS fitting? Default is TRUE.
`min_qc`	Minimum number of QC samples required. Default is 5.
`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

How It Works:

1. During `prep()`: A LOESS model is fit to QC sample responses vs run order for each feature/location.
2. During `bake()`: Correction factors are calculated as: `correction = median(QC_responses) / predicted_value`
   Each sample's response is multiplied by the correction factor at its run order position.

Data Levels:

This step supports both:

- **Feature-level data**: Applies correction to each selected numeric column
- **Curve-level data**: Applies correction to each location in the measure_list

Diagnostics:

The trained step stores drift model information accessible via `tidy()`:

- LOESS model parameters
- QC response trends
- Correction factors applied

Value

An updated recipe with the new step added.

See Also

`measure_detect_drift()` for drift detection before correction.

Other drift-correction: `step_measure_drift_linear()`, `step_measure_drift_spline()`, `step_measure_qc_bracket()`

Examples

```
library(recipes)

# Feature-level data with drift
data <- data.frame(
  sample_id = paste0("S", 1:20),
  sample_type = rep(c("qc", "unknown", "unknown", "unknown", "qc"), 4),
  run_order = 1:20,
  feature1 = 100 + (1:20) * 0.5 + rnorm(20, sd = 2),  # Upward drift
  feature2 = 50 - (1:20) * 0.3 + rnorm(20, sd = 1)    # Downward drift
)

rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_drift_qc_loess(feature1, feature2) |>
  prep()

corrected <- bake(rec, new_data = NULL)
```

---

step_measure_drift_spline

Spline-Based Drift Correction

Description

`step_measure_drift_spline()` creates a *specification* of a recipe step that corrects for signal drift using smoothing splines fit to QC samples. This offers more flexibility than linear correction while being more stable than LOESS for sparse QC data.

Usage

```
step_measure_drift_spline(
  recipe,
  ...,
  run_order_col = "run_order",
  sample_type_col = "sample_type",
  qc_type = "qc",
  apply_to = c("all", "unknown"),
  df = NULL,
  spar = NULL,
  min_qc = 4,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_drift_spline")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns. For feature-level data, select the numeric response columns. For curve-level data with `.measures`, leave empty to apply to all locations.
`run_order_col`	Name of the column containing run order (injection sequence). Must be numeric/integer.
`sample_type_col`	Name of the column containing sample type.
`qc_type`	Value(s) in `sample_type_col` that identify QC samples to use for drift modeling. Default is `"qc"`.
`apply_to`	Which samples to apply correction to: • `"all"` (default): Correct all samples • `"unknown"`: Only correct unknown samples
`df`	Degrees of freedom for the smoothing spline. Default is NULL, which uses cross-validation to select optimal df. Lower values = smoother.
`spar`	Smoothing parameter (alternative to df). If NULL (default), cross-validation is used.
`min_qc`	Minimum number of QC samples required. Default is 5.
`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

How It Works:

Uses `stats::smooth.spline()` to fit a flexible curve through QC responses. The spline automatically adapts to the data complexity when `df` is not specified.

Comparison with Other Methods:

Method	Best For	Min QC Samples
Linear	Simple linear drift	3
Spline	Moderate non-linearity	4+
LOESS	Complex patterns	5+

Value

An updated recipe with the new step added.

See Also

`step_measure_drift_linear()` for linear correction, `step_measure_drift_qc_loess()` for LOESS-based correction.

Other drift-correction: `step_measure_drift_linear()`, `step_measure_drift_qc_loess()`, `step_measure_qc_bracket()`

Examples

```
library(recipes)

# Data with non-linear drift
set.seed(123)
data <- data.frame(
  sample_id = paste0("S", 1:30),
  sample_type = rep(c("qc", "unknown", "unknown", "unknown", "unknown", "qc"), 5),
  run_order = 1:30,
  feature1 = 100 + sin((1:30) / 5) * 10 + rnorm(30, sd = 2)
)

rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_drift_spline(feature1) |>
  prep()

corrected <- bake(rec, new_data = NULL)
```

`step_measure_emsc` *Extended Multiplicative Scatter Correction (EMSC)*

Description

`step_measure_emsc()` creates a *specification* of a recipe step that applies Extended Multiplicative Scatter Correction to spectral data. EMSC accounts for wavelength-dependent scatter effects using polynomial terms.

Usage

```
step_measure_emsc(
  recipe,
  degree = 2L,
  reference = "mean",
  measures = NULL,
  role = NA,
  trained = FALSE,
  ref_spectrum = NULL,
  locations = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_emsc")
)
```

Arguments

`recipe`	A recipe object.
`degree`	Polynomial degree for wavelength-dependent terms. Default is 2. Higher values can model more complex scatter effects but risk overfitting.
`reference`	Reference spectrum method: `"mean"` (default) or `"median"`. Alternatively, a numeric vector can be supplied as the reference spectrum.
`measures`	An optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`ref_spectrum`	The learned reference spectrum (after training).
`locations`	The location values for polynomial terms (after training).
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Extended MSC (EMSC) extends standard MSC by modeling wavelength-dependent scatter effects. For a spectrum x_i and reference x_r, the model is:

$$x_i = a_i + b_i \cdot x_r + c_i \cdot \lambda + d_i \cdot \lambda^2 + ... + \epsilon$$

The corrected spectrum is:

$$EMSC(x_i) = \frac{x_i - a_i - c_i \cdot \lambda - d_i \cdot \lambda^2 - ...}{b_i}$$

The polynomial terms (λ, λ^2, etc.) account for wavelength-dependent baseline effects that vary between samples.

When to use EMSC vs MSC:

- Use MSC for simple additive/multiplicative scatter
- Use EMSC when scatter effects vary with wavelength
- Start with degree=2, increase if needed for complex scatter

Value

An updated recipe with the new step added.

See Also

`step_measure_msc()` for standard MSC

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_emsc(degree = 2) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_exclude` *Exclude Measurement Ranges*

Description

`step_measure_exclude()` creates a *specification* of a recipe step that removes measurement points within the specified x-axis range(s).

Usage

```
step_measure_exclude(
  recipe,
  ranges,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_exclude")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`ranges`	A list of numeric vectors, each of length 2 specifying ranges to exclude as `c(min, max)`. Points with location >= min and <= max in any range are removed.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

This step removes measurements falling within specified ranges. This is useful for:

- Removing solvent peaks in chromatography
- Excluding system peaks or artifacts
- Removing detector saturation regions
- Removing known interference regions in spectroscopy

Multiple ranges can be excluded by providing a list of ranges. Points falling within any of the specified ranges are removed.

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_trim()` for keeping specific ranges, `step_measure_resample()` for interpolating to a new grid

Other region-operations: `step_measure_resample()`, `step_measure_trim()`

Examples

```
library(recipes)

# Exclude specific regions (e.g., solvent peaks)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_exclude(ranges = list(c(1, 5), c(95, 100))) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_filter_fourier

Fourier Low-Pass Filtering

Description

`step_measure_filter_fourier()` creates a *specification* of a recipe step that applies Fourier-domain low-pass filtering to remove high-frequency noise.

Usage

```
step_measure_filter_fourier(
  recipe,
  measures = NULL,
  cutoff = 0.1,
  type = c("lowpass", "highpass"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_filter_fourier")
)
```

Arguments

recipe	A recipe object.
measures	An optional character vector of measure column names.
cutoff	The cutoff frequency as a fraction of the Nyquist frequency (0 to 0.5). Default is 0.1. Frequencies above this are attenuated. Tunable via fourier_cutoff().
type	Type of filter: "lowpass" (default) keeps low frequencies, "highpass" keeps high frequencies.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

Fourier filtering transforms the spectrum to the frequency domain using FFT, applies a frequency mask, and transforms back. This is effective for:

- Removing periodic noise
- Smoothing with precise frequency control
- Removing high-frequency detector noise

The cutoff is specified as a fraction of the Nyquist frequency. A cutoff of 0.1 keeps only the lowest 10% of frequencies.

Value

An updated recipe with the new step added.

See Also

Other measure-smoothing: step_measure_despike(), step_measure_savitzky_golay(), step_measure_smooth_gaussian(), step_measure_smooth_ma(), step_measure_smooth_median(), step_measure_smooth_wavelet()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_filter_fourier(cutoff = 0.1) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_impute *Impute Missing Values in Measurements*

Description

`step_measure_impute()` creates a *specification* of a recipe step that imputes (fills in) missing values (NA) in measurement data using interpolation or other methods.

Usage

```
step_measure_impute(
  recipe,
  measures = NULL,
  method = c("linear", "spline", "constant", "mean"),
  max_gap = Inf,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_impute")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`method`	Imputation method: • `"linear"` (default): Linear interpolation • `"spline"`: Cubic spline interpolation • `"constant"`: Nearest non-NA value • `"mean"`: Global mean of non-NA values
`max_gap`	Maximum gap size to impute. Gaps larger than this are left as NA. Default is `Inf` (impute all).
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Missing values can occur due to:

- Removed spikes (after despiking with replacement set to NA)
- Excluded regions
- Instrument gaps or dropouts

Linear and spline interpolation use the `stats::approx()` and `stats::spline()` functions respectively. They are most appropriate when gaps are small relative to spectral features.

Value

An updated recipe with the new step added.

See Also

Other measure-qc: `step_measure_qc_outlier()`, `step_measure_qc_saturated()`, `step_measure_qc_snr()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_impute(method = "linear") |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_input_long

Ingest Measurements from a Single Column

Description

`step_measure_input_long` creates a *specification* of a recipe step that converts measures organized in a column for the analytical results (and one or more columns of numeric indices) into an internal format used by the package.

Usage

```
step_measure_input_long(
  recipe,
  ...,
  location,
  col_name = ".measures",
  dim_names = NULL,
```

```
  dim_units = NULL,
  pad = FALSE,
  role = "measure",
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("measure_input_long")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose which *single* column contains the analytical measurements. The selection should be in the order of the measurement's profile.
`location`	One or more selector functions to choose which column(s) have the locations of the analytical values. For 1D data (spectra, chromatograms), select a single location column. For 2D or higher dimensional data (LC-DAD, 2D NMR, EEM), select multiple location columns. Columns will be renamed to `location_1`, `location_2`, etc. in order.
`col_name`	A single character string specifying the name of the output column that will contain the measure data. Defaults to `".measures"`. Use different names when creating multiple measure columns (e.g., `".uv_spectrum"` and `".ms_spectrum"`).
`dim_names`	Optional character vector of semantic names for each dimension (e.g., `c("retention_time", "wavelength")`). Only used for multi-dimensional data.
`dim_units`	Optional character vector of units for each dimension (e.g., `c("min", "nm")`). Only used for multi-dimensional data.
`pad`	Whether to pad the measurements to ensure that they all have the same number of values. This is useful when there are missing values in the measurements.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character vector of column names determined by the recipe.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step is designed for data in a format where there is a column for the analytical measurement (e.g., absorption, etc.) and one or more columns with the location of the value (e.g., wave number, retention time, wavelength, etc.).

`step_measure_input_long()` will collect those data and put them into a format used internally by this package. The data structure has a row for each independent experimental unit and a nested tibble with that sample's measure (measurement and location). It assumes that there are unique combinations of the other columns in the data that define individual patterns associated with the pattern. If this is not the case, the special values might be inappropriately restructured.

The best advice is to have a column of any type that indicates the unique sample number for each measure. For example, if there are 200 values in the measure and 7 samples, the input data (in long format) should have 1,400 rows. We advise having a column with 7 unique values indicating which of the rows correspond to each sample.

Multi-Dimensional Data

For 2D or higher dimensional data, provide multiple location columns:

```
# LC-DAD data with retention time and wavelength
step_measure_input_long(
  absorbance,
  location = vars(retention_time, wavelength),
  dim_names = c("time", "wavelength"),
  dim_units = c("min", "nm")
)
```

The result will be a `measure_nd_list` column instead of a `measure_list`.

Missing Data

Currently, **measure** assumes that there are equal numbers of values within a sample. If there are missing values in the measurements, you'll need to pad them with missing values (as opposed to an absent row in the long format). If not, an error will occur.

Tidying

When you `tidy()` this step, a tibble indicating which of the original columns were used to reformat the data.

See Also

Other input/output steps: `step_measure_input_wide()`, `step_measure_output_long()`, `step_measure_output_wide()`

Examples

```
library(recipes)

# 1D data (traditional usage)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_input_wide` *Ingest Measurements in Separate Columns*

Description

`step_measure_input_wide` creates a *specification* of a recipe step that converts measures organized in multiple columns into an internal format used by the package.

Usage

```
step_measure_input_wide(
  recipe,
  ...,
  role = "measure",
  trained = FALSE,
  columns = NULL,
  location_values = NULL,
  col_name = ".measures",
  skip = FALSE,
  id = rand_id("measure_input_wide")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose variables for this step. See `selections()` for more details.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of the selected variable names. This field is a placeholder and will be populated once `recipes::prep()` is used.

`location_values`
: A numeric vector of values that specify the location of the measurements (e.g., wavelength etc.) in the same order as the variables selected by .... If not specified, a sequence of integers (starting at 1L) is used.

`col_name`
: A single character string specifying the name of the output column that will contain the measure data. Defaults to `".measures"`. Use different names when creating multiple measure columns (e.g., `".uv_spectrum"` and `".ms_spectrum"`).

`skip`
: A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`
: A character string that is unique to this step to identify it.

Details

This step is designed for data in a format where the analytical measurements are in separate columns.

`step_measure_input_wide()` will collect those data and put them into a format used internally by this package. The data structure has a row for each independent experimental unit and a nested tibble with that sample's measure (measurement and location). It assumes that there are unique combinations of the other columns in the data that define individual patterns associated with the pattern. If this is not the case, the special values might be inappropriately restructured.

The best advice is to have a column of any type that indicates the unique sample number for each measure. For example, if there are 20 rows in the input data set, the columns that are *not* analytically measurements show have no duplicate combinations in the 20 rows.

Tidying

When you `tidy()` this step, a tibble indicating which of the original columns were used to reformat the data.

See Also

Other input/output steps: `step_measure_input_long()`, `step_measure_output_long()`, `step_measure_output_wide()`

Examples

```
data(meats, package = "modeldata")

# Outcome data is to the right
names(meats) |> tail(10)

# ------------------------------------------------------------------------------
# Ingest data without adding the location (i.e. wave number) for the spectra
```

```
rec <-
  recipe(water + fat + protein ~ ., data = meats) |>
  step_measure_input_wide(starts_with("x_")) |>
  prep()

summary(rec)

# ------------------------------------------------------------------------------
# Ingest data without adding the location (i.e. wave number) for the spectra

# Make up some locations for the spectra's x-axis
index <- seq(1, 2, length.out = 100)

rec <-
  recipe(water + fat + protein ~ ., data = meats) |>
  step_measure_input_wide(starts_with("x_"), location_values = index) |>
  prep()

summary(rec)
```

`step_measure_integrals` *Calculate Region Integrals*

Description

`step_measure_integrals()` creates a *specification* of a recipe step that calculates integrated areas for specified x-axis regions.

Usage

```
step_measure_integrals(
  recipe,
  regions,
  method = c("trapezoid", "simpson"),
  measures = NULL,
  prefix = "integral_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_integrals")
)
```

Arguments

`recipe`	A recipe object.

`regions`	A named or unnamed list of numeric vectors, each of length 2 specifying regions as `c(min, max)`. For example: `list(peak1 = c(1000, 1100), peak2 = c(1500, 1600))`.
`method`	Integration method: `"trapezoid"` (default) or `"simpson"`.
`measures`	An optional character vector of measure column names.
`prefix`	Prefix for output column names. Default is `"integral_"`.
`role`	Role for generated columns. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step calculates the integrated area under the curve for each specified region. The result is added as new predictor columns, one per region.

Column naming:

- If regions are named: `prefix + name` (e.g., `"integral_peak1"`)
- If regions are unnamed: `prefix + index` (e.g., `"integral_1"`)

Integration methods:

- `"trapezoid"`: Trapezoidal rule, fast and accurate for smooth data
- `"simpson"`: Simpson's rule, more accurate for smooth curves

Value

An updated recipe with the new step added.

See Also

Other measure-features: `step_measure_bin()`, `step_measure_moments()`, `step_measure_ratios()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_integrals(
    regions = list(low = c(1, 30), mid = c(40, 60), high = c(70, 100))
  ) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_interpolate` *Interpolate Gaps in Measurement Data*

Description

`step_measure_interpolate()` creates a *specification* of a recipe step that fills gaps or missing values in measurement data using interpolation.

Usage

```
step_measure_interpolate(
  recipe,
  ranges,
  method = c("linear", "spline"),
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_interpolate")
)
```

Arguments

`recipe`	A recipe object.
`ranges`	A list of numeric vectors specifying ranges to interpolate. Each element should be a vector of length 2: `c(min, max)`.
`method`	Interpolation method: "linear" or "spline". Default is "linear".
`measures`	An optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step is useful for:

- Filling gaps left by excluded regions that need restoration
- Handling missing or invalid data points
- Smoothing over detector saturation regions

The interpolation uses data points immediately outside the specified ranges to estimate values within the ranges.

Value

An updated recipe with the new step added.

Examples

```
library(recipes)

# Interpolate over a problematic region
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_interpolate(ranges = list(c(40, 50)), method = "spline") |>
  prep()
```

---

step_measure_kubelka_munk

Kubelka-Munk Transformation

Description

`step_measure_kubelka_munk()` creates a *specification* of a recipe step that applies the Kubelka-Munk transformation for diffuse reflectance data.

Usage

```
step_measure_kubelka_munk(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_kubelka_munk")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

The Kubelka-Munk transformation is used for diffuse reflectance spectroscopy to convert reflectance to a quantity proportional to concentration:

$$f(R) = \frac{(1 - R)^2}{2R}$$

where R is the reflectance (0 to 1).

Important: Reflectance values should be in the range (0, 1). Values at the boundaries will produce extreme values or `Inf`.

This transformation is commonly used in:

- NIR diffuse reflectance spectroscopy
- Analysis of powders and solid samples
- When Beer-Lambert law doesn't apply directly

The measurement locations are preserved unchanged.

Value

An updated version of `recipe` with the new step added.

See Also

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Assuming reflectance data in (0, 1) range
# Note: meats_long has transmittance, this is illustrative
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_kubelka_munk()
```

`step_measure_log` *Log Transformation*

Description

`step_measure_log()` creates a *specification* of a recipe step that applies a logarithmic transformation to measurement values.

Usage

```
step_measure_log(
  recipe,
  base = exp(1),
  offset = 0,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_log")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`base`	The base of the logarithm. Default is `exp(1)` (natural log). Use `10` for log10 transformation.
`offset`	A numeric offset added to values before taking the log. Default is `0`. Use a small positive value (e.g., `1` for `log1p`) to handle zero or near-zero values.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

This step applies the transformation:

$$y' = \log_b(y + \text{offset})$$

where b is the base.

Log transformation is commonly used for:

- Variance stabilization
- Normalizing skewed distributions
- Converting multiplicative relationships to additive

Warning: Non-positive values (after offset) will produce `-Inf` or `NaN`.

The measurement locations are preserved unchanged.

Value

An updated version of `recipe` with the new step added.

See Also

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Natural log transformation
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_log(offset = 1) |>
  prep()

# Log10 transformation
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_log(base = 10) |>
  prep()
```

---

step_measure_map *Apply a Custom Function to Measurements*

Description

`step_measure_map()` creates a *specification* of a recipe step that applies a custom function to each sample's measurements. Use this when the built-in preprocessing steps (SNV, MSC, Savitzky-Golay) don't cover your needs.

Usage

```
step_measure_map(
  recipe,
  fn,
  ...,
  measures = NULL,
  verbosity = 1L,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_map")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`fn`	A function to apply to each sample's measurement tibble. The function should accept a tibble with `location` and `value` columns and return a tibble with the same structure. Can also be a formula (e.g., `~ { .x$value <- log1p(.x$value); .x }`) which will be converted via `rlang::as_function()`.
`...`	Additional arguments passed to `fn` during baking.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`verbosity`	An integer controlling output verbosity: • `0`: Silent - suppress all messages and output from `fn` • `1`: Normal (default) - show output from `fn`
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step.

Details

This step is the "escape hatch" for custom sample-wise transformations that aren't covered by the built-in steps. It integrates fully with the recipes framework, meaning your custom transformation will be:

- Applied consistently during `prep()` and `bake()`
- Included when bundling recipes into workflows
- Reproducible across sessions

Function Requirements:

The function `fn` must:

- Accept a tibble with `location` and `value` columns
- Return a tibble with `location` and `value` columns
- Not change the number of rows (measurements must remain aligned)

When to Use This Step:

Use `step_measure_map()` for domain-specific transformations not covered by the built-in steps:

- Custom baseline correction algorithms
- Specialized normalization methods
- Instrument-specific corrections
- Experimental preprocessing techniques

For common operations, prefer the built-in steps:

- Scatter correction → `step_measure_snv()` or `step_measure_msc()`
- Smoothing/derivatives → `step_measure_savitzky_golay()`

Prototyping with measure_map():

When developing a custom transformation, you may find it helpful to prototype using `measure_map()` on baked data before wrapping it in a step. Once your function works correctly, use 'step_measure_

for production pipelines.

Value

An updated version of `recipe` with the new step added.

See Also

- `step_measure_snv()`, `step_measure_msc()`, `step_measure_savitzky_golay()` for built-in preprocessing steps
- `measure_map()` for prototyping custom transformations

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Example 1: Custom log transformation
log_transform <- function(x) {
  x$value <- log1p(x$value)
  x
}

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
```

```
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_map(log_transform) |>
  step_measure_snv() |>
  prep()

bake(rec, new_data = NULL)

# Example 2: Using formula syntax for inline transformations
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_map(~ {
    # Subtract minimum to remove offset
    .x$value <- .x$value - min(.x$value)
    .x
  }) |>
  prep()

# Example 3: Using external package functions
# (e.g., custom baseline from a spectroscopy package)
## Not run:
rec3 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_map(my_baseline_correction, method = "als") |>
  step_measure_output_wide()

## End(Not run)
```

`step_measure_mcr_als` *MCR-ALS Decomposition for Multi-Dimensional Data*

Description

`step_measure_mcr_als()` creates a *specification* of a recipe step that applies Multivariate Curve Resolution - Alternating Least Squares (MCR-ALS) to multi-dimensional measurement data.

Usage

```
step_measure_mcr_als(
  recipe,
  ...,
  n_components = 3L,
  max_iter = 500L,
  tol = 1e-06,
  non_negativity = TRUE,
  unimodality = FALSE,
```

```
  prefix = "mcr_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_mcr_als")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose measure columns. If empty, all nD measure columns are used.
`n_components`	Number of components to extract. Default is 3.
`max_iter`	Maximum number of iterations. Default is 500.
`tol`	Convergence tolerance. Default is 1e-6.
`non_negativity`	Logical. Should non-negativity constraints be applied? Default is `TRUE`.
`unimodality`	Logical. Should unimodality constraints be applied? Default is `FALSE`.
`prefix`	Prefix for output column names. Default is `"mcr_"`.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

MCR-ALS is a powerful technique for resolving mixtures into pure component contributions. It's particularly useful for:

- Chromatographic data (time x wavelength)
- Spectroscopic mixtures
- Process analytical data

Unlike PARAFAC, MCR-ALS is a bilinear method that works on 2D data (samples unfolded if 3D). It allows flexible constraints like non-negativity and unimodality.

Experimental Status:

This step is experimental and its API may change in future versions.

Requirements:

- Input must be `measure_nd_list` with 2 dimensions
- All samples must have the same grid (regular, aligned)

Value

An updated recipe with the new step added.

Note

This is an **experimental** feature. The implementation uses a simple ALS algorithm without advanced constraints. For production use, consider using dedicated MCR-ALS packages.

See Also

`step_measure_parafac()` for PARAFAC decomposition

Other measure-multiway: `step_measure_parafac()`, `step_measure_tucker()`

Examples

```
## Not run:
library(recipes)

# After ingesting chromatographic data
rec <- recipe(concentration ~ ., data = chrom_data) |>
  step_measure_input_long(
    absorbance,
    location = vars(time, wavelength)
  ) |>
  step_measure_mcr_als(n_components = 3) |>
  prep()

bake(rec, new_data = NULL)

## End(Not run)
```

`step_measure_moments` *Calculate Statistical Moments*

Description

`step_measure_moments()` creates a *specification* of a recipe step that calculates statistical moments from spectra.

Usage

```
step_measure_moments(
  recipe,
  moments = c("mean", "sd", "skewness", "kurtosis"),
  weighted = FALSE,
  measures = NULL,
  prefix = "moment_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_moments")
)
```

Arguments

recipe	A recipe object.
moments	Character vector specifying which moments to calculate. Options: "mean", "sd", "skewness", "kurtosis", "entropy". Default is c("mean", "sd", "skewness", "kurtosis").
weighted	Logical. If TRUE, moments are weighted by location values. Default is FALSE.
measures	An optional character vector of measure column names.
prefix	Prefix for output column names. Default is "moment_".
role	Role for generated columns. Default is "predictor".
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

This step calculates statistical moments that summarize the distribution of values in each spectrum:

Moment	Description
mean	Mean value of the spectrum
sd	Standard deviation of values
skewness	Asymmetry of the distribution
kurtosis	"Tailedness" of the distribution
entropy	Shannon entropy (requires positive values)

When weighted = TRUE, the location (x-axis) values are used as weights, which can be useful for calculating center of mass or weighted statistics.

Value

An updated recipe with the new step added.

See Also

Other measure-features: step_measure_bin(), step_measure_integrals(), step_measure_ratios()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_moments(moments = c("mean", "sd", "skewness")) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_msc` *Multiplicative Scatter Correction (MSC)*

Description

`step_measure_msc()` creates a *specification* of a recipe step that applies Multiplicative Scatter Correction to spectral data. MSC removes physical light scatter by accounting for additive and multiplicative effects.

Usage

```
step_measure_msc(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  ref_spectra = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_msc")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`ref_spectra`	A named list of numeric vectors containing the reference spectra computed during training for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Multiplicative Scatter Correction (MSC) is a normalization method that attempts to account for additive and multiplicative effects by aligning each spectrum to a reference spectrum. For a spectrum x_i and reference x_r, the transformation is:

$$x_i = m_i \cdot x_r + a_i$$

$$MSC(x_i) = \frac{x_i - a_i}{m_i}$$

where a_i and m_i are the additive (intercept) and multiplicative (slope) terms from regressing x_i on x_r.

The reference spectrum is computed as the mean of all training spectra during `prep()` and stored for use when applying the transformation to new data.

MSC is commonly used to remove physical light scatter effects in NIR spectroscopy caused by differences in particle size or path length.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

The measurement locations are preserved unchanged.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms` (set to `".measures"`) and `id` is returned.

References

Geladi, P., MacDougall, D., and Martens, H. 1985. Linearization and Scatter-Correction for Near-Infrared Reflectance Spectra of Meat. Applied Spectroscopy, 39(3):491-500.

See Also

`step_measure_snv()` for a simpler scatter correction method

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_msc() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_mw_averages

Calculate Molecular Weight Averages for SEC/GPC

Description

[Superseded]

`step_measure_mw_averages()` creates a *specification* of a recipe step that calculates molecular weight averages from size exclusion chromatography data.

This step has been superseded by `measure.sec::step_sec_mw_averages()`. For new code, we recommend using the `measure.sec` package which provides more complete SEC/GPC analysis functionality.

Usage

```
step_measure_mw_averages(
  recipe,
  measures = NULL,
  calibration = NULL,
  integration_range = NULL,
  output_cols = c("mn", "mw", "mz", "mp", "dispersity"),
  prefix = "mw_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_mw_averages")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`calibration`	Calibration method for converting x-axis to log(MW). Can be: • `NULL` (default): Assumes x-axis is already log10(MW)

- A numeric vector of length 2: Linear calibration `c(slope, intercept)` where `log10(MW) = slope * x + intercept`
- `"auto"`: Estimate from data range (assumes typical polymer range)

`integration_range`
: Optional numeric vector `c(min, max)` specifying the x-axis range for integration. If `NULL`, uses full range.

`output_cols`
: Character vector of metrics to calculate. Default includes all: `c("mn", "mw", "mz", "mp", "dispersity")`.

`prefix`
: Prefix for output column names. Default is `"mw_"`.

`role`
: Role for generated columns. Default is `"predictor"`.

`trained`
: Logical indicating if the step has been trained.

`skip`
: Logical. Should the step be skipped when baking?

`id`
: Unique step identifier.

Details

This step calculates standard molecular weight averages from SEC/GPC data:

Metric	Formula	Description
Mn	$\Sigma w \ / \ \Sigma(w \ / M)$	Number-average molecular weight
Mw	$\Sigma(w\ M) \ / \ \Sigma w$	Weight-average molecular weight
Mz	$\Sigma(w\ M^2) \ / \ \Sigma(w\ M)$	Z-average molecular weight
Mp	M at peak maximum	Peak molecular weight
Đ	Mw/Mn	Dispersity (polydispersity index)

The detector signal is assumed to be proportional to weight concentration. For RI detection, this is typically valid. For UV detection, response factors may need to be applied first using `step_measure_calibrate_y()`.

Prerequisites:

- Data should be baseline corrected
- X-axis should represent retention time/volume or log(MW)
- Integration limits should exclude solvent peaks

Value

An updated recipe with the new step added.

See Also

Other measure-chromatography: `step_measure_mw_distribution()`, `step_measure_mw_fractions()`

Examples

```
library(recipes)

# Assuming x-axis is already calibrated to log10(MW)
# rec <- recipe(~., data = gpc_data) |>
#   step_measure_input_wide(starts_with("signal_")) |>
#   step_measure_baseline_als() |>
#   step_measure_mw_averages() |>
#   prep()
```

---

step_measure_mw_distribution

Generate Molecular Weight Distribution Curve

Description

[**Superseded**]

`step_measure_mw_distribution()` creates a *specification* of a recipe step that generates molecular weight distribution curves from SEC/GPC data.

This step has been superseded by `measure.sec::step_sec_mw_distribution()`. For new code, we recommend using the `measure.sec` package which provides more complete SEC/GPC analysis functionality.

Usage

```
step_measure_mw_distribution(
  recipe,
  measures = NULL,
  type = c("differential", "cumulative", "both"),
  calibration = NULL,
  n_points = 100L,
  mw_range = NULL,
  normalize = TRUE,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_mw_distribution")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`type`	Type of distribution to generate: • `"differential"` (default): dW/d(log M) differential distribution • `"cumulative"`: Cumulative weight fraction distribution

- `"both"`: Generate both distributions

`calibration` Calibration method for converting x-axis to log(MW). See `step_measure_mw_averages()` for details.

`n_points` Number of points in the output distribution. Default is 100. If `NULL`, uses the original data resolution.

`mw_range` Optional numeric vector `c(min, max)` specifying the MW range for the output distribution. If `NULL`, uses the range from data.

`normalize` Logical. Should the differential distribution be normalized to integrate to 1? Default is `TRUE`.

`role` Role for generated columns. Default is `"predictor"`.

`trained` Logical indicating if the step has been trained.

`skip` Logical. Should the step be skipped when baking?

`id` Unique step identifier.

Details

This step transforms the raw chromatogram into standard MW distribution representations:

Differential Distribution (dW/d(log M)): The weight fraction per unit log(MW). This representation is preferred because the area under the curve represents the weight fraction in that MW range.

Cumulative Distribution: The cumulative weight fraction from low to high MW. Values range from 0 to 1.

The output replaces the `.measures` column with the distribution data, where `location` contains log10(MW) values and `value` contains the distribution values.

Value

An updated recipe with the new step added.

See Also

Other measure-chromatography: `step_measure_mw_averages()`, `step_measure_mw_fractions()`

Examples

```
library(recipes)

# Generate differential MW distribution
# rec <- recipe(~., data = gpc_data) |>
#   step_measure_input_wide(starts_with("signal_")) |>
#   step_measure_baseline_als() |>
#   step_measure_mw_distribution(type = "differential") |>
#   prep()
```

`step_measure_mw_fractions` *Calculate Molecular Weight Fractions for SEC/GPC*

Description

[Superseded]

`step_measure_mw_fractions()` creates a *specification* of a recipe step that calculates weight fractions above and below specified molecular weight cutoffs.

This step has been superseded by `measure.sec::step_sec_mw_fractions()`. For new code, we recommend using the `measure.sec` package which provides more complete SEC/GPC analysis functionality.

Usage

```
step_measure_mw_fractions(
  recipe,
  measures = NULL,
  cutoffs = c(1000, 10000, 1e+05),
  calibration = NULL,
  integration_range = NULL,
  prefix = "frac_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_mw_fractions")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`cutoffs`	Numeric vector of MW cutoff values. For each cutoff, the step calculates the weight fraction below and above that value.
`calibration`	Calibration method for converting x-axis to log(MW). See `step_measure_mw_averages()` for details.
`integration_range`	Optional numeric vector `c(min, max)` specifying the x-axis range for integration. If `NULL`, uses full range.
`prefix`	Prefix for output column names. Default is `"frac_"`.
`role`	Role for generated columns. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

For each cutoff value `C`, this step calculates:

- `frac_below_C`: Weight fraction with MW < C
- `frac_above_C`: Weight fraction with MW >= C

These fractions sum to 1.0 and are useful for characterizing polymer distributions. Common cutoffs include:

- 1000 Da for oligomer content
- 10000 Da for low MW fraction
- 100000 Da for high MW fraction

Value

An updated recipe with the new step added.

See Also

Other measure-chromatography: `step_measure_mw_averages()`, `step_measure_mw_distribution()`

Examples

```
library(recipes)

# Calculate fractions at multiple cutoffs
# rec <- recipe(~., data = gpc_data) |>
#   step_measure_input_wide(starts_with("signal_")) |>
#   step_measure_baseline_als() |>
#   step_measure_mw_fractions(cutoffs = c(1000, 10000, 100000)) |>
#   prep()
```

---

step_measure_normalize_auc

Normalize by Area Under Curve

Description

`step_measure_normalize_auc()` creates a *specification* of a recipe step that divides each spectrum by its area under the curve (computed using trapezoidal integration). This is useful for chromatography where peak areas are meaningful.

Usage

```
step_measure_normalize_auc(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_auc")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

The area under the curve is computed using trapezoidal integration:

$$AUC = \sum_{i=1}^{n-1} \frac{(y_i + y_{i+1})}{2} \cdot (x_{i+1} - x_i)$$

where y are the values and x are the locations.

After transformation, the AUC of each spectrum will equal 1.

If the AUC is zero or NA, a warning is issued and the original values are returned unchanged. At least 2 points are required for integration.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

step_measure_normalize_sum(), step_measure_normalize_peak()

Other measure-normalization: step_measure_normalize_max(), step_measure_normalize_peak(), step_measure_normalize_range(), step_measure_normalize_sum(), step_measure_normalize_vector()

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_auc() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_normalize_istd

Internal Standard Normalization

Description

step_measure_normalize_istd() is an alias for step_measure_normalize_peak() with domain-specific naming for chromatography and mass spectrometry users. It normalizes spectra by dividing by a value computed from a specific region (internal standard peak).

Usage

```
step_measure_normalize_istd(
  recipe,
  location_min,
  location_max,
  method = "mean",
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_istd")
)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
location_min	Numeric. The lower bound of the region to use for normalization. This parameter is tunable with peak_location_min().

location_max	Numeric. The upper bound of the region to use for normalization. This parameter is tunable with peak_location_max().
method	Character. The summary statistic to compute from the region. One of "mean" (default), "max", or "integral".
measures	An optional character vector of measure column names to process. If NULL (the default), all measure columns (columns with class measure_list) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
skip	A logical. Should the step be skipped when the recipe is baked by recipes::bake()? While all operations are baked when recipes::prep() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.
id	A character string that is unique to this step to identify it.

Details

This function is identical to step_measure_normalize_peak() but uses terminology familiar to chromatography and mass spectrometry practitioners.

Internal standard (ISTD) normalization is commonly used to correct for:

- Injection volume variations
- Ionization efficiency differences
- Matrix effects
- Instrument drift

The internal standard should be a compound that:

- Is chemically stable
- Does not naturally occur in samples
- Elutes in a distinct region
- Has consistent response

Value

An updated version of recipe with the new step added.

See Also

step_measure_normalize_peak() for the underlying implementation

Other measure-preprocessing: step_measure_absorbance(), step_measure_calibrate_x(), step_measure_calibrate_y(), step_measure_derivative(), step_measure_derivative_gap(), step_measure_emsc(), step_measure_kubelka_munk(), step_measure_log(), step_measure_map(), step_measure_msc(), step_measure_osc(), step_measure_ratio_reference(), step_measure_snv(), step_measure_subtract_blank(), step_measure_subtract_reference(), step_measure_transmittance()

Examples

```
library(recipes)

# Normalize to internal standard peak region (channels 50-60)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_istd(
    location_min = 50,
    location_max = 60,
    method = "integral"
  )
```

---

step_measure_normalize_max

Normalize by Maximum Value

Description

`step_measure_normalize_max()` creates a *specification* of a recipe step that divides each spectrum by its maximum value. This is useful for peak-focused analysis where you want the highest peak to equal 1.

Usage

```
step_measure_normalize_max(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_max")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`skip` A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id` A character string that is unique to this step to identify it.

Details

For each spectrum x, the transformation is:

$$x_{norm} = \frac{x}{\max(x)}$$

After transformation, the maximum value of each spectrum will equal 1.

If the maximum is zero or NA, a warning is issued and the original values are returned unchanged.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_normalize_sum()`, `step_measure_normalize_range()`

Other measure-normalization: `step_measure_normalize_auc()`, `step_measure_normalize_peak()`, `step_measure_normalize_range()`, `step_measure_normalize_sum()`, `step_measure_normalize_vector()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_max() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_normalize_peak` *Normalize to a Specific Peak Region*

Description

`step_measure_normalize_peak()` creates a *specification* of a recipe step that divides each spectrum by a summary statistic computed from a specified region. This is commonly used for internal standard normalization.

Usage

```
step_measure_normalize_peak(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  location_min = NULL,
  location_max = NULL,
  method = "mean",
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_peak")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`location_min`	Numeric. The lower bound of the region to use for normalization. This parameter is tunable with `peak_location_min()`.
`location_max`	Numeric. The upper bound of the region to use for normalization. This parameter is tunable with `peak_location_max()`.
`method`	Character. The summary statistic to compute from the region. One of `"mean"` (default), `"max"`, or `"integral"`.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id` A character string that is unique to this step to identify it.

Details

For each spectrum, this step:

1. Selects values in the region `[location_min, location_max]`
2. Computes a summary statistic (mean, max, or integral) from that region
3. Divides the entire spectrum by this value

This is useful when you have an internal standard peak at a known location and want to normalize all spectra to that peak.

The `location_min` and `location_max` parameters are tunable with `peak_location_min()` and `peak_location_max()` for hyperparameter optimization.

If no values fall within the specified region, an error is raised. If the computed normalizer is zero or NA, a warning is issued and the original values are returned unchanged.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_normalize_max()`, `step_measure_normalize_auc()`, `peak_location_min()`, `peak_location_max()`

Other measure-normalization: `step_measure_normalize_auc()`, `step_measure_normalize_max()`, `step_measure_normalize_range()`, `step_measure_normalize_sum()`, `step_measure_normalize_vector()`

Examples

```
library(recipes)

# Normalize to mean of region 40-60
rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_peak(location_min = 40, location_max = 60) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_normalize_range` *Normalize to Range 0-1*

Description

`step_measure_normalize_range()` creates a *specification* of a recipe step that applies min-max normalization to scale each spectrum to the range 0 to 1.

Usage

```
step_measure_normalize_range(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_range")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

For each spectrum x, the transformation is:

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

After transformation, the minimum value of each spectrum will be 0 and the maximum will be 1.

If the range is zero (constant spectrum), a warning is issued and centered values are returned (minimum subtracted but no scaling).

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_normalize_max()`, `step_measure_snv()`

Other measure-normalization: `step_measure_normalize_auc()`, `step_measure_normalize_max()`, `step_measure_normalize_peak()`, `step_measure_normalize_sum()`, `step_measure_normalize_vector()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_range() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_normalize_sum

Normalize by Sum (Total Intensity)

Description

`step_measure_normalize_sum()` creates a *specification* of a recipe step that divides each spectrum by its sum (total intensity). This is useful for comparing relative abundances across samples with different total signals.

Usage

```
step_measure_normalize_sum(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_sum")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

For each spectrum x, the transformation is:

$$x_{norm} = \frac{x}{\sum x}$$

After transformation, the sum of each spectrum will equal 1.

If the sum is zero or NA, a warning is issued and the original values are returned unchanged.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_normalize_max()`, `step_measure_normalize_auc()`

Other measure-normalization: `step_measure_normalize_auc()`, `step_measure_normalize_max()`, `step_measure_normalize_peak()`, `step_measure_normalize_range()`, `step_measure_normalize_vector()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
```

```
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_sum() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_normalize_vector` *Normalize by L2 (Euclidean) Norm*

Description

`step_measure_normalize_vector()` creates a *specification* of a recipe step that divides each spectrum by its L2 (Euclidean) norm. After transformation, each spectrum will have unit length in Euclidean space.

Usage

```
step_measure_normalize_vector(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_normalize_vector")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

For each spectrum x, the transformation is:

$$x_{norm} = \frac{x}{\|x\|_2} = \frac{x}{\sqrt{\sum x^2}}$$

After transformation, the L2 norm of each spectrum will equal 1.

If the L2 norm is zero or NA, a warning is issued and the original values are returned unchanged.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_normalize_sum()`, `step_measure_snv()`

Other measure-normalization: `step_measure_normalize_auc()`, `step_measure_normalize_max()`, `step_measure_normalize_peak()`, `step_measure_normalize_range()`, `step_measure_normalize_sum()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_normalize_vector() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_osc *Orthogonal Signal Correction (OSC)*

Description

`step_measure_osc()` creates a *specification* of a recipe step that applies Orthogonal Signal Correction to remove variation orthogonal to the outcome.

Usage

```
step_measure_osc(
  recipe,
  n_components = 1L,
  tolerance = 1e-06,
  max_iter = 100L,
  measures = NULL,
  role = NA,
  trained = FALSE,
  weights = NULL,
  loadings = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_osc")
)
```

Arguments

`recipe`	A recipe object.
`n_components`	Number of orthogonal components to remove. Default is 1.
`tolerance`	Convergence tolerance for NIPALS algorithm. Default is 1e-6.
`max_iter`	Maximum iterations for NIPALS. Default is 100.
`measures`	An optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`weights`	The learned orthogonal weights (after training).
`loadings`	The learned orthogonal loadings (after training).
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Orthogonal Signal Correction (OSC) removes variation in X that is orthogonal to Y (the outcome). This is useful for removing systematic variation that is not related to the response.

Algorithm:

1. Compute initial score t from Y using SVD
2. Orthogonalize t with respect to Y
3. Iterate NIPALS to find orthogonal components
4. Remove orthogonal components from X

Important:

- The recipe must have at least one outcome variable with role "outcome"
- Outcomes are automatically detected from the recipe's role definitions
- Multiple outcomes are supported (multivariate Y)

OSC was originally described by Wold et al. (1998) for NIR spectroscopy.

Value

An updated recipe with the new step added.

References

Wold, S., Antti, H., Lindgren, F., and Ohman, J. (1998). Orthogonal signal correction of near-infrared spectra. Chemometrics and Intelligent Laboratory Systems, 44(1-2), 175-185.

See Also

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_ratio_reference()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_osc(n_components = 2) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_output_long

Reorganize Measurements to Long Format

Description

`step_measure_output_long` creates a *specification* of a recipe

Usage

```
step_measure_output_long(
  recipe,
  values_to = ".measure",
  location_to = ".location",
  measures = NULL,
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = rand_id("measure_output_long")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`values_to`	A single character string for the column containing the analytical measurement.
`location_to`	A single character string for the column name prefix for location columns. For 1D data, this becomes the column name (default: `.location`). For nD data, this becomes a prefix with dimension suffixes (e.g., `.location_1`, `.location_2`).
`measures`	An optional single character string specifying which measure column to output. If `NULL` (the default) and only one measure column exists, that column will be used. If multiple measure columns exist and `measures` is `NULL`, an error will be thrown prompting you to specify which column to output.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

step that converts measures to a format with columns for the measurement and the corresponding location (i.e., "long" format).

This step is designed convert analytical measurements from their internal data structure to a long format with explicit location columns.

For 1D data, the output has two columns: the measurement value and a single location column.

For n-dimensional data (2D, 3D, etc.), the output has n+1 columns: the measurement value and n location columns named with the `location_to` prefix followed by dimension numbers (e.g., `.location_1`, `.location_2`).

See Also

Other input/output steps: `step_measure_input_long()`, `step_measure_input_wide()`, `step_measure_output_wide()`

Examples

```
library(dplyr)

data(glucose_bioreactors)
```

```
bioreactors_small$batch_sample <- NULL

small_tr <- bioreactors_small[1:200, ]
small_te <- bioreactors_small[201:210, ]

small_rec <-
  recipe(glucose ~ ., data = small_tr) |>
  update_role(batch_id, day, new_role = "id columns") |>
  step_measure_input_wide(`400`:`3050`) |>
  prep()

# Before reformatting:

small_rec |> bake(new_data = small_te)

# After reformatting:

output_rec <-
  small_rec |>
  step_measure_output_long() |>
  prep()

output_rec |> bake(new_data = small_te)
```

`step_measure_output_wide` *Reorganize Measurements to Separate Columns*

Description

`step_measure_output_wide` creates a *specification* of a recipe step that converts measures to multiple columns (i.e., "wide" format).

Usage

```
step_measure_output_wide(
  recipe,
  prefix = "measure_",
  measures = NULL,
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = rand_id("measure_output_wide")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.

`prefix`	A character string used to name the new columns.
`measures`	An optional single character string specifying which measure column to output. If `NULL` (the default) and only one measure column exists, that column will be used. If multiple measure columns exist and `measures` is `NULL`, an error will be thrown prompting you to specify which column to output.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake()`? While all operations are baked when `prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

This step is designed convert analytical measurements from their internal data structure to separate columns.

Wide outputs can be helpful when you want to use standard recipes steps with the measuresments, such as `recipes::step_pca()`, `recipes::step_pls()`, and so on.

See Also

Other input/output steps: `step_measure_input_long()`, `step_measure_input_wide()`, `step_measure_output_long()`

Examples

```
library(dplyr)

data(glucose_bioreactors)
bioreactors_small$batch_sample <- NULL

small_tr <- bioreactors_small[1:200, ]
small_te <- bioreactors_small[201:210, ]

small_rec <-
  recipe(glucose ~ ., data = small_tr) |>
  update_role(batch_id, day, new_role = "id columns") |>
  step_measure_input_wide(`400`:`3050`) |>
  prep()

# Before reformatting:

small_rec |> bake(new_data = small_te)

# After reformatting:
```

```
output_rec <-
  small_rec |>
  step_measure_output_wide() |>
  prep()

output_rec |> bake(new_data = small_te)
```

`step_measure_parafac` *PARAFAC Decomposition for Multi-Dimensional Data*

Description

`step_measure_parafac()` creates a *specification* of a recipe step that applies Parallel Factor Analysis (PARAFAC) to multi-dimensional measurement data, extracting component scores as features for modeling.

Usage

```
step_measure_parafac(
  recipe,
  ...,
  n_components = 3L,
  center = TRUE,
  scale = FALSE,
  max_iter = 500L,
  tol = 1e-06,
  prefix = "parafac_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_parafac")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose measure columns. If empty, all nD measure columns are used.
`n_components`	Number of PARAFAC components to extract. Default is 3.
`center`	Logical. Should data be centered before decomposition? Default is `TRUE`.
`scale`	Logical. Should data be scaled before decomposition? Default is `FALSE`.
`max_iter`	Maximum number of iterations. Default is 500.
`tol`	Convergence tolerance. Default is 1e-6.
`prefix`	Prefix for output column names. Default is `"parafac_"`.

`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

PARAFAC (also known as CANDECOMP/PARAFAC or CP decomposition) decomposes a three-way or higher array into a sum of rank-one tensors. For measurement data like EEM (excitation-emission matrices) or LC-DAD, this extracts interpretable components corresponding to underlying chemical species.

Requirements:

- Input must be `measure_nd_list` with 2+ dimensions
- All samples must have the same grid (regular, aligned)
- The `multiway` package must be installed (in Suggests)

Output:

Creates numeric feature columns: `parafac_1`, `parafac_2`, ..., `parafac_n` representing each sample's scores on the extracted components.

Value

An updated recipe with the new step added.

Note

This step requires the `multiway` package. Install with: `install.packages("multiway")`

See Also

`step_measure_tucker()` for Tucker decomposition

Other measure-multiway: `step_measure_mcr_als()`, `step_measure_tucker()`

Examples

```
## Not run:
library(recipes)

# After ingesting EEM data as 2D measurements
rec <- recipe(concentration ~ ., data = eem_data) |>
  step_measure_input_long(
    fluorescence,
    location = vars(excitation, emission)
  ) |>
  step_measure_parafac(n_components = 3) |>
  prep()

bake(rec, new_data = NULL)

## End(Not run)
```

`step_measure_peaks_deconvolve` *Deconvolve Overlapping Peaks*

Description

`step_measure_peaks_deconvolve()` creates a *specification* of a recipe step that resolves overlapping peaks using curve fitting. This step requires peaks to have been detected first using `step_measure_peaks_detect()`.

Usage

```
step_measure_peaks_deconvolve(
  recipe,
  model = "gaussian",
  optimizer = "auto",
  max_iter = 500L,
  tol = 1e-06,
  n_starts = 5L,
  constrain_positions = TRUE,
  quality_threshold = 0.8,
  store_components = FALSE,
  smart_init = TRUE,
  peaks_col = ".peaks",
  measures_col = ".measures",
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_deconvolve")
)
```

Arguments

`recipe`	A recipe object.
`model`	Peak model to use. Either a character string naming a registered model (`"gaussian"`, `"emg"`, `"bigaussian"`, `"lorentzian"`), or a `peak_model` object created directly. Use `peak_models()` to see all registered models. Default is `"gaussian"`.
`optimizer`	Optimization method: `"auto"` (default), `"lbfgsb"`, `"multistart"`, or `"nelder_mead"`. Auto-selection chooses based on problem complexity and signal-to-noise ratio.
`max_iter`	Maximum iterations for optimization. Default is 500.
`tol`	Convergence tolerance. Default is 1e-6.
`n_starts`	Number of random starts for `optimizer = "multistart"`. Default is 5.

`constrain_positions`
: Logical. If `TRUE`, enforce that peak centers maintain their relative ordering. Default is `TRUE`.

`quality_threshold`
: Minimum R-squared to accept fit. Fits below this threshold trigger a warning. Default is 0.8.

`store_components`
: Logical. If `TRUE`, store individual fitted peak curves in the output. Default is `FALSE`.

`smart_init`
: Logical. If `TRUE`, use smart initialization based on peak properties. Default is `TRUE`.

`peaks_col`
: Name of the peaks column. Default is ".peaks".

`measures_col`
: Name of the measures column. Default is ".measures".

`role`
: Not used.

`trained`
: Logical indicating if the step has been trained.

`skip`
: Logical. Should the step be skipped when baking?

`id`
: Unique step identifier.

Details

Peak deconvolution fits mathematical models to overlapping peaks to determine their individual contributions. This is essential for quantitative analysis when peaks are not baseline-resolved.

Peak Models:

Built-in models (use `peak_models()` to see all):

- `"gaussian"`: Symmetric Gaussian (3 params: height, center, width)
- `"emg"`: Exponentially Modified Gaussian (4 params, handles tailing)
- `"bigaussian"`: Bi-Gaussian (4 params, flexible asymmetry)
- `"lorentzian"`: Lorentzian/Cauchy peak (3 params, heavier tails)

Technique packs may register additional models.

Optimizers:

- `"auto"`: Selects based on problem complexity and SNR
- `"lbfgsb"`: L-BFGS-B (fast, local optimization)
- `"multistart"`: Multiple L-BFGS-B runs from perturbed starts (robust)
- `"nelder_mead"`: Derivative-free Nelder-Mead simplex

Quality Assessment:

Each fit is assessed for quality. The `.peaks` tibble gains columns:

- `fit_r_squared`: R-squared of the overall fit
- `fit_quality`: Quality grade (A/B/C/D/F)
- `purity`: How much of signal at peak max comes from this peak

Value

An updated recipe with the new step added. The `.peaks` column will be updated with deconvolved peak parameters, fitted areas, and quality metrics.

See Also

`optimize_deconvolution()`, `assess_deconv_quality()`, `peak_models()`, `gaussian_peak_model()`

Other peak-operations: `step_measure_peaks_detect()`, `step_measure_peaks_filter()`, `step_measure_peaks_integrate()`, `step_measure_peaks_properties()`, `step_measure_peaks_to_table()`

Examples

```
library(recipes)

# Create synthetic data with overlapping peaks
set.seed(42)
x <- seq(0, 20, by = 0.1)
y <- 1.5 * exp(-0.5 * ((x - 8) / 1)^2) +
  0.8 * exp(-0.5 * ((x - 12) / 1.5)^2) +
  rnorm(length(x), sd = 0.02)
df <- data.frame(id = "sample1", location = x, value = y)


# Deconvolve overlapping peaks
rec <- recipe(~., data = df) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(value, location = vars(location)) |>
  step_measure_peaks_detect(min_height = 0.5, min_prominence = 0.3) |>
  step_measure_peaks_deconvolve(model = "gaussian") |>
  prep()

result <- bake(rec, new_data = NULL)
# Check fitted peaks
result$.peaks[[1]]
```

`step_measure_peaks_detect` *Detect Peaks in Measurements*

Description

`step_measure_peaks_detect()` creates a *specification* of a recipe step that detects peaks in measurement data and stores them in a new `.peaks` column.

Usage

```
step_measure_peaks_detect(
  recipe,
  algorithm = "prominence",
  min_height = 0,
  min_distance = 0,
  min_prominence = 0,
  snr_threshold = FALSE,
  algorithm_params = list(),
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_detect")
)
```

Arguments

`recipe`	A recipe object.
`algorithm`	Peak detection algorithm. One of `"prominence"` (default), `"derivative"`, `"local_maxima"`, or any algorithm registered via `register_peak_algorithm()`. Use `peak_algorithms()` to see available algorithms.
`min_height`	Minimum peak height. If `snr_threshold = TRUE`, this is interpreted as a signal-to-noise ratio threshold.
`min_distance`	Minimum distance between peaks in x-axis units.
`min_prominence`	Minimum peak prominence (only for `algorithm = "prominence"`).
`snr_threshold`	Logical. If `TRUE`, `min_height` is interpreted as a signal-to-noise ratio. Noise is estimated as the MAD of the signal.
`algorithm_params`	Named list of additional algorithm-specific parameters. These are passed to the algorithm function along with the standard parameters.
`measures`	Optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step detects peaks in measurement data and creates a new `.peaks` column containing the detected peaks for each sample. The original `.measures` column is preserved.

Detection algorithms:

- `"prominence"` (default): Finds local maxima and calculates their prominence (how much a peak stands out from surrounding signal). More robust to noise.
- `"derivative"`: Finds peaks by detecting zero-crossings in the first derivative. Faster but more sensitive to noise.
- `"local_maxima"`: Finds all local maxima above a threshold. Simple and fast but may detect many spurious peaks.

Additional algorithms can be registered by technique packs using `register_peak_algorithm()`.

Peak properties stored:

- `peak_id`: Integer identifier
- `location`: X-axis position of peak apex
- `height`: Y-value at peak apex
- `left_base`, `right_base`: X-axis positions of peak boundaries
- `area`: Initially NA; use `step_measure_peaks_integrate()` to calculate

Use `step_measure_peaks_properties()` to calculate additional peak metrics such as prominence and full width at half maximum (FWHM).

Value

An updated recipe with the new step added.

See Also

`peak_algorithms()`, `register_peak_algorithm()`

Other peak-operations: `step_measure_peaks_deconvolve()`, `step_measure_peaks_filter()`, `step_measure_peaks_integrate()`, `step_measure_peaks_properties()`, `step_measure_peaks_to_table()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(min_height = 0.5, min_distance = 5) |>
  prep()

result <- bake(rec, new_data = NULL)
# Result now has .peaks column alongside .measures

# Use a different algorithm
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(algorithm = "derivative", min_height = 0.5) |>
  prep()
```

`step_measure_peaks_filter` *Filter Peaks by Criteria*

Description

`step_measure_peaks_filter()` creates a *specification* of a recipe step that filters detected peaks based on various criteria.

Usage

```
step_measure_peaks_filter(
  recipe,
  min_height = NULL,
  min_area = NULL,
  min_area_pct = NULL,
  min_prominence = NULL,
  max_peaks = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_filter")
)
```

Arguments

`recipe`	A recipe object.
`min_height`	Minimum peak height. Peaks below this are removed.
`min_area`	Minimum peak area. Requires prior integration.
`min_area_pct`	Minimum area as percentage of total. Peaks with area less than this percentage of total peak area are removed.
`min_prominence`	Minimum peak prominence. Requires a `prominence` column, typically added by `step_measure_peaks_properties()`.
`max_peaks`	Maximum number of peaks to keep (keeps largest by area or height).
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step removes peaks that don't meet specified criteria. Multiple criteria can be combined - peaks must pass ALL specified filters.

Value

An updated recipe with the new step added.

See Also

Other peak-operations: `step_measure_peaks_deconvolve()`, `step_measure_peaks_detect()`, `step_measure_peaks_integrate()`, `step_measure_peaks_properties()`, `step_measure_peaks_to_table()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(min_height = 0.3) |>
  step_measure_peaks_integrate() |>
  step_measure_peaks_filter(min_area_pct = 1) |>
  prep()

result <- bake(rec, new_data = NULL)
```

---

step_measure_peaks_integrate

Integrate Peak Areas

Description

`step_measure_peaks_integrate()` creates a *specification* of a recipe step that calculates the area under each detected peak.

Usage

```
step_measure_peaks_integrate(
  recipe,
  method = c("trapezoid", "simpson"),
  baseline = c("local", "none", "global"),
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_integrate")
)
```

Arguments

`recipe`	A recipe object.
`method`	Integration method. One of `"trapezoid"` (default) or `"simpson"`.
`baseline`	Baseline handling. One of `"local"` (linear interpolation between peak bases), `"none"` (integrate to zero), or `"global"` (use minimum value as baseline).
`measures`	Optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step calculates the area under each peak detected by `step_measure_peaks_detect()`. The areas are stored in the `area` column of the `.peaks` tibble.

Integration methods:

- `"trapezoid"`: Trapezoidal rule integration. Fast and accurate for well-resolved peaks.
- `"simpson"`: Simpson's rule integration. More accurate for smooth curves but requires odd number of points.

Baseline handling:

- `"local"`: Subtracts a linear baseline connecting the left and right peak bases before integration.
- `"none"`: Integrates directly to y=0.
- `"global"`: Subtracts the minimum value in the peak region.

Value

An updated recipe with the new step added.

See Also

Other peak-operations: `step_measure_peaks_deconvolve()`, `step_measure_peaks_detect()`, `step_measure_peaks_filter()`, `step_measure_peaks_properties()`, `step_measure_peaks_to_table()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(min_height = 0.5) |>
  step_measure_peaks_integrate() |>
```

```
  prep()

result <- bake(rec, new_data = NULL)
```

`step_measure_peaks_properties` *Calculate Peak Properties*

Description

`step_measure_peaks_properties()` creates a *specification* of a recipe step that calculates derived peak metrics from the measured signal and stores them in the `.peaks` tibble.

Usage

```
step_measure_peaks_properties(
  recipe,
  properties = c("prominence", "fwhm"),
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_properties")
)
```

Arguments

`recipe`	A recipe object.
`properties`	Character vector of peak properties to calculate. Supported values are `"prominence"` and `"fwhm"`.
`measures`	Optional character vector of measure column names.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step calculates additional peak metrics from the observed signal for each detected peak:

- `"prominence"`: Peak height above the higher of the left and right base intensities.
- `"fwhm"`: Full width at half maximum, estimated with linear interpolation after subtracting a local linear baseline between the left and right bases.

The calculated properties are added as new columns in the `.peaks` tibble and can be exported later with `step_measure_peaks_to_table()`.

Value

An updated recipe with the new step added.

See Also

Other peak-operations: `step_measure_peaks_deconvolve()`, `step_measure_peaks_detect()`, `step_measure_peaks_filter()`, `step_measure_peaks_integrate()`, `step_measure_peaks_to_table()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(min_height = 0.5) |>
  step_measure_peaks_properties(c("prominence", "fwhm")) |>
  prep()

result <- bake(rec, new_data = NULL)
result$.peaks[[1]]
```

---

step_measure_peaks_to_table

Convert Peaks to Tidy Table

Description

`step_measure_peaks_to_table()` creates a *specification* of a recipe step that converts the peaks list-column to a wide format with one column per peak property.

Usage

```
step_measure_peaks_to_table(
  recipe,
  prefix = "peak_",
  properties = c("location", "height", "area"),
  max_peaks = 10,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_peaks_to_table")
)
```

Arguments

recipe	A recipe object.
prefix	Prefix for generated column names. Default is "peak_".
properties	Which peak properties to include. Default includes location, height, and area for each peak.
max_peaks	Maximum number of peaks to include in output. If a sample has more peaks, only the first max_peaks are included.
role	Not used.
trained	Logical indicating if the step has been trained.
skip	Logical. Should the step be skipped when baking?
id	Unique step identifier.

Details

This step converts peak data to a wide format suitable for modeling. For each peak, it creates columns like peak_1_location, peak_1_height, peak_1_area, etc.

The .peaks and .measures columns are removed after conversion.

Value

An updated recipe with the new step added.

See Also

Other peak-operations: step_measure_peaks_deconvolve(), step_measure_peaks_detect(), step_measure_peaks_filter(), step_measure_peaks_integrate(), step_measure_peaks_properties()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_peaks_detect(min_height = 0.5) |>
  step_measure_peaks_integrate() |>
  step_measure_peaks_to_table(max_peaks = 5) |>
  prep()

result <- bake(rec, new_data = NULL)
```

`step_measure_qc_bracket` *QC Bracketing Interpolation*

Description

`step_measure_qc_bracket()` creates a *specification* of a recipe step that corrects for drift using linear interpolation between bracketing QC or reference samples. This is a simple, intuitive method where each sample is corrected based on the two nearest QC samples.

Usage

```
step_measure_qc_bracket(
  recipe,
  ...,
  run_order_col = "run_order",
  sample_type_col = "sample_type",
  qc_type = "qc",
  apply_to = c("all", "unknown"),
  extrapolate = TRUE,
  min_qc = 2,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_qc_bracket")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose feature columns. For feature-level data, select the numeric response columns. For curve-level data with `.measures`, leave empty to apply to all locations.
`run_order_col`	Name of the column containing run order (injection sequence). Must be numeric/integer.
`sample_type_col`	Name of the column containing sample type.
`qc_type`	Value(s) in `sample_type_col` that identify QC samples to use for drift modeling. Default is `"qc"`.
`apply_to`	Which samples to apply correction to: • `"all"` (default): Correct all samples • `"unknown"`: Only correct unknown samples
`extrapolate`	Logical. Should correction be extrapolated for samples before the first or after the last QC? Default is TRUE. If FALSE, those samples use the nearest QC's correction factor.

`min_qc`	Minimum number of QC samples required. Default is 5.
`role`	Not used by this step.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

How It Works:

For each sample at run order `t`:

1. Find the nearest QC samples before (`t1`) and after (`t2`)
2. Calculate correction factors at `t1` and `t2` (target / observed)
3. Linearly interpolate the correction factor for `t`
4. Apply the interpolated correction

This method is commonly used in clinical and bioanalytical laboratories where QC samples are injected at regular intervals throughout the run.

When to Use:

- Regular QC injection intervals
- Short analytical runs
- When you want simple, transparent corrections
- Regulatory environments where interpretability is important

Value

An updated recipe with the new step added.

See Also

Other drift-correction: `step_measure_drift_linear()`, `step_measure_drift_qc_loess()`, `step_measure_drift_spline()`

Examples

```
library(recipes)

# Data with QC samples at regular intervals
data <- data.frame(
  sample_id = paste0("S", 1:15),
  sample_type = c("qc", rep("unknown", 4), "qc", rep("unknown", 4), "qc",
                  rep("unknown", 3), "qc"),
  run_order = 1:15,
  feature1 = c(100, 101, 103, 105, 107, 105, 107, 109, 111, 113,
               110, 112, 114, 116, 115)  # Drift pattern
)

rec <- recipe(~ ., data = data) |>
  update_role(sample_id, new_role = "id") |>
```

```
  step_measure_qc_bracket(feature1) |>
  prep()

corrected <- bake(rec, new_data = NULL)
```

`step_measure_qc_outlier` *Detect Outlier Samples*

Description

`step_measure_qc_outlier()` creates a *specification* of a recipe step that detects outlier samples using Mahalanobis distance or PCA-based methods. A new column is added indicating outlier status.

Usage

```
step_measure_qc_outlier(
  recipe,
  measures = NULL,
  method = c("mahalanobis", "pca"),
  threshold = 3,
  n_components = NULL,
  new_col = ".outlier",
  new_col_score = ".outlier_score",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_qc_outlier")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`method`	Detection method: • `"mahalanobis"` (default): Mahalanobis distance with robust covariance • `"pca"`: PCA score-based outliers (Hotelling's T^2)
`threshold`	Threshold for outlier detection in standard deviation units. Default is 3. Tunable via `outlier_threshold()`.
`n_components`	For PCA method, number of components to use. Default is `NULL` (auto-select based on variance explained).
`new_col`	Name of the new outlier flag column. Default is `".outlier"`.
`new_col_score`	Name of the outlier score column. Default is `".outlier_score"`.

`role`	Role for new columns. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Outlier samples can arise from measurement errors, sample preparation issues, or genuine unusual samples. This step helps identify them.

Mahalanobis method: Computes the multivariate distance from each sample to the center of the distribution, accounting for correlations. Uses robust estimation of center and covariance via median and MAD.

PCA method: Projects data onto principal components and computes Hotelling's T^2 statistic. Samples with extreme scores are flagged.

Two columns are added:

- `.outlier`: Logical flag
- `.outlier_score`: Numeric score (higher = more extreme)

Value

An updated recipe with the new step added.

See Also

Other measure-qc: `step_measure_impute()`, `step_measure_qc_saturated()`, `step_measure_qc_snr()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_qc_outlier(threshold = 3) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_qc_saturated` *Detect Saturated Measurements*

Description

`step_measure_qc_saturated()` creates a *specification* of a recipe step that detects saturated (clipped) regions in measurements and adds metadata columns indicating saturation status.

Usage

```
step_measure_qc_saturated(
  recipe,
  measures = NULL,
  upper_limit = NULL,
  lower_limit = NULL,
  tolerance = 0.01,
  new_col_flag = ".saturated",
  new_col_pct = ".sat_pct",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_qc_saturated")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`upper_limit`	Upper saturation threshold. Default is `NULL` (auto-detect).
`lower_limit`	Lower saturation threshold. Default is `NULL` (auto-detect).
`tolerance`	How close to the limit counts as saturated. Default is 0.01.
`new_col_flag`	Name of column for saturation flag. Default is `".saturated"`.
`new_col_pct`	Name of column for saturation percentage. Default is `".sat_pct"`.
`role`	Role for new columns. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Saturation occurs when detector response reaches its maximum (or minimum) capacity. Saturated data points lose quantitative information and may need special handling.

If limits are not specified, they are auto-detected as values appearing as flat regions at extreme values (using `min()` and `max()`).

Two new columns are added:

- `.saturated`: Logical, TRUE if any saturation detected
- `.sat_pct`: Percentage of points that are saturated

Value

An updated recipe with the new step added.

See Also

Other measure-qc: `step_measure_impute()`, `step_measure_qc_outlier()`, `step_measure_qc_snr()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_qc_saturated() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_qc_snr *Calculate Signal-to-Noise Ratio*

Description

`step_measure_qc_snr()` creates a *specification* of a recipe step that calculates the signal-to-noise ratio (SNR) for each measurement and adds it as a new column. This is useful for quality control and filtering.

Usage

```
step_measure_qc_snr(
  recipe,
  measures = NULL,
  new_col = ".snr",
  signal_method = c("max", "range", "rms"),
  noise_method = c("diff", "mad", "residual"),
  role = "predictor",
```

```
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_qc_snr")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`new_col`	Name of the new column to store SNR values. Default is `".snr"`.
`signal_method`	How to estimate signal: • `"max"` (default): Maximum absolute value • `"range"`: Peak-to-peak range • `"rms"`: Root mean square
`noise_method`	How to estimate noise: • `"diff"` (default): RMS of first differences (estimates high-freq noise) • `"mad"`: Median absolute deviation of values • `"residual"`: Residuals from smoothed fit
`role`	Role for the new column. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

SNR is calculated as `signal / noise`, where signal and noise are estimated using the specified methods. Higher values indicate cleaner data.

The `"diff"` noise method is particularly useful because it estimates high-frequency noise without being affected by broad spectral features:

$$noise = \sqrt{\frac{1}{2(n-1)} \sum_{i=2}^{n} (x_i - x_{i-1})^2}$$

Value

An updated recipe with the new step added.

See Also

Other measure-qc: `step_measure_impute()`, `step_measure_qc_outlier()`, `step_measure_qc_saturated()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_qc_snr() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_ratio_reference

Compute Ratio to Reference Spectrum

Description

`step_measure_ratio_reference()` creates a *specification* of a recipe step that computes the ratio of each spectrum to a reference, optionally with blank subtraction.

Usage

```
step_measure_ratio_reference(
  recipe,
  reference,
  blank = NULL,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_ref = NULL,
  learned_blank = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_ratio_reference")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`reference`	A required external reference spectrum. Can be: • A `measure_tbl` object with `location` and `value` columns • A numeric vector (must match the number of locations in data) • A data.frame with `location` and `value` columns (will be interpolated)
`blank`	An optional blank spectrum to subtract from both sample and reference before computing the ratio. Same format options as `reference`.

`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_ref`	A named list containing the validated reference values for each measure column. This is `NULL` until the step is trained.
`learned_blank`	A named list containing the learned blank values for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

This step computes a ratio relative to a reference spectrum:

- Without blank: `result = sample / reference`
- With blank: `result = (sample - blank) / (reference - blank)`

This is useful for computing relative measurements, such as absorbance from transmittance when you have both sample and reference scans.

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_subtract_blank()` for simple blank subtraction

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Create reference and blank spectra
ref_spectrum <- rep(1.0, 100)
blank_spectrum <- rep(0.05, 100)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_ratio_reference(
    reference = ref_spectrum,
```

```
  blank = blank_spectrum
)
```

`step_measure_ratios` *Calculate Region Ratios*

Description

`step_measure_ratios()` creates a *specification* of a recipe step that calculates ratios between integrated regions.

Usage

```
step_measure_ratios(
  recipe,
  numerator,
  denominator,
  name = NULL,
  method = c("trapezoid", "simpson"),
  measures = NULL,
  prefix = "ratio_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_ratios")
)
```

Arguments

`recipe`	A recipe object.
`numerator`	A numeric vector of length 2 specifying the numerator region.
`denominator`	A numeric vector of length 2 specifying the denominator region.
`name`	Output column name. If `NULL`, auto-generated from prefix.
`method`	Integration method: `"trapezoid"` (default) or `"simpson"`.
`measures`	An optional character vector of measure column names.
`prefix`	Prefix for output column name if `name` is NULL. Default is `"ratio_"`.
`role`	Role for generated column. Default is `"predictor"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

This step calculates the ratio of integrated areas between two regions: `ratio = integral(numerator) / integral(denominator)`

This is useful for calculating peak ratios in spectroscopy, or relative concentrations in chromatography.

If the denominator integral is zero or NA, the ratio will be NA.

Value

An updated recipe with the new step added.

See Also

Other measure-features: `step_measure_bin()`, `step_measure_integrals()`, `step_measure_moments()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_ratios(
    numerator = c(1, 30),
    denominator = c(70, 100),
    name = "low_high_ratio"
  ) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_resample

Resample Measurements to New Grid

Description

`step_measure_resample()` creates a *specification* of a recipe step that interpolates measurements to a new regular x-axis grid.

Usage

```
step_measure_resample(
  recipe,
  n = NULL,
  spacing = NULL,
  range = NULL,
  method = c("linear", "spline"),
```

```
  measures = NULL,
  role = NA,
  trained = FALSE,
  new_locations = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_resample")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`n`	A positive integer specifying the number of points in the new grid. Mutually exclusive with `spacing`.
`spacing`	A positive numeric value specifying the spacing between points in the new grid. Mutually exclusive with `n`.
`range`	Optional numeric vector of length 2 specifying the range for the new grid as `c(min, max)`. If `NULL` (default), uses the range of the existing measurements.
`method`	The interpolation method. One of: • `"linear"`: Linear interpolation (default) • `"spline"`: Cubic spline interpolation (smoother)
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`new_locations`	The computed new grid locations (after training).
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

This step interpolates measurements to a new regular grid of x-axis values. This is useful for:

- Aligning data from different instruments with different sampling rates
- Reducing data density for faster processing
- Ensuring uniform spacing for methods that require it
- Matching measurements to a reference grid

The new grid is determined during `prep()` based on the training data. If `range` is not specified, the grid spans from the minimum to maximum location values in the training data.

Interpolation methods:

- `"linear"`: Fast and simple, may introduce slight distortion at peaks
- `"spline"`: Smoother interpolation that preserves peak shape better

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_trim()` for keeping specific ranges, `step_measure_exclude()` for removing specific ranges

Other region-operations: `step_measure_exclude()`, `step_measure_trim()`

Examples

```
library(recipes)

# Resample to 50 evenly spaced points
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_resample(n = 50) |>
  prep()

bake(rec, new_data = NULL)

# Resample with specific spacing
rec2 <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_resample(spacing = 2, method = "spline") |>
  prep()

bake(rec2, new_data = NULL)
```

---

step_measure_savitzky_golay

Savitzky-Golay Pre-Processing

Description

`step_measure_savitzky_golay` creates a *specification* of a recipe step that smooths and filters the measurement sequence.

Usage

```
step_measure_savitzky_golay(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  degree = 3,
```

```
  window_side = 11,
  differentiation_order = 0,
  skip = FALSE,
  id = rand_id("measure_savitzky_golay")
)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
measures	An optional character vector of measure column names to process. If NULL (the default), all measure columns (columns with class measure_list) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
degree	An integer for the polynomial degree to use for smoothing.
window_side	An integer for how many units there are on each side of the window. This means that window_side = 1 has a total window width of 3 (e.g., width is 2 * window_side + 1).
differentiation_order	An integer for the degree of filtering (zero indicates no differentiation).
skip	A logical. Should the step be skipped when the recipe is baked by recipes::bake()? While all operations are baked when recipes::prep() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations
id	A character string that is unique to this step to identify it.

Details

This method can both smooth out random noise and reduce between-predictor correlation. It fits a polynomial to a window of measurements and this results in fewer measurements than the input. Measurements are assumed to be equally spaced.

The polynomial degree should be less than the window size. Also, window size must be greater than polynomial degree. If either case is true, the original argument values are increased to satisfy these conditions (with a warning).

No selectors should be supplied to this step function. The data should be in a special internal format produced by step_measure_input_wide() or step_measure_input_long().

The measurement locations are reset to integer indices starting at one.

Value

An updated version of recipe with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns is returned.

See Also

Other measure-smoothing: `step_measure_despike()`, `step_measure_filter_fourier()`, `step_measure_smooth_gaussian()`, `step_measure_smooth_ma()`, `step_measure_smooth_median()`, `step_measure_smooth_wavelet()`

Examples

```
if (rlang::is_installed("prospectr")) {
  rec <-
    recipe(water + fat + protein ~ ., data = meats_long) |>
    update_role(id, new_role = "id") |>
    step_measure_input_long(transmittance, location = vars(channel)) |>
    step_measure_savitzky_golay(
      differentiation_order = 1,
      degree = 3,
      window_side = 5
    ) |>
    prep()
}
```

---

step_measure_scale_auto

Auto-Scaling (Z-Score Normalization)

Description

`step_measure_scale_auto()` creates a *specification* of a recipe step that applies auto-scaling (also known as z-score normalization or standardization) at each measurement location. This centers and scales to unit variance.

Usage

```
step_measure_scale_auto(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_params = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_scale_auto")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_params`	A named list containing learned means and locations for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Auto-scaling (standardization) transforms each variable to have zero mean and unit variance. This gives equal importance to all measurement locations regardless of their original scale.

For a data matrix X, the transformation is:

$$X_{scaled} = \frac{X - \bar{X}}{s_X}$$

where $\bar{X}$ and s_X are the column-wise mean and standard deviation computed from the training data.

If a column has zero standard deviation (constant values), that column is only centered, not scaled (the divisor is set to 1).

The means and standard deviations are learned during `prep()` from the training data and stored for use when applying the transformation to new data during `bake()`.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

step_measure_center(), step_measure_scale_pareto()

Other measure-scaling: step_measure_center(), step_measure_scale_pareto(), step_measure_scale_range(
step_measure_scale_vast()

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_scale_auto() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_scale_pareto

Pareto Scaling

Description

`step_measure_scale_pareto()` creates a *specification* of a recipe step that applies Pareto scaling at each measurement location. This is a compromise between no scaling and auto-scaling, commonly used in metabolomics.

Usage

```
step_measure_scale_pareto(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_params = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_scale_pareto")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.

`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_params`	A named list containing learned means and locations for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Pareto scaling divides by the square root of the standard deviation rather than the standard deviation itself. This reduces the relative importance of large values while still giving more weight to larger fold changes.

For a data matrix X, the transformation is:

$$X_{scaled} = \frac{X - \bar{X}}{\sqrt{s_X}}$$

where $\bar{X}$ and s_X are the column-wise mean and standard deviation computed from the training data.

If a column has zero standard deviation (constant values), that column is only centered, not scaled.

The means and standard deviations are learned during `prep()` from the training data and stored for use when applying the transformation to new data during `bake()`.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

References

van den Berg, R.A., Hoefsloot, H.C., Westerhuis, J.A., Smilde, A.K., and van der Werf, M.J. 2006. Centering, scaling, and transformations: improving the biological information content of metabolomics data. BMC Genomics, 7:142.

See Also

`step_measure_scale_auto()`, `step_measure_center()`

Other measure-scaling: `step_measure_center()`, `step_measure_scale_auto()`, `step_measure_scale_range()`, `step_measure_scale_vast()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_scale_pareto() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_scale_range *Range Scaling*

Description

`step_measure_scale_range()` creates a *specification* of a recipe step that applies range scaling at each measurement location. This centers and divides by the range (max - min) of each variable.

Usage

```
step_measure_scale_range(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_params = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_scale_range")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.

`learned_params`
: A named list containing learned means and locations for each measure column. This is `NULL` until the step is trained.

`skip`
: A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id`
: A character string that is unique to this step to identify it.

Details

Range scaling centers the data and divides by the range, giving bounded values suitable for methods sensitive to variable scale.

For a data matrix X, the transformation is:

$$X_{scaled} = \frac{X - \bar{X}}{\max(X) - \min(X)}$$

where $\bar{X}$ is the column-wise mean and the range is computed from the training data.

If a column has zero range (constant values), that column is only centered, not scaled.

The means and ranges are learned during `prep()` from the training data and stored for use when applying the transformation to new data during `bake()`.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

See Also

`step_measure_scale_auto()`, `step_measure_center()`

Other measure-scaling: `step_measure_center()`, `step_measure_scale_auto()`, `step_measure_scale_pareto()`
`step_measure_scale_vast()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_scale_range() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_scale_vast` *VAST Scaling (Variable Stability Scaling)*

Description

`step_measure_scale_vast()` creates a *specification* of a recipe step that applies VAST (Variable Stability) scaling at each measurement location. This focuses on variables with high stability (low coefficient of variation).

Usage

```
step_measure_scale_vast(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_params = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_scale_vast")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_params`	A named list containing learned means and locations for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

VAST scaling divides by the product of the standard deviation and the coefficient of variation (CV = SD/mean). This gives more weight to variables that are stable across samples (low CV).

For a data matrix X, the transformation is:

$$X_{scaled} = \frac{X - \bar{X}}{s_X \cdot CV_X}$$

where $\bar{X}$, s_X, and $CV_X = s_X/|\bar{X}|$ are computed from the training data.

If a column has zero divisor (constant values or zero mean), that column is only centered, not scaled.

The means, standard deviations, and CVs are learned during `prep()` from the training data and stored for use when applying the transformation to new data during `bake()`.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

References

van den Berg, R.A., Hoefsloot, H.C., Westerhuis, J.A., Smilde, A.K., and van der Werf, M.J. 2006. Centering, scaling, and transformations: improving the biological information content of metabolomics data. BMC Genomics, 7:142.

See Also

`step_measure_scale_auto()`, `step_measure_scale_pareto()`

Other measure-scaling: `step_measure_center()`, `step_measure_scale_auto()`, `step_measure_scale_pareto()`
`step_measure_scale_range()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_scale_vast() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_smooth_gaussian` *Gaussian Kernel Smoothing*

Description

`step_measure_smooth_gaussian()` creates a *specification* of a recipe step that applies Gaussian kernel smoothing. This produces smooth results while preserving the general shape of peaks.

Usage

```
step_measure_smooth_gaussian(
  recipe,
  measures = NULL,
  sigma = 1,
  window = NULL,
  edge_method = c("reflect", "constant", "NA"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_smooth_gaussian")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`sigma`	The standard deviation of the Gaussian kernel. Default is 1. Larger values produce more smoothing. Tunable via `smooth_sigma()`.
`window`	The window size. If `NULL` (default), automatically set to `ceiling(6 * sigma) | 1` (6 sigma rule, ensuring odd).
`edge_method`	How to handle edges. One of `"reflect"` (default), `"constant"`, or `"NA"`.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Gaussian smoothing convolves the spectrum with a Gaussian kernel:

$$G(x) = \exp(-x^2/2\sigma^2)$$

The kernel is normalized to sum to 1. This provides smooth, natural-looking results that preserve peak shapes better than moving average.

Value

An updated recipe with the new step added.

See Also

Other measure-smoothing: step_measure_despike(), step_measure_filter_fourier(), step_measure_savitzky_golay(), step_measure_smooth_ma(), step_measure_smooth_median(), step_measure_smooth_wavelet()

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_smooth_gaussian(sigma = 2) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_smooth_ma

Moving Average Smoothing

Description

step_measure_smooth_ma() creates a *specification* of a recipe step that applies moving average smoothing to measurement data. This is a simple and fast method for reducing high-frequency noise.

Usage

```
step_measure_smooth_ma(
  recipe,
  measures = NULL,
  window = 5L,
  edge_method = c("reflect", "constant", "NA"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_smooth_ma")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`window`	The window size for the moving average. Must be an odd integer of at least 3. Default is 5. Larger values produce more smoothing. Tunable via `smooth_window()`.
`edge_method`	How to handle edges where the full window doesn't fit. One of `"reflect"` (default, reflects values at boundaries), `"constant"` (pads with edge values), or `"NA"` (returns NA for edge values).
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Moving average smoothing replaces each point with the mean of its neighbors within a sliding window. This is equivalent to convolution with a uniform kernel.

For a window size of `w`, the smoothed value at position `i` is:

$$y_i = \frac{1}{w} \sum_{j=-k}^{k} x_{i+j}$$

where `k = (w-1)/2` is the half-window size.

Value

An updated recipe with the new step added.

See Also

Other measure-smoothing: `step_measure_despike()`, `step_measure_filter_fourier()`, `step_measure_savitzky_golay()`, `step_measure_smooth_gaussian()`, `step_measure_smooth_median()`, `step_measure_smooth_wavelet()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_smooth_ma(window = 5) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_smooth_median` *Median Filter Smoothing*

Description

`step_measure_smooth_median()` creates a *specification* of a recipe step that applies median filter smoothing. This is a robust method that is particularly effective at removing spike noise while preserving edges.

Usage

```
step_measure_smooth_median(
  recipe,
  measures = NULL,
  window = 5L,
  edge_method = c("reflect", "constant", "NA"),
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_smooth_median")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`window`	The window size for the moving average. Must be an odd integer of at least 3. Default is 5. Larger values produce more smoothing. Tunable via `smooth_window()`.
`edge_method`	How to handle edges where the full window doesn't fit. One of `"reflect"` (default, reflects values at boundaries), `"constant"` (pads with edge values), or `"NA"` (returns NA for edge values).
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Median filtering replaces each point with the median of its neighbors within a sliding window. Unlike moving average, median filtering is robust to outliers and spikes, making it ideal for:

- Removing cosmic ray spikes in Raman spectroscopy
- Cleaning detector artifacts
- Preserving sharp edges while removing noise

Value

An updated recipe with the new step added.

See Also

Other measure-smoothing: `step_measure_despike()`, `step_measure_filter_fourier()`, `step_measure_savitzky_golay()`, `step_measure_smooth_gaussian()`, `step_measure_smooth_ma()`, `step_measure_smooth_wavelet()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_smooth_median(window = 5) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_smooth_wavelet
Wavelet Denoising

Description

`step_measure_smooth_wavelet()` creates a *specification* of a recipe step that applies wavelet-based denoising to measurement data. This method is particularly effective for signals with localized features like peaks.

Usage

```
step_measure_smooth_wavelet(
  recipe,
  measures = NULL,
  wavelet = "DaubExPhase",
  filter_number = 4L,
  threshold_type = c("soft", "hard"),
  threshold_policy = c("universal", "sure", "cv"),
  levels = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_smooth_wavelet")
)
```

Arguments

`recipe`	A recipe object.
`measures`	An optional character vector of measure column names.
`wavelet`	The wavelet family to use. Default is `"DaubExPhase"`. Options include `"DaubExPhase"`, `"DaubLeAsworthy"`, `"Lawton"`.
`filter_number`	The filter number within the wavelet family. Default is 4. Higher numbers give smoother wavelets.
`threshold_type`	Type of thresholding: `"soft"` (default) or `"hard"`. Soft thresholding shrinks coefficients toward zero; hard thresholding sets small coefficients exactly to zero.
`threshold_policy`	How to determine the threshold: • `"universal"` (default): Uses universal threshold sqrt(2*log(n)) • `"sure"`: Stein's Unbiased Risk Estimate • `"cv"`: Cross-validation
`levels`	Number of decomposition levels. Default is `NULL` (auto).
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Wavelet denoising works by:

1. Decomposing the signal into wavelet coefficients
2. Thresholding small coefficients (presumed to be noise)
3. Reconstructing the signal from remaining coefficients

This approach is powerful because:

- It adapts to local signal characteristics
- It preserves sharp features like peaks
- It can separate noise from signal at multiple scales

Requires the `wavethresh` package to be installed.

Value

An updated recipe with the new step added.

Note

Wavelet transforms require signal lengths that are powers of 2. Signals are automatically padded to the next power of 2 and trimmed after processing.

See Also

Other measure-smoothing: `step_measure_despike()`, `step_measure_filter_fourier()`, `step_measure_savitzky_golay()`, `step_measure_smooth_gaussian()`, `step_measure_smooth_ma()`, `step_measure_smooth_median()`

Examples

```
library(recipes)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_smooth_wavelet() |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_snv` *Standard Normal Variate (SNV) Transformation*

Description

`step_measure_snv()` creates a *specification* of a recipe step that applies Standard Normal Variate transformation to spectral data. SNV normalizes each spectrum to have zero mean and unit standard deviation.

Usage

```
step_measure_snv(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_snv")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed. Use this to limit processing to specific measure columns when working with multiple measurement types.

`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`skip`	A logical. Should the step be skipped when the recipe is baked by `recipes::bake()`? While all operations are baked when `recipes::prep()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.

Details

Standard Normal Variate (SNV) is a row-wise transformation that normalizes each spectrum independently. For a spectrum x, the transformation is:

$$SNV(x) = \frac{x - \bar{x}}{s_x}$$

where $\bar{x}$ is the mean and s_x is the standard deviation of the spectrum values.

SNV is commonly used to remove multiplicative effects of scatter and particle size in NIR spectroscopy. After SNV transformation, each spectrum will have a mean of zero and a standard deviation of one.

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

The measurement locations are preserved unchanged.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with column `terms` (set to `".measures"`) and `id` is returned.

See Also

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_referenc`
`step_measure_subtract_blank()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

rec <-
  recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_snv() |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_standard_addition

Standard Addition Correction

Description

`step_measure_standard_addition()` creates a *specification* of a recipe step that performs standard addition correction to compensate for matrix effects. This method creates a sample-specific calibration for each unknown to accurately quantify in the presence of matrix interference.

Usage

```
step_measure_standard_addition(
  recipe,
  ...,
  addition_col = "addition",
  sample_id_col,
  min_points = 3,
  output_suffix = "_corrected",
  diagnostics = TRUE,
  role = "outcome",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_standard_addition")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose response columns to correct using standard addition.
`addition_col`	Name of the column containing the amount of standard added (spike amount). Default is `"addition"`.

`sample_id_col`	Name of the column identifying unique samples. Each sample gets its own standard addition curve.
`min_points`	Minimum number of addition points required per sample. Default is 3.
`output_suffix`	Suffix for output concentration columns. Default is `"_corrected"`.
`diagnostics`	Include diagnostic information (R^2, slope, intercept)? Default is TRUE.
`role`	Recipe role for new columns. Default is `"outcome"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Standard Addition Method:

Standard addition works by:

1. Splitting each unknown sample into multiple aliquots
2. Adding increasing known amounts of analyte to each aliquot
3. Measuring response for all aliquots
4. Fitting regression: `response = intercept + slope * addition`
5. Calculating original concentration from the x-intercept

The x-intercept (where response = 0) is at `-intercept / slope`. Since intercept is positive (response from original sample) and slope is positive (response increases with addition), the original concentration is: `concentration = intercept / slope`

Data Format:

The input data should have:

- A sample identifier column (each unique sample)
- An addition amount column (0 for unspiked, then increasing amounts)
- Response column(s) to be corrected

When to Use:

Use standard addition when:

- Significant matrix effects are present
- Matrix-matched calibrators are not available
- Sample-to-sample matrix variation is expected

Limitations:

- Requires multiple measurements per sample
- Assumes linear response over the addition range
- Does not correct for non-specific interferences

Value

An updated recipe with the new step added.

See Also

measure_matrix_effect(), measure_calibration()

Other calibration: measure_matrix_effect(), step_measure_dilution_correct(), step_measure_surrogate_

Examples

```
library(recipes)

# Standard addition data for two samples
sa_data <- data.frame(
  sample_id = rep(c("Sample1", "Sample2"), each = 4),
  addition = rep(c(0, 10, 20, 30), 2),
  response = c(
    # Sample 1: original conc ~15
    150, 250, 350, 450,
    # Sample 2: original conc ~25
    250, 350, 450, 550
  )
)

rec <- recipe(~ ., data = sa_data) |>
  step_measure_standard_addition(
    response,
    addition_col = "addition",
    sample_id_col = "sample_id"
  ) |>
  prep()

bake(rec, new_data = NULL)
```

---

step_measure_subtract_blank

Subtract Blank Measurement

Description

step_measure_subtract_blank() creates a *specification* of a recipe step that subtracts or divides by a blank measurement. The blank can be provided externally or learned from training data.

Usage

```
step_measure_subtract_blank(
  recipe,
  blank = NULL,
  blank_col = NULL,
  blank_value = NULL,
  method = "subtract",
  measures = NULL,
```

```
  role = NA,
  trained = FALSE,
  learned_blank = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_subtract_blank")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`blank`	An optional external blank to use. Can be: • A `measure_tbl` object with `location` and `value` columns • A numeric vector (must match the number of locations in data) • A data.frame with `location` and `value` columns (will be interpolated) If `NULL`, the blank is learned from training data using `blank_col` and `blank_value`.
`blank_col`	An optional column name (unquoted) that identifies sample types. Used with `blank_value` to identify blank samples in training data.
`blank_value`	The value in `blank_col` that identifies blank samples. When the step is prepped, the mean of all blank samples is computed and stored for use during baking.
`method`	The correction method to apply: • `"subtract"` (default): Subtract the blank from each spectrum • `"divide"`: Divide each spectrum by the blank
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_blank`	A named list containing the learned blank values for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

Blank subtraction is a fundamental preprocessing step in analytical chemistry. It removes background signal that is present in all measurements but is not related to the analyte of interest.

Two modes of operation:

1. **External blank**: You provide a blank spectrum directly via the `blank` argument. This is useful when you have a known reference blank.

2. **Learned blank**: You specify which samples are blanks in your training data using `blank_col` and `blank_value`. During `prep()`, the mean of all blank samples is computed and stored. This approach is useful for batch-specific blank correction.

Common use cases:

- UV-Vis: Remove solvent absorbance
- IR: Remove atmospheric CO2/H2O interference
- Fluorescence: Remove buffer background and Raman scatter
- Chromatography: Remove ghost peaks and solvent artifacts

No selectors should be supplied to this step function. The data should be in the internal format produced by `step_measure_input_wide()` or `step_measure_input_long()`.

Value

An updated version of `recipe` with the new step added to the sequence of any existing operations.

Tidying

When you `tidy()` this step, a tibble with columns `terms`, `method`, `blank_source`, and `id` is returned.

See Also

`step_measure_subtract_reference()` for simpler external reference

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_referenc` `step_measure_snv()`, `step_measure_subtract_reference()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Example with external blank (numeric vector)
blank_spectrum <- rep(0.1, 100)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_subtract_blank(blank = blank_spectrum)

# Example learning blank from training data
# (assuming sample_type column with "blank" values)
# rec <- recipe(outcome ~ ., data = my_data) |>
#   step_measure_input_long(...) |>
#   step_measure_subtract_blank(blank_col = sample_type, blank_value = "blank")
```

`step_measure_subtract_reference` *Subtract or Divide by Reference Spectrum*

Description

`step_measure_subtract_reference()` creates a *specification* of a recipe step that subtracts or divides each spectrum by an external reference. This is a simpler version of `step_measure_subtract_blank()` that always uses an externally provided reference.

Usage

```
step_measure_subtract_reference(
  recipe,
  reference,
  method = "subtract",
  measures = NULL,
  role = NA,
  trained = FALSE,
  learned_ref = NULL,
  skip = FALSE,
  id = recipes::rand_id("measure_subtract_reference")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`reference`	A required external reference spectrum. Can be: • A `measure_tbl` object with `location` and `value` columns • A numeric vector (must match the number of locations in data) • A data.frame with `location` and `value` columns (will be interpolated)
`method`	The correction method to apply: • `"subtract"` (default): Subtract the blank from each spectrum • `"divide"`: Divide each spectrum by the blank
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns (columns with class `measure_list`) will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`learned_ref`	A named list containing the validated reference values for each measure column. This is `NULL` until the step is trained.
`skip`	A logical. Should the step be skipped when the recipe is baked?
`id`	A character string that is unique to this step to identify it.

Details

This step applies a simple reference correction to each spectrum:

- `method = "subtract"`: `result = sample - reference`
- `method = "divide"`: `result = sample / reference`

Unlike `step_measure_subtract_blank()`, this step always requires an externally provided reference and does not support learning from training data.

Value

An updated version of `recipe` with the new step added.

See Also

`step_measure_subtract_blank()` for blank correction with learning

Other measure-preprocessing: `step_measure_absorbance()`, `step_measure_calibrate_x()`, `step_measure_calibrate_y()`, `step_measure_derivative()`, `step_measure_derivative_gap()`, `step_measure_emsc()`, `step_measure_kubelka_munk()`, `step_measure_log()`, `step_measure_map()`, `step_measure_msc()`, `step_measure_normalize_istd()`, `step_measure_osc()`, `step_measure_ratio_referenc` `step_measure_snv()`, `step_measure_subtract_blank()`, `step_measure_transmittance()`

Examples

```
library(recipes)

# Create a reference spectrum
ref_spectrum <- rep(1.0, 100)

rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_subtract_reference(reference = ref_spectrum, method = "divide")
```

---

step_measure_surrogate_recovery

Surrogate/Internal Standard Recovery

Description

`step_measure_surrogate_recovery()` creates a *specification* of a recipe step that calculates recovery percentages for surrogate or internal standards. This is essential for quality control in analytical workflows where spiked compounds are used to monitor method performance.

Usage

```
step_measure_surrogate_recovery(
  recipe,
  ...,
  expected_col = NULL,
  expected_value = NULL,
  recovery_suffix = "_recovery",
  action = c("add_column", "flag", "filter"),
  flag_col = ".surrogate_pass",
  min_recovery = 70,
  max_recovery = 130,
  role = "surrogate",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_surrogate_recovery")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose surrogate columns (measured concentrations or responses).
`expected_col`	Name of a column containing expected concentrations for each sample. Mutually exclusive with `expected_value`.
`expected_value`	A fixed numeric value for expected concentration. Used when all surrogates have the same expected value. Mutually exclusive with `expected_col`.
`recovery_suffix`	Suffix appended to column names for recovery output. Default is `"_recovery"`.
`action`	What to do with recovery calculations: • `"add_column"` (default): Add new columns with recovery percentages • `"flag"`: Add a boolean column indicating if recovery is within limits • `"filter"`: Remove rows where any surrogate is outside limits
`flag_col`	Name of the flag column when `action = "flag"`. Default is `".surrogate_pass"`.
`min_recovery`	Minimum acceptable recovery percentage. Default is 70.
`max_recovery`	Maximum acceptable recovery percentage. Default is 130.
`role`	Recipe role for new recovery columns. Default is `"surrogate"`.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Recovery Calculation:

Recovery is calculated as: `recovery_pct = (measured / expected) * 100`
Where:

- `measured` is the observed concentration/response of the surrogate
- `expected` is the known spike amount or theoretical value

Acceptance Criteria:

Typical acceptance limits vary by application:

- **ICH M10 (Bioanalytical)**: 70-130% for surrogates
- **EPA Methods**: Often 50-150% or method-specific
- **FDA Guidance**: Application-specific, often 80-120%

Use Cases:

- Monitor extraction efficiency in sample preparation
- Track instrument performance across runs
- Identify samples with matrix effects or procedural errors

Value

An updated recipe with the new step added.

See Also

`step_measure_dilution_correct()`, `measure_matrix_effect()`

Other calibration: `measure_matrix_effect()`, `step_measure_dilution_correct()`, `step_measure_standard_a`

Examples

```
library(recipes)

# Example: QC data with spiked surrogates
qc_data <- data.frame(
  sample_id = paste0("QC", 1:10),
  surrogate_1 = rnorm(10, mean = 100, sd = 10),
  surrogate_2 = rnorm(10, mean = 50, sd = 5),
  analyte = rnorm(10, mean = 75, sd = 8)
)

# Add recovery columns for surrogates with expected value of 100 and 50
rec <- recipe(~ ., data = qc_data) |>
  update_role(sample_id, new_role = "id") |>
  step_measure_surrogate_recovery(
    surrogate_1,
    expected_value = 100,
    min_recovery = 80,
    max_recovery = 120
  ) |>
```

```
  prep()

bake(rec, new_data = NULL)
```

`step_measure_transmittance` *Convert Absorbance to Transmittance*

Description

`step_measure_transmittance()` creates a *specification* of a recipe step that converts absorbance values to transmittance.

Usage

```
step_measure_transmittance(
  recipe,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_transmittance")
)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`measures`	An optional character vector of measure column names to process. If `NULL` (the default), all measure columns will be processed.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the step has been trained.
`skip`	A logical. Should the step be skipped when baking?
`id`	A character string that is unique to this step.

Details

This step applies the inverse Beer-Lambert law transformation:

$$T = 10^{-A}$$

where A is absorbance and T is transmittance.

The measurement locations are preserved unchanged.

Value

An updated version of `recipe` with the new step added.

See Also

step_measure_absorbance() for the inverse transformation

Other measure-preprocessing: step_measure_absorbance(), step_measure_calibrate_x(), step_measure_calibrate_y(), step_measure_derivative(), step_measure_derivative_gap(), step_measure_emsc(), step_measure_kubelka_munk(), step_measure_log(), step_measure_map(), step_measure_msc(), step_measure_normalize_istd(), step_measure_osc(), step_measure_ratio_referenc step_measure_snv(), step_measure_subtract_blank(), step_measure_subtract_reference()

Examples

```
library(recipes)

# Convert to absorbance then back to transmittance (round-trip)
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_absorbance() |>
  step_measure_transmittance() |>
  prep()
```

---

step_measure_trim *Trim Measurements to Specified Range*

Description

step_measure_trim() creates a *specification* of a recipe step that keeps only the measurement points within the specified x-axis range(s).

Usage

```
step_measure_trim(
  recipe,
  range,
  measures = NULL,
  role = NA,
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_trim")
)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
range	A numeric vector of length 2 specifying the range to keep as c(min, max). Points with location >= min and <= max are retained.

measures	An optional character vector of measure column names to process. If NULL (the default), all measure columns will be processed.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the step has been trained.
skip	A logical. Should the step be skipped when baking?
id	A character string that is unique to this step.

Details

This step filters measurements to keep only points within the specified range. This is useful for:

- Defining integration windows (e.g., keep only 8-18 mL elution range)
- Removing noisy regions at start/end of measurement
- Focusing analysis on a region of interest

Points with location values outside the range are removed. The order of remaining points is preserved.

Value

An updated version of recipe with the new step added.

See Also

step_measure_exclude() for removing specific ranges, step_measure_resample() for interpolating to a new grid

Other region-operations: step_measure_exclude(), step_measure_resample()

Examples

```
library(recipes)

# Keep only a specific wavelength range
rec <- recipe(water + fat + protein ~ ., data = meats_long) |>
  update_role(id, new_role = "id") |>
  step_measure_input_long(transmittance, location = vars(channel)) |>
  step_measure_trim(range = c(10, 90)) |>
  prep()

bake(rec, new_data = NULL)
```

`step_measure_tucker` *Tucker Decomposition for Multi-Dimensional Data*

Description

`step_measure_tucker()` creates a *specification* of a recipe step that applies Tucker decomposition to multi-dimensional measurement data, extracting component scores as features for modeling.

Usage

```
step_measure_tucker(
  recipe,
  ...,
  ranks = 3L,
  center = TRUE,
  scale = FALSE,
  max_iter = 500L,
  tol = 1e-06,
  prefix = "tucker_",
  role = "predictor",
  trained = FALSE,
  skip = FALSE,
  id = recipes::rand_id("measure_tucker")
)
```

Arguments

`recipe`	A recipe object.
`...`	One or more selector functions to choose measure columns. If empty, all nD measure columns are used.
`ranks`	A vector of ranks for each mode. If a single integer, the same rank is used for all modes. Default is 3.
`center`	Logical. Should data be centered before decomposition? Default is `TRUE`.
`scale`	Logical. Should data be scaled before decomposition? Default is `FALSE`.
`max_iter`	Maximum number of iterations. Default is 500.
`tol`	Convergence tolerance. Default is 1e-6.
`prefix`	Prefix for output column names. Default is `"tucker_"`.
`role`	Not used.
`trained`	Logical indicating if the step has been trained.
`skip`	Logical. Should the step be skipped when baking?
`id`	Unique step identifier.

Details

Tucker decomposition (also known as higher-order SVD or multilinear SVD) decomposes a tensor into a core tensor multiplied by factor matrices along each mode. Unlike PARAFAC, Tucker allows different ranks for each mode, providing more flexibility.

Requirements:

- Input must be `measure_nd_list` with 2+ dimensions
- All samples must have the same grid (regular, aligned)
- The `multiway` package must be installed (in Suggests)

Output:

Creates numeric feature columns representing the unfolded core tensor scores for each sample.

Value

An updated recipe with the new step added.

Note

This step requires the `multiway` package. Install with: `install.packages("multiway")`

See Also

`step_measure_parafac()` for PARAFAC decomposition

Other measure-multiway: `step_measure_mcr_als()`, `step_measure_parafac()`

Examples

```
## Not run:
library(recipes)

# After ingesting 2D data as nD measurements
rec <- recipe(concentration ~ ., data = lc_dad_data) |>
  step_measure_input_long(
    absorbance,
    location = vars(time, wavelength)
  ) |>
  step_measure_tucker(ranks = c(5, 3)) |>
  prep()

bake(rec, new_data = NULL)

## End(Not run)
```

`subtract_rf_baseline` *Subtract baseline using robust fitting method*

Description

A standalone function for robust fitting baseline subtraction using local regression with iterative reweighting. For use within a recipe workflow, see `step_measure_baseline_rf()`.

Usage

```
subtract_rf_baseline(data, yvar, span = 2/3, maxit = c(5, 5))
```

Arguments

`data`	A dataframe containing the variable for baseline subtraction
`yvar`	The name of the column for baseline subtraction
`span`	Controls the amount of smoothing based on the fraction of data to use in computing each fitted value, defaults to `2/3`.
`maxit`	The number of iterations to use the robust fit, defaults to `c(5, 5)` where the first value specifies iterations for asymmetric weighting function and the second value for symmetric weighting function.

Value

A dataframe matching column in data plus `raw` and `baseline` columns

See Also

`step_measure_baseline_rf()` for the recipe step version.

Examples

```
library(dplyr)
meats_long |>
  group_by(id) |>
  subtract_rf_baseline(yvar = transmittance)
```

sum_peak_models *Sum Multiple Peak Models*

Description

Evaluates multiple peak models and sums their contributions.

Usage

```
sum_peak_models(x, models, params_list)
```

Arguments

x	Numeric vector of x values.
models	List of peak_model objects (one per peak).
params_list	List of parameter lists (one per peak).

Value

Numeric vector of summed peak values.

Examples

```
# Two overlapping Gaussian peaks
model1 <- create_peak_model("gaussian")
model2 <- create_peak_model("gaussian")
x <- seq(0, 20, by = 0.1)
params1 <- list(height = 1, center = 8, width = 1)
params2 <- list(height = 0.8, center = 12, width = 1.5)
y <- sum_peak_models(x, list(model1, model2), list(params1, params2))
plot(x, y, type = "l")
```

summary.measure_validation_report *Summarize a Validation Report*

Description

Creates a summary table of all validation sections in the report, showing section status, result counts, and notes.

Usage

```
## S3 method for class 'measure_validation_report'
summary(object, ...)
```

Arguments

`object`	A `measure_validation_report` object.
`...`	Additional arguments (currently ignored).

Value

A tibble with columns:

- `section`: Section name
- `status`: Pass/fail/info status
- `n_results`: Number of results in section
- `notes`: Additional notes

Returns `NULL` invisibly if the report has no validation sections.

Examples

```
# Create a report with some sections
report <- measure_validation_report(
  title = "Test Report",
  specificity = "No interference observed"
)
summary(report)
```

---

tidy.measure_calibration

Tidy a Calibration Curve

Description

Extract coefficients and statistics from a calibration curve in tidy format.

Usage

```
## S3 method for class 'measure_calibration'
tidy(x, ...)

## S3 method for class 'measure_calibration_verify'
tidy(x, ...)
```

Arguments

`x`	A measure_calibration object.
`...`	Additional arguments (unused).

Value

A tibble with columns:

- `term`: Coefficient name (intercept, slope, quadratic)
- `estimate`: Coefficient estimate
- `std_error`: Standard error
- `statistic`: t-statistic
- `p_value`: p-value

Examples

```
data <- data.frame(
  nominal_conc = c(0, 10, 25, 50, 100),
  response = c(0.5, 15.2, 35.8, 72.1, 148.3)
)
cal <- measure_calibration_fit(data, response ~ nominal_conc)
tidy(cal)
```

`tidy.measure_lod` *Tidy LOD/LOQ Results*

Description

Tidy LOD/LOQ Results

Usage

```
## S3 method for class 'measure_lod'
tidy(x, ...)
```

Arguments

`x`	A measure_lod, measure_loq, or measure_lod_loq object.
`...`	Additional arguments (unused).

Value

A tibble with the limit value(s) and method information.

tidy.measure_uncertainty_budget
Tidy an Uncertainty Budget

Description

Extract uncertainty budget information in tidy format.

Usage

```
## S3 method for class 'measure_uncertainty_budget'
tidy(x, type = c("components", "summary"), ...)
```

Arguments

`x`	A measure_uncertainty_budget object.
`type`	What to return: • `"components"` (default): Table of individual components • `"summary"`: Single row summary of the budget
`...`	Additional arguments (unused).

Value

A tibble with budget information.

Examples

```
u1 <- uncertainty_component("Repeatability", 0.05, type = "A", df = 9)
u2 <- uncertainty_component("Calibrator", 0.02, type = "B")
budget <- measure_uncertainty_budget(u1, u2)

tidy(budget)
tidy(budget, type = "summary")
```

tidy.measure_validation_report
Tidy a Validation Report

Description

Extracts key parameters and statistics from all validation sections into a tidy tibble format suitable for further analysis or reporting.

Usage

```
## S3 method for class 'measure_validation_report'
tidy(x, ...)
```

Arguments

x	A measure_validation_report object.
...	Additional arguments (currently ignored).

Value

A tibble with columns:

- section: Section name
- parameter: Parameter name
- value: Parameter value
- unit: Unit of measurement (if available)
- status: Pass/fail status (if available)

Returns an empty tibble if no sections contain tidy-able data.

Examples

```
# Create sample data
blank_data <- data.frame(
  response = rnorm(10, mean = 50, sd = 15),
  sample_type = "blank"
)
lod_result <- measure_lod(blank_data, response_col = "response")

report <- measure_validation_report(
  title = "Test Report",
  lod_loq = lod_result
)
tidy(report)
```

---

uncertainty_component

Create an Uncertainty Component

Description

Defines a single uncertainty component for use in an uncertainty budget. This follows ISO GUM terminology with Type A (statistical) and Type B (other means) uncertainty evaluation.

Usage

```
uncertainty_component(
  name,
  value,
  type = c("A", "B"),
  sensitivity = 1,
  df = Inf,
  distribution = c("normal", "rectangular", "triangular", "u-shaped"),
  coverage_factor = 1
)
```

Arguments

`name`	Name/description of the uncertainty source.
`value`	Standard uncertainty value (u).
`type`	Type of evaluation: • `"A"`: Statistical evaluation (from repeated measurements) • `"B"`: Evaluated by other means (from specifications, certificates, etc.)
`sensitivity`	Sensitivity coefficient (c). Default is 1. The contribution to combined uncertainty is `|c| * u`.
`df`	Degrees of freedom for this component. Default is `Inf` (for Type B with no DOF information).
`distribution`	Distribution assumed for Type B: • `"normal"`: Normal distribution (default for Type A) • `"rectangular"`: Uniform distribution (common for Type B) • `"triangular"`: Triangular distribution • `"u-shaped"`: U-shaped distribution
`coverage_factor`	Coverage factor (k) used to derive this value from an expanded uncertainty. Default is 1 (value is already standard uncertainty).

Details

Type A Evaluation:

For Type A components, the standard uncertainty is typically the standard error of the mean: `u = s / sqrt(n)`, with `df = n - 1`.

Type B Evaluation:

For Type B components from expanded uncertainties with coverage k: `u = U / k`. For rectangular distributions: `u = a / sqrt(3)`.

Value

An `uncertainty_component` object.

See Also

measure_uncertainty_budget() for combining components, measure_uncertainty() for quick uncertainty calculation.

Examples

```
# Type A: Repeatability from 10 measurements
u_repeat <- uncertainty_component(
  name = "Repeatability",
  value = 0.05,  # Standard error of mean
  type = "A",
  df = 9
)

# Type B: Calibrator uncertainty from certificate (k=2)
u_cal <- uncertainty_component(
  name = "Calibrator",
  value = 0.02 / 2,  # Divide expanded uncertainty by k
  type = "B",
  df = 50
)

# Type B: Temperature effect (rectangular distribution)
u_temp <- uncertainty_component(
  name = "Temperature",
  value = 0.1 / sqrt(3),  # Half-width / sqrt(3) for rectangular
  type = "B",
  distribution = "rectangular"
)
```

---

uncertainty_type_a *Create Type A Uncertainty from Repeated Measurements*

Description

Helper function to calculate Type A uncertainty from a vector of repeated measurements.

Usage

```
uncertainty_type_a(x, name = "Type A", sensitivity = 1)
```

Arguments

x	Numeric vector of repeated measurements.
name	Name for this uncertainty component.
sensitivity	Sensitivity coefficient (default 1).

Value

An uncertainty_component object.

Examples

```
measurements <- c(10.1, 10.3, 9.9, 10.2, 10.0)
u_repeat <- uncertainty_type_a(measurements, "Repeatability")
```

---

uncertainty_type_b_expanded

Create Type B Uncertainty from Expanded Uncertainty

Description

Helper function to create a Type B uncertainty component from an expanded uncertainty value (e.g., from a certificate).

Usage

```
uncertainty_type_b_expanded(
  expanded_U,
  k = 2,
  name = "Type B",
  df = Inf,
  sensitivity = 1
)
```

Arguments

expanded_U	The expanded uncertainty value.
k	Coverage factor used for the expanded uncertainty.
name	Name for this uncertainty component.
df	Degrees of freedom (default Inf).
sensitivity	Sensitivity coefficient (default 1).

Value

An uncertainty_component object.

Examples

```
# From a calibrator certificate: U = 0.05, k = 2
u_cal <- uncertainty_type_b_expanded(0.05, k = 2, name = "Calibrator")
```

`uncertainty_type_b_rectangular`
Create Type B Uncertainty from Rectangular Distribution

Description

Helper function to create a Type B uncertainty component from a rectangular (uniform) distribution, common for specifications or tolerances.

Usage

```
uncertainty_type_b_rectangular(half_width, name = "Type B", sensitivity = 1)
```

Arguments

`half_width`	The half-width of the rectangular distribution (a). Standard uncertainty will be `a / sqrt(3)`.
`name`	Name for this uncertainty component.
`sensitivity`	Sensitivity coefficient (default 1).

Value

An uncertainty_component object.

Examples

```
# Temperature stability +/- 0.5 degrees
u_temp <- uncertainty_type_b_rectangular(0.5, name = "Temperature")
```

`unregister_peak_algorithm`
Unregister a Peak Detection Algorithm

Description

Removes a peak detection algorithm from the registry.

Usage

```
unregister_peak_algorithm(name)
```

Arguments

`name`	Algorithm name to remove.

Value

Invisible `TRUE` if removed, `FALSE` if not found.

See Also

`register_peak_algorithm()`

`unregister_peak_model` *Unregister a Peak Model*

Description

Removes a peak model from the registry.

Usage

```
unregister_peak_model(name)
```

Arguments

`name`	Model name to remove.

Value

Invisible `TRUE` if removed, `FALSE` if not found.

`validate_measure` *Validate measure data*

Description

Performs comprehensive validation checks on measure data, including axis monotonicity, duplicate detection, missing value detection, and spacing regularity.

Usage

```
validate_measure(
  x,
  checks = c("monotonic", "duplicates", "missing", "spacing"),
  tolerance = 1e-06,
  action = c("error", "warn", "message")
)
```

Arguments

x	A measure_tbl, measure_list, or data frame with measure column.
checks	Character vector of checks to perform. Default is all checks: "monotonic", "duplicates", "missing", "spacing".
tolerance	Numeric tolerance for spacing regularity check. Default is 1e-6.
action	What to do when validation fails: "error" (default), "warn", or "message".

Value

Invisibly returns a list with validation results. Each element is a list with valid (logical), message (character), and details.

Examples

```
# Create valid measure data
spec <- new_measure_tbl(location = 1:100, value = sin(1:100 / 10))
validate_measure(spec)

# Data with issues
spec_dup <- new_measure_tbl(location = c(1, 2, 2, 3), value = c(1, 2, 3, 4))
try(validate_measure(spec_dup))

# Only check specific issues
validate_measure(spec, checks = c("monotonic", "missing"))
```

---

validate_peak_model_params

Validate Peak Model Parameters

Description

Checks that a parameter list has all required parameters for a model.

Usage

```
validate_peak_model_params(model, params)
```

Arguments

model	A peak_model object.
params	Named list of parameters to validate.

Value

Invisible TRUE if valid, otherwise throws an error.

`window_side` *Parameter for measure steps*

Description

`window_side()` and `differentiation_order()` are used with Savitzky-Golay processing.

Usage

```
window_side(range = c(1L, 5L), trans = NULL)

differentiation_order(range = c(0L, 4L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Details

This parameter is often used to correct for zero-count data in tables or proportions.

Value

A function with classes `"quant_param"` and `"param"`.

Examples

```
window_side()
differentiation_order()
```

Index

* **accuracy**
  measure_accuracy, 49
  measure_carryover, 63
  measure_linearity, 82
* **batch-correction**
  step_measure_batch_reference, 198
* **calibration**
  measure_matrix_effect, 92
  step_measure_dilution_correct, 221
  step_measure_standard_addition, 321
  step_measure_surrogate_recovery, 327
* **control-charts**
  measure_control_chart, 66
  measure_control_limits, 67
  measure_system_suitability, 110
* **datasets**
  glucose_bioreactors, 36
  hplc_chromatograms, 39
  maldi_spectra, 48
  measure_column_patterns, 64
  measure_sample_types, 105
  meats_long, 122
  sec_calibration, 146
  sec_chromatograms, 147
* **drift-correction**
  step_measure_drift_linear, 223
  step_measure_drift_qc_loess, 225
  step_measure_drift_spline, 227
  step_measure_qc_bracket, 291
* **input/output steps**
  step_measure_input_long, 235
  step_measure_input_wide, 238
  step_measure_output_long, 274
  step_measure_output_wide, 276
* **measure-align**
  step_measure_align_cow, 153
  step_measure_align_dtw, 154
  step_measure_align_ptw, 156
  step_measure_align_reference, 157
  step_measure_align_shift, 159
* **measure-augmentation**
  step_measure_augment_noise, 160
  step_measure_augment_scale, 162
  step_measure_augment_shift, 163
* **measure-baseline**
  step_measure_baseline_airpls, 164
  step_measure_baseline_als, 166
  step_measure_baseline_arpls, 168
  step_measure_baseline_aspls, 170
  step_measure_baseline_auto, 172
  step_measure_baseline_custom, 173
  step_measure_baseline_fastchrom, 176
  step_measure_baseline_gpc, 177
  step_measure_baseline_iarpls, 180
  step_measure_baseline_minima, 181
  step_measure_baseline_morph, 183
  step_measure_baseline_morphological, 184
  step_measure_baseline_poly, 186
  step_measure_baseline_py, 188
  step_measure_baseline_rf, 191
  step_measure_baseline_rolling, 193
  step_measure_baseline_snip, 194
  step_measure_baseline_tophat, 196
  step_measure_detrend, 219
* **measure-channel**
  step_measure_channel_align, 208
  step_measure_channel_combine, 210
  step_measure_channel_ratio, 212
* **measure-chromatography**
  step_measure_mw_averages, 255
  step_measure_mw_distribution, 257
  step_measure_mw_fractions, 259

∗ **measure-differencing**
step_measure_savitzky_golay, 303
∗ **measure-features**
step_measure_bin, 200
step_measure_integrals, 240
step_measure_moments, 251
step_measure_ratios, 300
∗ **measure-multiway**
step_measure_mcr_als, 249
step_measure_parafac, 278
step_measure_tucker, 333
∗ **measure-normalization**
step_measure_normalize_auc, 260
step_measure_normalize_max, 264
step_measure_normalize_peak, 266
step_measure_normalize_range, 268
step_measure_normalize_sum, 269
step_measure_normalize_vector, 271
∗ **measure-preprocessing**
step_measure_absorbance, 151
step_measure_calibrate_x, 201
step_measure_calibrate_y, 204
step_measure_derivative, 214
step_measure_derivative_gap, 215
step_measure_emsc, 229
step_measure_kubelka_munk, 243
step_measure_log, 245
step_measure_map, 246
step_measure_msc, 253
step_measure_normalize_istd, 262
step_measure_osc, 272
step_measure_ratio_reference, 298
step_measure_snv, 319
step_measure_subtract_blank, 323
step_measure_subtract_reference, 326
step_measure_transmittance, 330
∗ **measure-qc**
step_measure_impute, 234
step_measure_qc_outlier, 293
step_measure_qc_saturated, 295
step_measure_qc_snr, 296
∗ **measure-region**
step_measure_interpolate, 242
∗ **measure-scaling**
step_measure_center, 206
step_measure_scale_auto, 305
step_measure_scale_pareto, 307
step_measure_scale_range, 309
step_measure_scale_vast, 311
∗ **measure-smoothing**
step_measure_despike, 217
step_measure_filter_fourier, 232
step_measure_savitzky_golay, 303
step_measure_smooth_gaussian, 313
step_measure_smooth_ma, 314
step_measure_smooth_median, 316
step_measure_smooth_wavelet, 317
∗ **method-comparison**
measure_bland_altman, 55
measure_deming_regression, 70
measure_passing_bablok, 96
measure_proficiency_score, 99
∗ **peak-deconvolution**
add_param_jitter, 7
assess_deconv_quality, 10
check_quality_gates, 23
initialize_peak_params, 42
optimize_deconvolution, 128
∗ **peak-models**
bigaussian_peak_model, 19
emg_peak_model, 28
gaussian_peak_model, 32
lorentzian_peak_model, 47
∗ **peak-operations**
step_measure_peaks_deconvolve, 280
step_measure_peaks_detect, 282
step_measure_peaks_filter, 285
step_measure_peaks_integrate, 286
step_measure_peaks_properties, 288
step_measure_peaks_to_table, 289
∗ **precision**
measure_gage_rr, 76
measure_intermediate_precision, 80
measure_repeatability, 102
measure_reproducibility, 104
∗ **region-operations**
step_measure_exclude, 231
step_measure_resample, 301
step_measure_trim, 331

add_param_jitter, 7, *10*, *23*, *42*, *129*
add_validation_section, 8

align_max_shift, 9
align_max_shift(), *159*
align_segment_length *(align_max_shift)*, 9
align_segment_length(), *153*
all_pass, 9
assess_deconv_quality, *8*, 10, *23*, *42*, *129*
assess_deconv_quality(), *282*
augment.measure_calibration, 11
autoplot-measure, 12
autoplot.measure_bland_altman, 13
autoplot.measure_calibration, 14
autoplot.measure_calibration(), *57*, *59*
autoplot.measure_control_chart, 15
autoplot.measure_control_chart(), *67*
autoplot.measure_deming_regression, 15
autoplot.measure_linearity, 16
autoplot.measure_list *(autoplot-measure)*, 12
autoplot.measure_matrix_effect, 16
autoplot.measure_passing_bablok *(autoplot.measure_deming_regression)*, 15
autoplot.measure_proficiency_score, 17
autoplot.measure_tbl *(autoplot-measure)*, 12
autoplot.measure_uncertainty_budget, 17
autoplot.measure_uncertainty_budget(), *114*
autoplot.recipe *(autoplot-measure)*, 12

bake(), *236*, *239*, *275*, *277*
baseline_alpha *(baseline_lambda)*, 18
baseline_alpha(), *170*
baseline_asymmetry *(baseline_lambda)*, 18
baseline_asymmetry(), *167*, *168*
baseline_degree *(baseline_lambda)*, 18
baseline_degree(), *186*
baseline_half_window *(baseline_lambda)*, 18
baseline_lambda, 18
baseline_lambda(), *165*, *167*, *168*
baseline_span *(baseline_lambda)*, 18
baseline_window *(baseline_lambda)*, 18
bigaussian_peak_model, 19, *28*, *32*, *47*
bin_width, 20
bioreactors_large *(glucose_bioreactors)*, 36
bioreactors_small *(glucose_bioreactors)*, 36

check_axis_consistency, 20
check_measure_recipe, 21
check_quality_gates, *8*, *10*, 23, *42*, *129*
create_peak_model, 24
create_peak_model(), *137*, *144*
criteria_bioanalytical *(criteria_presets)*, 24
criteria_bland_altman *(criteria_presets)*, 24
criteria_ich_q2 *(criteria_presets)*, 24
criteria_matrix_effects *(criteria_presets)*, 24
criteria_method_comparison *(criteria_presets)*, 24
criteria_presets, 24
criteria_proficiency_testing *(criteria_presets)*, 24
criteria_proficiency_testing(), *100*
criteria_surrogate_recovery *(criteria_presets)*, 24
criterion, 26
criterion(), *53*, *69*, *70*

derivative_gap *(derivative_order)*, 27
derivative_order, 27
derivative_segment *(derivative_order)*, 27
despike_threshold *(smooth_window)*, 150
despike_threshold(), *218*
differentiation_order *(window_side)*, 347

emg_peak_model, *19*, 28, *32*, *47*
emsc_degree *(bin_width)*, 20

find_measure_cols, 29
find_measure_nd_cols, 29
find_peaks_cols, 30

fortify-measure, 30
fortify.measure_calibration, 31
fortify.measure_list
(*fortify-measure*), 30
fortify.measure_tbl
(*fortify-measure*), 30
fourier_cutoff (*smooth_window*), 150
fourier_cutoff(), *233*

gaussian_peak_model, *19*, *28*, 32, *47*
gaussian_peak_model(), *282*
get_failures, 33
get_measure_col_ndim, 33
get_peak_algorithm, 34
get_peak_algorithm(), *130*, *143*
get_validation_section, 35
glance.measure_calibration, 35
glucose_bioreactors, 36

has_measure_col, 37
has_peak_algorithm, 38
has_peak_model, 38
has_validation_section, 39
hplc_chromatograms, 39, *49*, *148*

infer_axis_type, 41
infer_axis_type(), *54*
initialize_peak_params, *8*, *10*, *23*, 42, *129*
is_measure_calibration, 43
is_measure_list, 43
is_measure_list(), *123*
is_measure_nd_list, 44
is_measure_nd_list(), *124*
is_measure_nd_tbl, 45
is_measure_nd_tbl(), *125*
is_measure_tbl, 45
is_measure_tbl(), *126*
is_peak_model, 46
is_peaks_list, 46

lorentzian_peak_model, *19*, *28*, *32*, 47

maldi_spectra, 48
measure_accuracy, 49, *64*, *83*
measure_accuracy(), *64*, *83*, *94*, *100*, *119*, *121*
measure_apply, 51
measure_assess, 52
measure_assess(), *9*, *26*, *27*, *33*, *69*, *70*
measure_axis_info, 54
measure_bland_altman, 55, *72*, *97*, *100*
measure_bland_altman(), *72*, *97*
measure_calibration, *11*, *14*, *31*, *36*, 56, *59–61*, *84*, *86*, *88*, *337*
measure_calibration(), *323*
measure_calibration-class
(*measure_calibration*), 56
measure_calibration_fit, 57
measure_calibration_fit(), *56*, *57*, *60–62*, *83*, *119*, *121*
measure_calibration_predict, 59
measure_calibration_predict(), *57*, *59*, *62*, *121*, *222*
measure_calibration_verify, 61
measure_carryover, *51*, 63, *83*
measure_carryover(), *51*, *119*, *121*
measure_column_patterns, 64
measure_column_summary, 65
measure_control_chart, *15*, 66, *69*, *111*
measure_control_chart(), *69*
measure_control_limits, 67, *67*, *111*
measure_control_limits(), *67*
measure_criteria, *26*, *61*, 69
measure_criteria(), *27*, *53*, *62*
measure_deming_regression, *56*, 70, *97*, *100*
measure_deming_regression(), *56*, *97*
measure_detect_drift, 72
measure_detect_drift(), *226*
measure_dim_names, 73
measure_dim_units, 74
measure_fold, 75
measure_fold(), *114*, *115*
measure_gage_rr, 76, *81*, *103*, *105*
measure_grid_info, 77
measure_identify_columns, 78
measure_intermediate_precision, *77*, 80, *103*, *105*
measure_intermediate_precision(), *77*, *103*, *105*, *119*, *121*
measure_is_regular, 81
measure_linearity, *51*, *64*, 82
measure_linearity(), *16*, *51*, *119*, *121*
measure_lod, 84
measure_lod(), *87*, *89*, *119*, *121*
measure_lod_loq, 86

measure_lod_loq(), *85*, *89*, *119*, *121*
measure_loq, 87
measure_loq(), *85*, *87*, *119*, *121*
measure_map, 89
measure_map(), *91*, *92*, *248*
measure_map_safely, 91
measure_map_safely(), *90*
measure_matrix_effect, 92, *222*, *323*, *329*
measure_matrix_effect(), *323*, *329*
measure_ndim, 95
measure_ndim(), *125*
measure_packs, 95
measure_packs(), *108*, *140*
measure_passing_bablok, *56*, *72*, 96, *100*
measure_passing_bablok(), *56*, *72*
measure_plot_summary, 98
measure_proficiency_score, *56*, *72*, *97*, 99
measure_project, 101
measure_quality_summary, 102
measure_repeatability, *77*, *81*, 102, *105*
measure_repeatability(), *77*, *81*, *105*, *119*, *121*
measure_reproducibility, *77*, *81*, *103*, 104
measure_reproducibility(), *81*, *103*, *119*
measure_sample_types, 105, *116*, *117*
measure_slice, 105
measure_standardize_sample_type, 106
measure_standardize_sample_type(), *117*
measure_steps, 108
measure_steps(), *96*, *141*
measure_summarize, 109
measure_summarize(), *90*
measure_system_suitability, *67*, *69*, 110
measure_system_suitability(), *64*, *69*, *119*
measure_uncertainty, 112
measure_uncertainty(), *342*
measure_uncertainty_budget, *17*, 113, *339*
measure_uncertainty_budget(), *119*, *121*, *342*
measure_unfold, 114
measure_unfold(), *75*
measure_validate_metadata, 116
measure_validation_report, 118
measure_validation_report(), *145*
meats_long, *49*, 122

new_measure_list, 123
new_measure_list(), *126*
new_measure_nd_list, 124
new_measure_nd_list(), *125*
new_measure_nd_tbl, 125
new_measure_nd_tbl(), *124*
new_measure_tbl, 126
new_measure_tbl(), *123*
new_peak_model, 126

optimize_deconvolution, *8*, *10*, *23*, *42*, 128
optimize_deconvolution(), *282*
osc_n_components *(bin_width)*, 20
outlier_threshold, 129
outlier_threshold(), *293*

peak_algorithms, 130
peak_algorithms(), *34*, *38*, *143*, *283*, *284*
peak_location_max *(peak_location_min)*, 131
peak_location_max(), *263*, *266*, *267*
peak_location_min, 131
peak_location_min(), *262*, *266*, *267*
peak_model_area, 132
peak_model_area(), *137*
peak_model_bounds, 133
peak_model_bounds(), *127*, *135*
peak_model_gradient, 134
peak_model_gradient(), *127*, *137*
peak_model_gradient_numerical, 134
peak_model_gradient_numerical(), *134*
peak_model_initial_guess, 135
peak_model_initial_guess(), *133*
peak_model_param_names, 136
peak_model_value, 136
peak_model_value(), *127*, *132*, *134*
peak_models, 137
peak_models(), *24*, *144*, *280–282*
plot_measure_comparison, 138
prep(), *236*, *239*, *275*, *277*
print.measure_validation_report, 139

recipes::bake(), *206*, *253*, *261*, *263*, *265*, *266*, *268*, *270*, *271*, *304*, *306*, *308*, *310*, *311*, *320*
recipes::prep(), *206*, *238*, *253*, *261*, *263*, *265*, *266*, *268*, *270*, *271*, *304*, *306*, *308*, *310*, *311*, *320*
recipes::step_pca(), *277*
recipes::step_pls(), *277*
recipes::update_role(), *149*
register_measure_pack, 140
register_measure_pack(), *96*, *141*
register_measure_step, 141
register_measure_step(), *108*, *140*
register_peak_algorithm, 142
register_peak_algorithm(), *34*, *130*, *283*, *284*, *345*
register_peak_model, 143
register_peak_model(), *24*, *137*
render_validation_report, 144
render_validation_report(), *120*, *121*
rlang::as_function(), *89*, *91*, *174*, *247*

sec_calibration, 146, *148*
sec_chromatograms, *40*, *146*, 147, *147*
selections(), *238*
set_measure_roles, 149
smooth_sigma *(smooth_window)*, 150
smooth_sigma(), *313*
smooth_window, 150
smooth_window(), *218*, *315*, *316*
stats::smooth.spline(), *228*
step_measure_absorbance, 151, *203*, *205*, *215*, *216*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_absorbance(), *331*
step_measure_align_cow, 153, *156–158*, *160*
step_measure_align_dtw, 154, *154*, *157*, *158*, *160*
step_measure_align_ptw, *154*, 156, *156*, *158*, *160*
step_measure_align_reference, *154*, *156*, 157, *157*, *160*
step_measure_align_shift, *154*, *156–158*, 159
step_measure_align_shift(), *158*
step_measure_augment_noise, 160, *163*, *164*
step_measure_augment_scale, *161*, 162, *164*
step_measure_augment_shift, *161*, 163, *163*
step_measure_baseline_airpls, 164, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_als, 166, *166*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_als(), *175*, *179*, *190*
step_measure_baseline_arpls, *166*, 168, *168*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_aspls, *166*, *168*, *169*, 170, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_auto, *166*, *168*, *169*, *171*, 172, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_custom, *166*, *168*, *169*, *171*, 173, *173*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_custom(), *190*
step_measure_baseline_fastchrom, *166*, *168*, *169*, *171*, *173*, *175*, 176, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_gpc, *166*, *168*, *169*, *171*, *173*, *175*, 177, *177*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_iarpls, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, 180, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_minima, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, 181, *181*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_morph, *166*, *168*,

169, *171*, *173*, *175*, *177*, *179*, *181*, *182*, 183, *185*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_morphological, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, 184, *184*, *187*, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_poly, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, 186, *190*, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_poly(), *175*, *220*
step_measure_baseline_py, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, 188, *192*, *194*, *196*, *197*, *220*
step_measure_baseline_rf, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, 191, *194*, *196*, *197*, *220*
step_measure_baseline_rf(), *335*
step_measure_baseline_rolling, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, 193, *196*, *197*, *220*
step_measure_baseline_snip, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, 194, *194*, *197*, *220*
step_measure_baseline_tophat, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, 196, *196*, *220*
step_measure_batch_reference, 198
step_measure_bin, 200, *241*, *252*, *301*
step_measure_calibrate_x, *152*, 201, *205*, *215*, *216*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_calibrate_x(), *205*
step_measure_calibrate_y, *152*, *203*, 204, *215*, *216*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_calibrate_y(), *203*, *256*
step_measure_center, 206, *307*, *308*, *310*, *312*
step_measure_center(), *307*, *308*, *310*
step_measure_channel_align, 208, *211*, *213*
step_measure_channel_align(), *211*, *213*
step_measure_channel_combine, *209*, 210, *213*
step_measure_channel_ratio, *209*, *211*, 212
step_measure_derivative, *152*, *203*, *205*, 214, *216*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_derivative(), *216*
step_measure_derivative_gap, *152*, *203*, *205*, 215, *215*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_derivative_gap(), *215*
step_measure_despike, 217, *233*, *305*, *314*, *315*, *317*, *319*
step_measure_detrend, *166*, *168*, *169*, *171*, *173*, *175*, *177*, *179*, *181*, *182*, *184*, *185*, *187*, *190*, *192*, *194*, *196*, *197*, 219
step_measure_detrend(), *179*
step_measure_dilution_correct, *94*, 221, *323*, *329*
step_measure_dilution_correct(), *329*
step_measure_drift_linear, 223, *226*, *229*, *292*
step_measure_drift_linear(), *229*
step_measure_drift_qc_loess, *224*, 225, *229*, *292*
step_measure_drift_qc_loess(), *199*, *224*, *229*
step_measure_drift_spline, *224*, *226*, 227, *292*
step_measure_drift_spline(), *224*
step_measure_emsc, *152*, *203*, *205*, *215*, *216*, 229, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_exclude, 231, *303*, *332*
step_measure_exclude(), *303*, *332*
step_measure_filter_fourier, *218*, 232, *305*, *314*, *315*, *317*, *319*
step_measure_impute, 234, *294*, *296*, *297*
step_measure_input_long, 235, *239*, *275*, *277*

step_measure_input_long(), *167*, *175*, *179*, *187*, *190*, *192*, *203*, *205*, *207*, *220*, *254*, *261*, *265*, *267*, *269*, *270*, *272*, *304*, *306*, *308*, *310*, *312*, *320*, *325*
step_measure_input_wide, *237*, 238, *275*, *277*
step_measure_input_wide(), *167*, *175*, *179*, *187*, *190*, *192*, *203*, *205*, *207*, *220*, *254*, *261*, *265*, *267*, *269*, *270*, *272*, *304*, *306*, *308*, *310*, *312*, *320*, *325*
step_measure_integrals, *201*, 240, *252*, *301*
step_measure_interpolate, 242
step_measure_kubelka_munk, *152*, *203*, *205*, *215*, *216*, *230*, 243, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_log, *152*, *203*, *205*, *215*, *217*, *230*, *244*, 245, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_map, *152*, *203*, *205*, *215*, *217*, *230*, *244*, 246, *246*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_map(), *89*, *90*
step_measure_mcr_als, 249, *279*, *334*
step_measure_moments, *201*, *241*, 251, *301*
step_measure_msc, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, 253, *263*, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_msc(), *230*, *248*
step_measure_mw_averages, *148*, 255, *258*, *260*
step_measure_mw_averages(), *258*, *259*
step_measure_mw_distribution, *256*, 257, *260*
step_measure_mw_fractions, *256*, *258*, 259
step_measure_normalize_auc, 260, *265*, *267*, *269*, *270*, *272*
step_measure_normalize_auc(), *267*, *270*
step_measure_normalize_istd, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, 262, *274*, *299*, *320*, *325*, *327*, *331*
step_measure_normalize_max, *262*, 264, *267*, *269*, *270*, *272*
step_measure_normalize_max(), *267*, *269*, *270*
step_measure_normalize_peak, *262*, *265*, 266, *269*, *270*, *272*
step_measure_normalize_peak(), *262*, *263*
step_measure_normalize_range, *262*, *265*, *267*, 268, *270*, *272*
step_measure_normalize_range(), *265*
step_measure_normalize_sum, *262*, *265*, *267*, 269, *269*, *272*
step_measure_normalize_sum(), *262*, *265*, *272*
step_measure_normalize_vector, *262*, *265*, *267*, *269*, *270*, 271
step_measure_osc, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, 272, *299*, *320*, *325*, *327*, *331*
step_measure_output_long, *237*, *239*, 274, *277*
step_measure_output_wide, *237*, *239*, *275*, 276
step_measure_parafac, *251*, 278, *334*
step_measure_parafac(), *251*, *334*
step_measure_peaks_deconvolve, 280, *284*, *286*, *287*, *289*, *290*
step_measure_peaks_detect, 282, *282*, *286*, *287*, *289*, *290*
step_measure_peaks_detect(), *280*
step_measure_peaks_filter, *282*, *284*, 285, *287*, *289*, *290*
step_measure_peaks_integrate, *282*, *284*, 286, *286*, *289*, *290*
step_measure_peaks_properties, *282*, *284*, *286*, *287*, 288, *290*
step_measure_peaks_properties(), *284*, *285*
step_measure_peaks_to_table, *282*, *284*, *286*, *287*, 289, *289*
step_measure_peaks_to_table(), *288*
step_measure_qc_bracket, *224*, *226*, *229*, 291
step_measure_qc_outlier, *235*, 293, *296*, *297*
step_measure_qc_saturated, *235*, *294*, 295, *297*

step_measure_qc_snr, *235*, *294*, 296, *296*
step_measure_ratio_reference, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, 298, *320*, *325*, *327*, *331*
step_measure_ratios, *201*, *241*, *252*, 300
step_measure_resample, *232*, 301, *332*
step_measure_resample(), *232*, *332*
step_measure_savitzky_golay, *218*, *233*, 303, *314*, *315*, *317*, *319*
step_measure_savitzky_golay(), *214–216*, *248*
step_measure_scale_auto, *207*, 305, *308*, *310*, *312*
step_measure_scale_auto(), *207*, *308*, *310*, *312*
step_measure_scale_pareto, *207*, 307, *307*, *310*, *312*
step_measure_scale_pareto(), *207*, *307*, *312*
step_measure_scale_range, *207*, *307*, *308*, 309, *312*
step_measure_scale_vast, *207*, *307*, *308*, *310*, 311
step_measure_smooth_gaussian, *218*, *233*, *305*, 313, *315*, *317*, *319*
step_measure_smooth_ma, *218*, *233*, *305*, 314, *314*, *317*, *319*
step_measure_smooth_median, *218*, *233*, *305*, *314*, *315*, 316, *319*
step_measure_smooth_wavelet, *218*, *233*, *305*, *314*, *315*, 317, *317*
step_measure_snv, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, 319, *325*, *327*, *331*
step_measure_snv(), *248*, *254*, *269*, *272*
step_measure_standard_addition, *94*, *222*, 321, *329*
step_measure_standard_addition(), *94*
step_measure_subtract_blank, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, 323, *327*, *331*
step_measure_subtract_blank(), *299*, *326*, *327*
step_measure_subtract_reference, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, 326, *331*
step_measure_subtract_reference(), *325*
step_measure_surrogate_recovery, *94*, *222*, *323*, 327
step_measure_surrogate_recovery(), *222*
step_measure_transmittance, *152*, *203*, *205*, *215*, *217*, *230*, *244*, *246*, *248*, *254*, *263*, *274*, *299*, *320*, *325*, *327*, 330
step_measure_transmittance(), *152*
step_measure_trim, *232*, *303*, 331
step_measure_trim(), *232*, *303*
step_measure_tucker, *251*, *279*, 333
step_measure_tucker(), *279*
subtract_rf_baseline, 335
subtract_rf_baseline(), *192*
sum_peak_models, 336
summary.measure_validation_report, 336

tidy(), *168*, *175*, *179*, *187*, *190*, *192*, *203*, *205*, *207*, *220*, *237*, *239*, *254*, *305*, *320*, *325*
tidy.measure_calibration, 337
tidy.measure_calibration(), *57*, *59*
tidy.measure_calibration_verify *(tidy.measure_calibration)*, 337
tidy.measure_lod, 338
tidy.measure_uncertainty_budget, 339
tidy.measure_uncertainty_budget(), *114*
tidy.measure_validation_report, 339

uncertainty_component, 340, *343*, *344*
uncertainty_component(), *112–114*
uncertainty_type_a, 342
uncertainty_type_b_expanded, 343
uncertainty_type_b_rectangular, 344
unregister_peak_algorithm, 344
unregister_peak_model, 345

validate_measure, 345
validate_peak_model_params, 346

window_side, 347