Package 'gseries'

June 10, 2025

Title Tools for Cohesive Time Series Data

Version 3.0.0

Created April 30, 2025, at 4:57:41 PM EDT

Description 'R' version of 'G-Series', Statistics Canada's generalized system devoted to time series benchmarking and reconciliation. The methods used in 'G-Series' essentially come from Dagum and Cholette (2006) <doi:10.1007/0-387-35439-5>.

License GPL (>= 3)

URL https://StatCan.github.io/gensol-gseries/en/, https://StatCan.github.io/gensol-gseries/fr/

BugReports https://github.com/StatCan/gensol-gseries/issues/

Email g-series@statcan.gc.ca

Depends R (>= 4.0)

Imports ggplot2, ggtext, graphics, grDevices, gridExtra, lifecycle, osqp, rlang (>= 1.1.0), stats, utils, xmpdf

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

LazyData true

Contents

`benchmarking` *Restore temporal constraints*

Description

Replication of the G-Series 2.0 SAS® BENCHMARKING procedure (PROC BENCHMARKING). See the G-Series 2.0 documentation for details (Statistics Canada 2016).

This function ensures coherence between time series data of the same target variable measured at different frequencies (e.g., sub-annually and annually). Benchmarking consists of imposing the level of the benchmark series (e.g., annual data) while minimizing the revisions of the observed movement in the indicator series (e.g., sub-annual data) as much as possible. The function also allows nonbinding benchmarking where the benchmark series can also be revised.

The function may also be used for benchmarking-related topics such as *temporal distribution* (the reciprocal action of benchmarking: disaggregation of the benchmark series into more frequent observations), *calendarization* (a special case of temporal distribution) and *linking* (the connection of different time series segments into a single consistent time series).

Several series can be benchmarked in a single function call.

Usage

```
benchmarking(
  series_df,
  benchmarks_df,
  rho,
  lambda,
```

```
  biasOption,
  bias = NA,
  tolV = 0.001,
  tolP = NA,
  warnNegResult = TRUE,
  tolN = -0.001,
  var = "value",
  with = NULL,
  by = NULL,
  verbose = FALSE,

  # New in G-Series 3.0
  constant = 0,
  negInput_option = 0,
  allCols = FALSE,
  quiet = FALSE
)
```

Arguments

`series_df` (mandatory)

Data frame (object of class "data.frame") that contains the indicator time series data to be benchmarked. In addition to the series data variable(s), specified with argument `var`, the data frame must also contain two numeric variables, `year` and `period`, identifying the periods of the indicator time series.

`benchmarks_df` (mandatory)

Data frame (object of class "data.frame") that contains the benchmarks. In addition to the benchmarks data variable(s), specified with argument `with`, the data frame must also contain four numeric variables, `startYear`, `startPeriod`, `endYear` and `endPeriod`, identifying the indicator time series periods covered by each benchmark.

`rho` (mandatory)

Real number in the $[0, 1]$ interval that specifies the value of the autoregressive parameter ρ. See section **Details** for more information on the effect of parameter ρ.

`lambda` (mandatory)

Real number, with suggested values in the $[-3, 3]$ interval, that specifies the value of the adjustment model parameter λ. Typical values are `lambda = 0.0` for an additive model and `lambda = 1.0` for a proportional model.

`biasOption` (mandatory)

Specification of the bias estimation option:

- `1`: Do not estimate the bias. The bias used to correct the indicator series will be the value specified with argument `bias`.
- `2`: Estimate the bias, display the result, but do not use it. The bias used to correct the indicator series will be the value specified with argument `bias`.
- `3`: Estimate the bias, display the result and use the estimated bias to correct the indicator series. Any value specified with argument `bias` will be ignored.

Argument `biasOption` is ignored when `rho = 1.0`. See section **Details** for more information on the bias.

`bias` (optional)

Real number, or `NA`, specifying the value of the user-defined bias to be used for the correction of the indicator series prior to benchmarking. The bias is added to the indicator series with an additive model (argument `lambda = 0.0`) while it is multiplied otherwise (argument `lambda != 0.0`). No bias correction is applied when `bias = NA`, which is equivalent to specifying `bias = 0.0` when `lambda = 0.0` and `bias = 1.0` otherwise. Argument `bias` is ignored when `biasOption = 3` or `rho = 1.0`. See section **Details** for more information on the bias.

Default value is `bias = NA` (no user-defined bias).

`tolV`, `tolP` (optional)

Nonnegative real number, or `NA`, specifying the tolerance, in absolute value or percentage, to be used for the validation of the output binding benchmarks (alterability coefficient of 0.0). This validation compares the input binding benchmark values with the equivalent values calculated from the benchmarked series (output) data. Arguments `tolV` and `tolP` cannot be both specified (one must be specified while the other must be `NA`).

Example: to set a tolerance of 10 *units*, specify `tolV = 10, tolP = NA`; to set a tolerance of 1%, specify `tolV = NA, tolP = 0.01`.

Default values are `tolV = 0.001` and `tolP = NA`.

`warnNegResult` (optional)

Logical argument specifying whether a warning message is generated when a negative value created by the function in the benchmarked (output) series is smaller than the threshold specified by argument `tolN`.

Default value is `warnNegResult = TRUE`.

`tolN` (optional)

Negative real number specifying the threshold for the identification of negative values. A value is considered negative when it is smaller than this threshold.

Default value is `tolN = -0.001`.

`var` (optional)

String vector (minimum length of 1) specifying the variable name(s) in the indicator series data frame (argument `series_df`) containing the values and (optionally) the user-defined alterability coefficients of the series to be benchmarked. These variables must be numeric.

The syntax is `var = c("series1 </ alt_ser1>", "series2 </ alt_ser2>", ...)`. Default alterability coefficients of 1.0 are used when a user-defined alterability coefficients variable is not specified alongside an indicator series variable. See section **Details** for more information on alterability coefficients.

Example: `var = "value / alter"` would benchmark indicator series data frame variable `value` with the alterability coefficients contained in variable `alter` while `var = c("value / alter", "value2")` would additionally benchmark variable `value2` with default alterability coefficients of 1.0.

Default value is `var = "value"` (benchmark variable `value` using default alterability coefficients of 1.0).

`with` (optional)

String vector (same length as argument `var`), or `NULL`, specifying the variable name(s) in the benchmarks data frame (argument `benchmarks_df`) containing the values and (optionally) the user-defined alterability coefficients of the benchmarks. These variables must be numeric. Specifying `with = NULL` results in using benchmark variable(s) with the same names(s) as those specified with argu-

ment var without user-defined benchmark alterability coefficients (i.e., default alterability coefficients of 0.0 corresponding to binding benchmarks).

The syntax is with = NULL or with = c("bmk1 </ alt_bmk1>", "bmk2 </ alt_bmk2>", ...). Default alterability coefficients of 0.0 (binding benchmarks) are used when a user-defined alterability coefficients variable is not specified alongside a benchmark variable. See section **Details** for more information on alterability coefficients.

Example: with = "val_bmk" would use benchmarks data frame variable val_bmk with default benchmark alterability coefficients of 0.0 to benchmark the indicator series while with = c("val_bmk", "val_bmk2 / alt_bmk2") would additionally benchmark a second indicator series using benchmark variable val_bmk2 with the benchmark alterability coefficients contained in variable alt_bmk2.

Default value is with = NULL (same benchmark variable(s) as argument var using default benchmark alterability coefficients of 0.0).

by (optional)

String vector (minimum length of 1), or NULL, specifying the variable name(s) in the input data frames (arguments series_df and benchmarks_df) to be used to form groups (for *BY-group* processing) and allow the benchmarking of multiple series in a single function call. BY-group variables can be numeric or character (factors or not), must be present in both input data frames and will appear in all three output data frames (see section **Value**). BY-group processing is not implemented when by = NULL. See "Benchmarking Multiple Series" in section **Details** for more information.

Default value is by = NULL (no BY-group processing).

verbose (optional)

Logical argument specifying whether information on intermediate steps with execution time (real time, not CPU time) should be displayed. Note that specifying argument quiet = TRUE would *nullify* argument verbose.

Default value is verbose = FALSE.

constant (optional)

Real number that specifies a value to be temporarily added to both the indicator series and the benchmarks before solving proportional benchmarking problems (lambda != 0.0). The temporary constant is removed from the final output benchmarked series. E.g., specifying a (small) constant would allow proportional benchmarking with rho = 1 (e.g., proportional Denton benchmarking) on indicator series that include values of 0. Otherwise, proportional benchmarking with values of 0 in the indicator series is only possible when rho < 1. Specifying a constant with additive benchmarking (lambda = 0.0) has no impact on the resulting benchmarked data. The data variables in the graphTable output data frame include the constant, corresponding to the benchmarking problem that was actually solved.

Default value is constant = 0 (no temporary additive constant).

negInput_option

(optional)

Handling of negative values in the input data for proportional benchmarking (lambda != 0.0):

- 0: Do not allow negative values with proportional benchmarking. An error message is displayed in the presence of negative values in the input indicator series or benchmarks and missing (NA) values are returned for the benchmarked series. This corresponds to the G-Series 2.0 behaviour.

- 1: Allow negative values with proportional benchmarking but display a warning message.
- 2: Allow negative values with proportional benchmarking without displaying any message.

Default value is `negInput_option = 0` (do not allow negative values with proportional benchmarking).

`allCols` (optional)

Logical argument specifying whether all variables in the indicator series data frame (argument `series_df`), other than `year` and `period`, determine the set of series to benchmark. Values specified with arguments `var` and `with` are ignored when `allCols = TRUE`, which automatically implies default alterability coefficients, and variables with the same names as the indicator series must exist in the benchmarks data frame (argument `benchmarks_df`).

Default value is `allCols = FALSE`.

`quiet` (optional)

Logical argument specifying whether or not to display only essential information such as warning messages, error messages and variable (series) or BY-group information when multiple series are benchmarked in a single call to the function. We advise against *wrapping* your `benchmarking()` call with `suppressMessages()` to further suppress the display of variable (series) or BY-group information when processing multiple series as this would make troubleshooting difficult in case of issues with individual series. Note that specifying `quiet = TRUE` would also *nullify* argument `verbose`.

Default value is `quiet = FALSE`.

Details

When $\rho < 1$, this function returns the generalized least squared solution of a special case of the general regression-based benchmarking model proposed by Dagum and Cholette (2006). The model, in matrix form, is:

$$\begin{bmatrix} s^\dagger \\ a \end{bmatrix} = \begin{bmatrix} I \\ J \end{bmatrix} \theta + \begin{bmatrix} e \\ \varepsilon \end{bmatrix}$$

where

- a is the vector of length M of the benchmarks.
- $s^\dagger = \begin{cases} s + b & \text{if } \lambda = 0 \\ s \cdot b & \text{otherwise} \end{cases}$ is the vector of length T of the bias corrected indicator series values, with s denoting the initial (input) indicator series.
- b is the bias, which is specified with argument `bias` when argument `bias_option != 3` or, when `bias_option = 3`, is estimated as $\hat{b} = \begin{cases} \frac{1_M{}^{\mathrm{T}}(a - Js)}{1_M{}^{\mathrm{T}} J 1_T} & \text{if } \lambda = 0 \\ \frac{1_M{}^{\mathrm{T}} a}{1_M{}^{\mathrm{T}} Js} & \text{otherwise} \end{cases}$, where $1_X = (1, ..., 1)^{\mathrm{T}}$ is a vector of 1 of length X.
- J is the $M \times T$ matrix of temporal aggregation constraints with elements $j_{m,t} = \begin{cases} 1 & \text{if benchmark } m \text{ covers period } t \\ 0 & \text{otherwise} \end{cases}$.
- θ is the vector of the final (benchmarked) series values.
- $e \sim (0, V_e)$ is the vector of the measurement errors of $s^\dagger$ with covariance matrix $V_e = C\Omega_e C$.
- $C = \text{diag}\left(\sqrt{c_{s^\dagger}} \left|s^\dagger\right|^\lambda\right)$ where $c_{s^\dagger}$ is the vector of the alterability coefficients of $s^\dagger$, assuming $0^0 = 1$.

- Ω_e is a $T \times T$ matrix with elements $\omega_{e_{i,j}} = \rho^{|i-j|}$ representing the autocorrelation of an AR(1) process, again assuming $0^0 = 1$.
- $\varepsilon \sim (0, V_\varepsilon)$ is the vector of the measurement errors of the benchmarks a with covariance matrix $V_\varepsilon = \text{diag}(c_a a)$ where c_a is the vector of the alterability coefficients of the benchmarks a.

The generalized least squared solution is:

$$\hat{\theta} = s^\dagger + V_e J^{\mathrm{T}} \left(J V_e J^{\mathrm{T}} + V_\varepsilon\right)^+ \left(a - J s^\dagger\right)$$

where A^+ designates the Moore-Penrose inverse of matrix A.

When $\rho = 1$, the function returns the solution of the (modified) Denton method:

$$\hat{\theta} = s + W\left(a - Js\right)$$

where

- W is the upper-right corner matrix from the following matrix product

$$\begin{bmatrix} D^+\Delta^{\mathrm{T}}\Delta D^+ & J^{\mathrm{T}} \\ J & 0 \end{bmatrix}^+ \begin{bmatrix} D^+\Delta^{\mathrm{T}}\Delta D^+ & 0 \\ J & I_M \end{bmatrix} = \begin{bmatrix} I_T & W \\ 0 & W_\nu \end{bmatrix}$$

- $D = \text{diag}\left(|s|^\lambda\right)$, assuming $0^0 = 1$. Note that D corresponds to C with $c_{s^\dagger} = 1.0$ and without bias correction (arguments `bias_option = 1` and `bias = NA`).
- Δ is a $T-1 \times T$ matrix with elements $\delta_{i,j} = \begin{cases} -1 & \text{if } i = j \\ 1 & \text{if } j = i+1 \\ 0 & \text{otherwise} \end{cases}$.
- W_ν is a $M \times M$ matrix associated with the Lagrange multipliers of the corresponding minimization problem, expressed as:

$$\begin{aligned} \underset{\theta}{\text{minimize}} \quad & \sum_{t \geq 2} \left[\frac{(s_t - \theta_t)}{|s_t|^\lambda} - \frac{(s_{t-1} - \theta_{t-1})}{|s_{t-1}|^\lambda} \right]^2 \\ \text{subject to} \quad & a = J\theta \end{aligned}$$

See Quenneville et al. (2006) and Dagum and Cholette (2006) for details.

Autoregressive Parameter ρ and *bias*:

Parameter ρ (argument `rho`) is associated to the change between the (input) indicator and the (output) benchmarked series for two consecutive periods and is often called the *movement preservation parameter*. The larger the value of ρ, the more the indicator series period to period movements are preserved in the benchmarked series. With $\rho = 0$, period to period movement preservation is not enforced and the resulting benchmarking adjustments are not smooth, as in the case of prorating ($\rho = 0$ and $\lambda = 0.5$) where the adjustments take the shape of a *step function*. At the other end of the spectrum is $\rho = 1$, referred to as *Denton benchmarking*, where period to period movement preservation is maximized, which results in the smoothest possible set of benchmarking adjustments available with the function.

The *bias* represents the expected discrepancies between the benchmarks and the indicator series. It can be used to pre-adjust the indicator series in order to reduce, on average, the discrepancies between the two sources of data. Bias correction, which is specified with arguments `biasOption` and `bias`, can be particularly useful for periods not covered by benchmarks when $\rho < 1$. In this context, parameter ρ dictates the speed at which the projected benchmarking adjustments converge to the bias (or converge to *no adjustment* without bias correction) for periods not covered

by a benchmark. The smaller the value of ρ, the faster the convergence to the bias, with immediate convergence when $\rho = 0$ and no convergence at all (the adjustment of the last period covered by a benchmark is repeated) when $\rho = 1$ (Denton benchmarking). Arguments `biasOption` and `bias` are actually ignored when $\rho = 1$ since correcting for the bias has no impact on Denton benchmarking solutions. The suggested value for ρ is 0.9 for monthly indicators and $0.9^3 = 0.729$ for quarterly indicators, representing a reasonable compromise between maximizing movement preservation and reducing revisions as new benchmarks become available in the future (benchmarking *timeliness issue*). In practice, note that Denton benchmarking could be *approximated* with the regression-based model by using a ρ value that is smaller than, but very close to, 1.0 (e.g., $\rho = 0.999$). See Dagum and Cholette (2006) for a complete discussion on this topic.

Alterability Coefficients:

Alterability coefficients $c_{s^\dagger}$ and c_a conceptually represent the measurement errors associated with the (bias corrected) indicator time series values $s^\dagger$ and benchmarks a respectively. They are nonnegative real numbers which, in practice, specify the extent to which an initial value can be modified in relation to other values. Alterability coefficients of 0.0 define fixed (binding) values while alterability coefficients greater than 0.0 define free (nonbinding) values. Increasing the alterability coefficient of an intial value results in more changes for that value in the benchmarking solution and, conversely, less changes when decreasing the alterability coefficient. The default alterability coefficients are 0.0 for the benchmarks (binding benchmarks) and 1.0 for the indicator series values (nonbinding indicator series). Important notes:

- With a value of $\rho = 1$ (argument `rho = 1`, associated to Denton Benchmarking), only the default alterability coefficients (0.0 for a benchmark and 1.0 for a an indicator series value) are valid. The specification of user-defined alterability coefficients variables is therefore not allowed. If such variables are specified (see arguments `var` and `with`), the function ignores them and displays a warning message in the console.
- Alterability coefficients $c_{s^\dagger}$ come into play after the indicator series has been corrected for the bias, when applicable ($c_{s^\dagger}$ is associated to $s^\dagger$, not s). This means that specifying an alterability coefficient of 0.0 for a given indicator series value **will not** result in an unchanged value after benchmarking **with bias correction** (see arguments `biasOption` and `bias`).

Nonbinding benchmarks, when applicable, can be recovered (calculated) from the benchmarked series (see output data frame `series` in section **Value**). The output `benchmarks` data frame always contains the original benchmarks provided in the input benchmarks data frame (argument `benchmarks_df`).

Benchmarking Multiple Series:

Multiple series can be benchmarked in a single `benchmarking()` call, by specifying `allCols = TRUE`, by (manually) specifying multiple variables with argument `var` (and argument `with`) or with BY-group processing (argument `by != NULL`). An important distinction is that all indicator series specified with `allCols = TRUE` or with argument `var` (and benchmarks with argument `with`) are expected to be of the same length, i.e., same set of periods and same set (number) of benchmarks. Benchmarking series of different lengths (different sets of periods) or with different sets (number) of benchmarks must be done with BY-group processing on stacked indicator series and benchmarks input data frames (see utility functions `stack_tsDF()` and `stack_bmkDF()`). Arguments `by` and `var` can be combined in order to implement BY-group processing for multiple series as illustrated by *Example 2* in the **Examples** section. While multiple variables with argument `var` (or `allCols = TRUE`) without BY-group processing (argument `by = NULL`) is slightly more efficient (faster), a BY-group approach with a single series variable is usually recommended as it is more general (works in all contexts). The latter is illustrated by *Example 3* in the **Examples** section. The BY variables specified with argument `by` appear in all three output data frames.

Arguments `constant` and `negInput_option`:

These arguments extend the usage of proportional benchmarking to a larger set of problems. Their default values correspond to the G-Series 2.0 behaviour (SAS® PROC BENCHMARKING) for which equivalent options are not defined. Although proportional benchmarking may not necessarily be the most appropriate approach (additive benchmarking may be more appropriate) when the values of the indicator series approach 0 (unstable period-to-period ratios) or "cross the 0 line" and can therefore go from positive to negative and vice versa (confusing, difficult to interpret period-to-period ratios), these cases are not invalid mathematically speaking (i.e., the associated proportional benchmarking problem can be solved). It is strongly recommended, however, to carefully analyze and validate the resulting benchmarked data in these situations and make sure they correspond to reasonable, interpretable solutions.

Treatment of Missing (`NA`) Values:

- If a missing value appears in one of the variables of the benchmarks input data frame (other than the BY variables), the observations with the missing values are dropped, a warning message is displayed and the function executes.
- If a missing value appears in the `year` and/or `period` variables of the indicator series input data frame and BY variables are specified, the corresponding BY-group is skipped, a warning message is displayed and the function moves on to the next BY-group. If no BY variables are specified, a warning message is displayed and no processing is done.
- If a missing value appears in one of the indicator series variables in the indicator series input data frame and BY variables are specified, the corresponding BY-group is skipped, a warning message is displayed and the function moves on to the next BY-group. If no BY variables are specified, the affected indicator series is not processed, a warning message is displayed and the function moves on to the next indicator series (when applicable).

Value

The function returns is a list of three data frames:

- `series`: data frame containing the benchmarked data (primary function output). BY variables specified with argument `by` would be included in the data frame but not alterability coefficient variables specified with argument `var`.
- `benchmarks`: copy of the input benchmarks data frame (excluding invalid benchmarks when applicable). BY variables specified with argument `by` would be included in the data frame but not alterability coefficient variables specified with argument `with`.
- `graphTable`: data frame containing supplementary data useful for producing analytical tables and graphs (see function `plot_graphTable()`). It contains the following variables in addition to the BY variables specified with argument `by`:
  - `varSeries`: Name of the indicator series variable
  - `varBenchmarks`: Name of the benchmark variable
  - `altSeries`: Name of the user-defined indicator series alterability coefficients variable
  - `altSeriesValue`: Indicator series alterability coefficients
  - `altbenchmarks`: Name of the user-defined benchmark alterability coefficients variable
  - `altBenchmarksValue`: Benchmark alterability coefficients
  - `t`: Indicator series period identifier (1 to T)
  - `m`: Benchmark coverage periods identifier (1 to M)
  - `year`: Data point calendar year
  - `period`: Data point period (cycle) value (1 to `periodicity`)
  - `constant`: Temporary additive constant (argument `constant`)
  - `rho`: Autoregressive parameter ρ (argument `rho`)

- `lambda`: Adjustment model parameter λ (argument `lambda`)
- `bias`: Bias adjustment (default, user-defined or estimated bias according to arguments `biasOption` and `bias`)
- `periodicity`: The maximum number of periods in a year (e.g. 4 for a quarterly indicator series)
- `date`: Character string combining the values of variables `year` and `period`
- `subAnnual`: Indicator series values
- `benchmarked`: Benchmarked series values
- `avgBenchmark`: Benchmark values divided by the number of coverage periods
- `avgSubAnnual`: Indicator series values (variable `subAnnual`) averaged over the benchmark coverage period
- `subAnnualCorrected`: Bias corrected indicator series values
- `benchmarkedSubAnnualRatio`: Difference ($\lambda = 0$) or ratio ($\lambda \neq 0$) of the values of variables `benchmarked` and `subAnnual`
- `avgBenchmarkSubAnnualRatio`: Difference ($\lambda = 0$) or ratio ($\lambda \neq 0$) of the values of variables `avgBenchmark` and `avgSubAnnual`
- `growthRateSubAnnual`: Period to period difference ($\lambda = 0$) or relative difference ($\lambda \neq 0$) of the indicator series values (variable `subAnnual`)
- `growthRateBenchmarked`: Period to period difference ($\lambda = 0$) or relative difference ($\lambda \neq 0$) of the benchmarked series values (variable `benchmarked`)

Notes:

- The output `benchmarks` data frame always contains the original benchmarks provided in the input benchmarks data frame. Modified nonbinding benchmarks, when applicable, can be recovered (calculated) from the output `series` data frame.
- The function returns a `NULL` object if an error occurs before data processing could start. Otherwise, if execution gets far enough so that data processing could start, then an incomplete object would be returned in case of errors (e.g., output `series` data frame with `NA` values for the benchmarked data).
- The function returns "data.frame" objects that can be explicitly coerced to other types of objects with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce any of them to a tibble).

References

Dagum, E. B. and P. Cholette (2006). **Benchmarking, Temporal Distribution and Reconciliation Methods of Time Series**. Springer-Verlag, New York, Lecture Notes in Statistics, Vol. 186

Fortier, S. and B. Quenneville (2007). "Theory and Application of Benchmarking in Business Surveys". **Proceedings of the Third International Conference on Establishment Surveys (ICES-III)**. Montréal, June 2007.

Latendresse, E., M. Djona and S. Fortier (2007). "Benchmarking Sub-Annual Series to Annual Totals – From Concepts to SAS® Procedure and Enterprise Guide® Custom Task". **Proceedings of the SAS® Global Forum 2007 Conference**. Cary, NC: SAS Institute Inc.

Quenneville, B., S. Fortier, Z.-G. Chen and E. Latendresse (2006). "Recent Developments in Benchmarking to Annual Totals in X-12-ARIMA and at Statistics Canada". **Proceedings of the Eurostat Conference on Seasonality, Seasonal Adjustment and Their Implications for Short-Term Analysis and Forecasting**. Luxembourg, May 2006.

Quenneville, B., P. Cholette, S. Fortier and J. Bérubé (2010). "Benchmarking Sub-Annual Indicator Series to Annual Control Totals (Forillon v1.04.001)". **Internal document**. Statistics Canada, Ottawa, Canada.

Quenneville, B. and S. Fortier (2012). "Restoring Accounting Constraints in Time Series – Methods and Software for a Statistical Agency". **Economic Time Series: Modeling and Seasonality**. Chapman & Hall, New York.

Statistics Canada (2012). **Theory and Application of Benchmarking (Course code 0436)**. Statistics Canada, Ottawa, Canada.

Statistics Canada (2016). "The BENCHMARKING Procedure". **G-Series 2.0 User Guide**. Statistics Canada, Ottawa, Canada.

See Also

stock_benchmarking() plot_graphTable() bench_graphs plot_benchAdj() gs.gInv_MP() aliases

Examples

```
# Set the working directory (for the PDF files)
iniwd <- getwd()
setwd(tempdir())


###########
# Example 1: Simple case with a single quarterly series to benchmark to annual values

# Quarterly indicator series
my_series1 <- ts_to_tsDF(ts(c(1.9, 2.4, 3.1, 2.2, 2.0, 2.6, 3.4, 2.4, 2.3),
                            start = c(2015, 1),
                            frequency = 4))
my_series1

# Annual benchmarks for quarterly data
my_benchmarks1 <- ts_to_bmkDF(ts(c(10.3, 10.2),
                                 start = 2015,
                                 frequency = 1),
                              ind_frequency = 4)
my_benchmarks1

# Benchmarking using...
#   - recommended `rho` value for quarterly series (`rho = 0.729`)
#   - proportional model (`lambda = 1`)
#   - bias-corrected indicator series with the estimated bias (`biasOption = 3`)
out_bench1 <- benchmarking(my_series1,
                           my_benchmarks1,
                           rho = 0.729,
                           lambda = 0,
                           biasOption = 3)

# Generate the benchmarking graphs
plot_graphTable(out_bench1$graphTable, "Ex1_graphs.pdf")


###########
# Example 2: Two quarterly series to benchmark to annual values,
#            with BY-groups and user-defined alterability coefficients

# Sales data (same sales for groups A and B; only alter coefs for van sales differ)
qtr_sales <- ts(matrix(c(# Car sales
                         1851, 2436, 3115, 2205, 1987, 2635, 3435, 2361, 2183, 2822,
```

```
                        3664, 2550, 2342, 3001, 3779, 2538, 2363, 3090, 3807, 2631,
                        2601, 3063, 3961, 2774, 2476, 3083, 3864, 2773, 2489, 3082,
                        # Van sales
                        1900, 2200, 3000, 2000, 1900, 2500, 3800, 2500, 2100, 3100,
                        3650, 2950, 3300, 4000, 3290, 2600, 2010, 3600, 3500, 2100,
                        2050, 3500, 4290, 2800, 2770, 3080, 3100, 2800, 3100, 2860),
                      ncol = 2),
                start = c(2011, 1),
                frequency = 4,
                names = c("car_sales", "van_sales"))

ann_sales <- ts(matrix(c(# Car sales
                         10324, 10200, 10582, 11097, 11582, 11092,
                         # Van sales
                         12000, 10400, 11550, 11400, 14500, 16000),
                       ncol = 2),
                start = 2011,
                frequency = 1,
                names = c("car_sales", "van_sales"))

# Quarterly indicator series (with default alter coefs for now)
my_series2 <- rbind(cbind(data.frame(group = rep("A", nrow(qtr_sales)),
                                     alt_van = rep(1, nrow(qtr_sales))),
                          ts_to_tsDF(qtr_sales)),
                    cbind(data.frame(group = rep("B", nrow(qtr_sales)),
                                     alt_van = rep(1, nrow(qtr_sales))),
                          ts_to_tsDF(qtr_sales)))

# Set binding van sales (alter coef = 0) for 2012 Q1 and Q2 in group A (rows 5 and 6)
my_series2$alt_van[c(5,6)] <- 0
head(my_series2, n = 10)
tail(my_series2)


# Annual benchmarks for quarterly data (without alter coefs)
my_benchmarks2 <- rbind(cbind(data.frame(group = rep("A", nrow(ann_sales))),
                              ts_to_bmkDF(ann_sales, ind_frequency = 4)),
                        cbind(data.frame(group = rep("B", nrow(ann_sales))),
                              ts_to_bmkDF(ann_sales, ind_frequency = 4)))
my_benchmarks2

# Benchmarking using...
#   - recommended `rho` value for quarterly series (`rho = 0.729`)
#   - proportional model (`lambda = 1`)
#   - without bias correction (`biasOption = 1` and `bias` not specified)
#   - `quiet = TRUE` to avoid generating the function header
out_bench2 <- benchmarking(my_series2,
                           my_benchmarks2,
                           rho = 0.729,
                           lambda = 1,
                           biasOption = 1,
                           var = c("car_sales", "van_sales / alt_van"),
                           with = c("car_sales", "van_sales"),
                           by = "group",
                           quiet = TRUE)

# Generate the benchmarking graphs
```

```
plot_graphTable(out_bench2$graphTable, "Ex2_graphs.pdf")

# Check the value of van sales for 2012 Q1 and Q2 in group A (fixed values)
all.equal(my_series2$van_sales[c(5,6)], out_bench2$series$van_sales[c(5,6)])


###########
# Example 3: same as example 2, but benchmarking all 4 series as BY-groups
#            (4 BY-groups of 1 series instead of 2 BY-groups of 2 series)

qtr_sales2 <- ts.union(A = qtr_sales, B = qtr_sales)
my_series3 <- stack_tsDF(ts_to_tsDF(qtr_sales2))
my_series3$alter <- 1
my_series3$alter[my_series3$series == "A.van_sales"
                & my_series3$year == 2012 & my_series3$period <= 2] <- 0
head(my_series3)
tail(my_series3)


ann_sales2 <- ts.union(A = ann_sales, B = ann_sales)
my_benchmarks3 <- stack_bmkDF(ts_to_bmkDF(ann_sales2, ind_frequency = 4))
head(my_benchmarks3)
tail(my_benchmarks3)

out_bench3 <- benchmarking(my_series3,
                           my_benchmarks3,
                           rho = 0.729,
                           lambda = 1,
                           biasOption = 1,
                           var = "value / alter",
                           with = "value",
                           by = "series",
                           quiet = TRUE)

# Generate the benchmarking graphs
plot_graphTable(out_bench3$graphTable, "Ex3_graphs.pdf")

# Convert data frame `out_bench3$series` to a "mts" object
qtr_sales2_bmked <- tsDF_to_ts(unstack_tsDF(out_bench3$series), frequency = 4)

# Print the first 10 observations
ts(qtr_sales2_bmked[1:10, ], start = start(qtr_sales2), deltat = deltat(qtr_sales2))

# Check the value of van sales for 2012 Q1 and Q2 in group A (fixed values)
all.equal(window(qtr_sales2[, "A.van_sales"], start = c(2012, 1), end = c(2012, 2)),
         window(qtr_sales2_bmked[, "A.van_sales"], start = c(2012, 1), end = c(2012, 2)))


# Reset the working directory to its initial location
setwd(iniwd)
```

---

bench_graphs *Generate a benchmarking graphic*

Description

Functions used internally by `plot_graphTable()` to generate the benchmarking graphics in a PDF file:

- `ori_plot()`: Original Scale Plot (`plot_graphTable()` argument `ori_plot_flag = TRUE`)
- `adj_plot()`: Adjustment Scale Plot (`plot_graphTable()` argument `adj_plot_flag = TRUE`)
- `GR_plot()`: Growth Rates Plot (`plot_graphTable()` argument `GR_plot_flag = TRUE`)
- `GR_table()`: Growth Rates Table (`plot_graphTable()` argument `GR_table_flag = TRUE`)

When these functions are called directly, the `graphTable` data frame should only contain a **single series** and the graphic is generated in the current (active) graphics device.

Usage

```
ori_plot(
  graphTable,
  title_str = "Original Scale",
  subtitle_str = NULL,
  mth_gap = NULL,
  points_set = NULL,
  pt_sz = 2,
  display_ggplot = TRUE,
  .setup = TRUE
)

adj_plot(
  graphTable,
  title_str = "Adjustment Scale",
  subtitle_str = NULL,
  mth_gap = NULL,
  full_set = NULL,
  pt_sz = 2,
  display_ggplot = TRUE,
  .setup = TRUE
)

GR_plot(
  graphTable,
  title_str = "Growth Rates",
  subtitle_str = NULL,
  factor = NULL,
  type_chars = NULL,
  periodicity = NULL,
  display_ggplot = TRUE,
  .setup = TRUE
)

GR_table(
  graphTable,
  title_str = "Growth Rates Table",
  subtitle_str = NULL,
  factor = NULL,
```

```
  type_chars = NULL,
  display_ggplot = TRUE,
  .setup = TRUE
)
```

Arguments

`graphTable`	(mandatory) Data frame (object of class "data.frame") corresponding to the benchmarking function output`graphTable` data frame.
`title_str`, `subtitle_str`	(optional) Graphic title and subtitle strings (character constants). `subtitle_str` is automatically built from the `graphTable` data frame contents when `NULL` and contains the `graphTable` data frame name on the 2nd line and the benchmarking parameters on the 3rd line. Specifying empty strings (`""`) would remove the titles. Simple Markdown and HTML syntax is allowed (e.g., for bold, italic or colored fonts) through package ggtext (see `help(package = "ggtext")`). **Default values** are `subtitle_str = NULL` and a function specific string for `title_str` (see **Usage**).
`mth_gap`	(optional) Number of months between consecutive periods (e.g. 1 for monthly data, 3 for quarterly data, etc.). Based on the `graphTable` data frame contents when `NULL` (calculated as `12 / graphTable$periodicity[1]`). **Default value** is `mth_gap = NULL`.
`points_set`, `full_set`	(optional) Character vector of the elements (variables of the `graphTable` data frame) to include in the plot. Automatically built when `NULL`. See `plot_graphTable()` for the (default) list of variables used for each type of graphic. **Default values** are `points_set = NULL` and `full_set = NULL`.
`pt_sz`	(optional) Size of the data points shape (symbol) for ggplot2. **Default value** is `pt_sz = 2`.
`display_ggplot`	(optional) Logical arguments indicating whether or not the ggplot object(s) should be displayed in the current (active) graphics device. **Default value** is `display_ggplot = TRUE`.
`.setup`	(optional) Logical argument indicating whether the setup steps must be executed or not. Must be `TRUE` when the function is called directly (i.e., outside of the `plot_graphTable()` context). **Default value** is `.setup = TRUE`.
`factor`, `type_chars`	(optional) Growth rates factor (1 or 100) and value label suffix ("" or "(%)") according to the adjustment model parameter λ. Based on the `graphTable` data frame contents when `NULL` (based on `graphTable$lambda[1]`). **Default values** are `factor = NULL` and `type_chars = NULL`.

`periodicity` (optional)
Number of periods in a year. Based on the `graphTable` data frame contents when `NULL` (defined as `graphTable$periodicity[1]`).
Default value is `periodicity = NULL`.

Details

See `plot_graphTable()` for a detailed description of the four benchmarking graphics associated to these individual functions. These graphics are optimized for the US Letter paper size format in landscape view, i.e., 11in wide (27.9cm, 1056px with 96 DPI) and 8.5in tall (21.6cm, 816px with 96 DPI). Keep this in mind when viewing or saving graphics generated by calls to these individual functions (i.e., outside of the `plot_graphTable()` context). Also note that `GR_plot()` and `GR_table()` will often generate more than one graphic (more than one *page*), unless the number of periods included in the input `graphTable` data frame is reduced (e.g., subsetting the data frame by ranges of calendar years).

Value

In addition to displaying the corresponding graphic(s) in the current (active) graphics device (except when `display_ggplot = FALSE`), each function also invisibly returns a list containing the generated ggplot object(s). Notes:

- `ori_plot()` and `adj_plot()` generate a single ggplot object (single graphic) while `GR_plot()` and `GR_table()` will often generate several ggplot objects (several graphics).
- The returned ggplot object(s) can be displayed *manually* with `print()`, in which case the following ggplot2 theme updates (used internally when `display_ggplot = TRUE`) are suggested:

```
ggplot2::theme_update(
  plot.title = ggtext::element_markdown(hjust = 0.5),
  plot.subtitle = ggtext::element_markdown(hjust = 0.5),
  legend.position = "bottom",
 plot.margin = ggplot2::margin(t = 1.5, r = 1.5, b = 1.5, l = 1.5, unit = "cm"))
```

See Also

`plot_graphTable()` `plot_benchAdj()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Deactivate the graphics device creation for the pkgdown website HTML reference page
# (irrelevant in that context)
new_grDev <- !(identical(Sys.getenv("IN_PKGDOWN"), "true"))


# Initial quarterly time series (indicator series to be benchmarked)
qtr_ts <- ts(c(1.9, 2.4, 3.1, 2.2, 2.0, 2.6, 3.4, 2.4, 2.3),
             start = c(2015, 1), frequency = 4)

# Annual time series (benchmarks)
ann_ts <- ts(c(10.3, 10.2), start = 2015, frequency = 1)

# Proportional benchmarking
out_bench <- benchmarking(ts_to_tsDF(qtr_ts),
                          ts_to_bmkDF(ann_ts, ind_frequency = 4),
                          rho = 0.729, lambda = 1, biasOption = 3,
```

```
                          quiet = TRUE)


# Open a new graphics device that is 11in wide and 8.5in tall
# (US Letter paper size format in landscape view)
if (new_grDev) {
  dev.new(width = 11, height = 8.5, unit = "in", noRStudioGD = TRUE)
}

# Generate the benchmarking graphics
ori_plot(out_bench$graphTable)
adj_plot(out_bench$graphTable)
GR_plot(out_bench$graphTable)
GR_table(out_bench$graphTable)


# Simulate multiple series benchmarking (3 series)

qtr_mts <- ts.union(ser1 = qtr_ts, ser2 = qtr_ts * 100, ser3 = qtr_ts * 10)
ann_mts <- ts.union(ser1 = ann_ts, ser2 = ann_ts * 100, ser3 = ann_ts * 10)

# Using argument `allCols = TRUE` (identify series with column `varSeries`)
out_bench2 <- benchmarking(ts_to_tsDF(qtr_mts),
                           ts_to_bmkDF(ann_mts, ind_frequency = 4),
                           rho = 0.729, lambda = 1, biasOption = 3,
                           allCols = TRUE,
                           quiet = TRUE)

# Original and adjustment scale plots for the 2nd series (ser2)
ser2_res <- out_bench2$graphTable[out_bench2$graphTable$varSeries == "ser2", ]
ori_plot(ser2_res)
adj_plot(ser2_res)

# Using argument `by = "series"` (identify series with column `series`)
out_bench3 <- benchmarking(stack_tsDF(ts_to_tsDF(qtr_mts)),
                           stack_bmkDF(ts_to_bmkDF(ann_mts, ind_frequency = 4)),
                           rho = 0.729, lambda = 1, biasOption = 3,
                           by = "series",
                           quiet = TRUE)

# Growth rates plot for the 3rd series (ser3)
ser3_res <- out_bench3$graphTable[out_bench3$graphTable$series == "ser3", ]
GR_plot(ser3_res)


# Close the graphics device
if (new_grDev) {
  dev.off()
}
```

---

build_balancing_problem

Build the elements of balancing problems.

Description

This function is used internally by `tsbalancing()` to build the elements of the balancing problems. It can also be useful to derive the indirect series associated to equality balancing constraints manually (outside of the `tsbalancing()` context).

Usage

```
build_balancing_problem(
  in_ts,
  problem_specs_df,
  in_ts_name = deparse1(substitute(in_ts)),
  ts_freq = stats::frequency(in_ts),
  periods = gs.time2str(in_ts),
  n_per = nrow(as.matrix(in_ts)),
  specs_df_name = deparse1(substitute(problem_specs_df)),
  temporal_grp_periodicity = 1,
  alter_pos = 1,
  alter_neg = 1,
  alter_mix = 1,
  lower_bound = -Inf,
  upper_bound = Inf,
  validation_only = FALSE
)
```

Arguments

`in_ts` (mandatory)

Time series (object of class "ts" or "mts") that contains the time series data to be reconciled. They are the balancing problems' input data (initial solutions).

`problem_specs_df` (mandatory)

Balancing problem specifications data frame. Using a sparse format inspired from the SAS/OR® LP procedure's *sparse data input format* (SAS Institute 2015), it contains only the relevant information such as the nonzero coefficients of the balancing constraints as well as the non-default alterability coefficients and lower/upper bounds (i.e., values that would take precedence over those defined with arguments `alter_pos`, `alter_neg`, `alter_mix`, `alter_temporal`, `lower_bound` and `upper_bound`).

The information is provided using four mandatory variables (`type`, `col`, `row` and `coef`) and one optional variable (`timeVal`). An observation (a row) in the problem specs data frame either defines a label for one of the seven types of the balancing problem elements with columns `type` and `row` (see *Label definition records* below) or specifies coefficients (numerical values) for those balancing problem elements with variables `col`, `row`, `coef` and `timeVal` (see *Information specification records* below).

- **Label definition records** (`type` is not missing (is not `NA`))
  - `type` (chr): reserved keyword identifying the type of problem element being defined:
    - `EQ`: equality ($=$) balancing constraint
    - `LE`: lower or equal ($\leq$) balancing constraint
    - `GE`: greater or equal ($\geq$) balancing constraint

    - lowerBd: period value lower bound
    - upperBd: period value upper bound
    - alter: period values alterability coefficient
    - alterTmp: temporal total alterability coefficient
  - row (chr): label to be associated to the problem element (type *keyword*)
  - *all other variables are irrelevant and should contain missing data (*NA *values)*

- **Information specification records** (type is missing (is NA))
  - type (chr): not applicable (NA)
  - col (chr): series name or reserved word _rhs_ to specify a balancing constraint right-hand side (RHS) value.
  - row (chr): problem element label.
  - coef (num): problem element value:
    - balancing constraint series coefficient or RHS value
    - series period value lower or upper bound
    - series period value or temporal total alterability coefficient
  - timeVal (num): optional time value to restrict the application of series bounds or alterability coefficients to a specific time period (or temporal group). It corresponds to the time value, as returned by stats::time(), of a given input time series (argument in_ts) period (observation) and is conceptually equivalent to $year + (period - 1)/frequency$.

Note that empty strings ("" or '') for character variables are interpreted as missing (NA) by the function. Variable row identifies the elements of the balancing problem and is the key variable that makes the link between both types of records. The same label (row) cannot be associated with more than one type of problem element (type) and multiple labels (row) cannot be defined for the same given type of problem element (type), except for balancing constraints (values "EQ", "LE" and "GE" of column type). User-friendly features of the problem specs data frame include:

- The order of the observations (rows) is not important.
- Character values (variables type, row and col) are not case sensitive (e.g., strings "Constraint 1" and "CONSTRAINT 1" for row would be considered as the same problem element label), except when col is used to specify a series name (a column of the input time series object) where **case sensitivity is enforced**.
- The variable names of the problem specs data frame are also not case sensitive (e.g., type, Type or TYPE are all valid) and time_val is an accepted variable name (instead of timeVal).

Finally, the following table lists valid aliases for the type *keywords* (type of problem element):

Keyword	Aliases
EQ	==, =
LE	<=, <
GE	>=, >
lowerBd	lowerBound, lowerBnd, + *same terms with '_', '.' or ' ' between words*
upperBd	upperBound, upperBnd, + *same terms with '_', '.' or ' ' between words*

alterTmp alterTemporal, alterTemp, + *same terms with '_', '.' or ' ' between words*

Reviewing the **Examples** should help conceptualize the balancing problem specifications data frame.

in_ts_name (optional)

String containing the value of argument in_ts.

Default value is in_ts_name = deparse1(substitute(in_ts)).

ts_freq (optional)

Frequency of the time series object (argument in_ts).

Default value is ts_freq = stats::frequency(in_ts).

periods (optional)

Character vector describing the time series object (argument in_ts) periods.

Default value is periods = gs.time2str(in_ts).

n_per (optional)

Number of periods of the time series object (argument in_ts).

Default value is n_per = nrow(as.matrix(in_ts)).

specs_df_name (optional)

String containing the value of argument problem_specs_df.

Default value is specs_df_name = deparse1(substitute(problem_specs_df)).

temporal_grp_periodicity

(optional)

Positive integer defining the number of periods in temporal groups for which the totals should be preserved. E.g., specify temporal_grp_periodicity = 3 with a monthly time series for quarterly total preservation and temporal_grp_periodicity = 12 (or temporal_grp_periodicity = frequency(in_ts)) for annual total preservation. Specifying temporal_grp_periodicity = 1 (*default*) corresponds to period-by-period processing without temporal total preservation.

Default value is temporal_grp_periodicity = 1 (period-by-period processing without temporal total preservation).

alter_pos (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with **positive** coefficients in all balancing constraints in which they are involved (e.g., component series in aggregation table raking problems). Alterability coefficients provided in the problem specification data frame (argument problem_specs_df) override this value.

Default value is alter_pos = 1.0 (nonbinding values).

alter_neg (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with **negative** coefficients in all balancing constraints in which they are involved (e.g., marginal totals in aggregation table raking problems). Alterability coefficients provided in the problem specification data frame (argument problem_specs_df) override this value.

Default value is alter_neg = 1.0 (nonbinding values).

alter_mix (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with a mix of **positive and negative** coefficients

in the balancing constraints in which they are involved. Alterability coefficients provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `alter_mix = 1.0` (nonbinding values).

`lower_bound` (optional)

Real number specifying the default lower bound for the time series values. Lower bounds provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `lower_bound = -Inf` (unbounded).

`upper_bound` (optional)

Real number specifying the default upper bound for the time series values. Upper bounds provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `upper_bound = Inf` (unbounded).

`validation_only`

(optional)

Logical argument specifying whether the function should only perform input data validation or not. When `validation_only = TRUE`, the specified *balancing constraints* and *period value (lower and upper) bounds* constraints are validated against the input time series data, allowing for discrepancies up to the value specified with argument `validation_tol`. Otherwise, when `validation_only = FALSE` (default), the input data are first reconciled and the resulting (output) data are then validated.

Default value is `validation_only = FALSE`.

Details

See `tsbalancing()` for a detailed description of *time series balancing* problems.

Any missing (`NA`) value found in the input time series object (argument `in_ts`) would be replaced with 0 in `values_ts` and trigger a warning message.

The returned elements of the balancing problems do not include the implicit temporal totals (i.e., elements `A2`, `op2` and `b2` only contain the balancing constraints).

Multi-period balancing problem elements `A2`, `op2` and `b2` (when `temporal_grp_periodicity > 1`) are constructed *column by column* (in "column-major order"), corresponding to the default behaviour of R for converting objects of class "matrix" into vectors. I.e., the balancing constraints conceptually correspond to:

- `A1 %*% values_ts[t, ] op1 b1` for problems involving a single period (`t`)
- `A2 %*% as.vector(values_ts[t1:t2, ]) op2 b2` for problems involving `temporal_grp_periodicity` periods (`t1:t2`).

Note that argument `alter_temporal` has not been applied yet at this point and `altertmp$coefs_ts` only contains the coefficients specified in the problem specs data frame (argument `problem_specs_df`). I.e., `altertmp$coefs_ts` contains missing (`NA`) values except for the temporal total alterability coefficients included in (specified with) `problem_specs_df`. This is done in order to simplify the identification of the first non missing (non `NA`) temporal total alterability coefficient of each complete temporal group (to occur later, when applicable, inside `tsbalancing()`).

Value

A list with the elements of the balancing problems (excluding the temporal totals info):

- `labels_df`: cleaned-up version of the *label definition records* from `problem_specs_df` (`type` is not missing (is not `NA`)); extra columns:
  - `type.lc` : `tolower(type)`
  - `row.lc` : `tolower(row)`
  - `con.flag`: `type.lc %in% c("eq", "le", "ge")`
- `coefs_df` : cleaned-up version of the information specification records from `problem_specs_df` (`type` is missing (is `NA`); extra columns:
  - `row.lc` : `tolower(row)`
  - `con.flag`: `labels_df$con.flag` allocated through `row.lc`
- `values_ts`: reduced version of `in_ts` with only the relevant series (see vector `ser_names`)
- `lb` : lower bound info (`type.lc = "lowerbd"`) for the relevant series; list object with the following elements:
  - `coefs_ts` : lower bound values for series and period
  - `nondated_coefs` : vector of nondated lower bounds from `problem_specs_df` (`timeVal` is `NA`)
  - `nondated_id_vec`: vector of `ser_names` id's associated to vector `nondated_coefs`
  - `dated_id_vec` : vector of `ser_names` id's associated to dated lower bounds from `problem_specs_df` (`timeVal` is not `NA`)
- `ub` : `lb` equivalent for upper bounds (`type.lc = "upperbd"`)
- `alter` : `lb` equivalent for period value alterability coefficients (`type.lc = "alter"`)
- `altertmp` : `lb` equivalent for temporal total alterability coefficients (`type.lc = "altertmp"`)
- `ser_names`: vector of the relevant series names (set of series involved in the balancing constraints)
- `pos_ser` : vector of series names that have only positive nonzero coefficients across all balancing constraints
- `neg_ser` : vector of series names that have only negative nonzero coefficients across all balancing constraints
- `mix_ser` : vector of series names that have both positive and negative nonzero coefficients across all balancing constraints
- `A1`,`op1`,`b1`: balancing constraint elements for problems involving a single period (e.g., each period of an incomplete temporal group)
- `A2`,`op2`,`b2`: balancing constraint elements for problems involving `temporal_grp_periodicity` periods (e.g., the set of periods of a complete temporal group)

See Also

`tsbalancing()` `build_raking_problem()`

Examples

```
################################################################################
#        Indirect series derivation framework with `tsbalancing()` metadata
################################################################################
#
# Is is assumed (agreed) that...
```

```
#
# a) All balancing constraints are equality constraints (`type = EQ`).
# b) All constraints have only one nonbinding (free) series: the series to be derived
#    (i.e., all series have an alter. coef of 0 except the series to be derived).
# c) Each constraint derives a different (new) series.
# d) Constraints are the same for all periods (i.e., no "dated" alter. coefs
#    specified with column `timeVal`).
##############################################################################################


# Derive the 5 marginal totals of a 2 x 3 two-dimensional data cube using `tsbalancing()`
# metadata (data cube aggregation constraints respect the above assumptions).


# Build the balancing problem specs through the (simpler) raking metadata.
my_specs <- rkMeta_to_blSpecs(
  data.frame(series = c("A1", "A2", "A3",
                        "B1", "B2", "B3"),
             total1 = c(rep("totA", 3),
                        rep("totB", 3)),
             total2 = rep(c("tot1", "tot2", "tot3"), 2)),
  alterSeries = 0,  # binding (fixed) component series
  alterTotal1 = 1,  # nonbinding (free) marginal totals (to be derived)
  alterTotal2 = 1)  # nonbinding (free) marginal totals (to be derived)
my_specs

# 6 periods (quarters) of data with marginal totals set to zero (0): they MUST exist
# in the input data AND contain valid (non missing) data.
my_ts <- ts(data.frame(A1 = c(12, 10, 12,  9, 15,  7),
                       B1 = c(20, 21, 15, 17, 19, 18),
                       A2 = c(14,  9,  8,  9, 11, 10),
                       B2 = c(20, 29, 20, 24, 21, 17),
                       A3 = c(13, 15, 17, 14, 16, 12),
                       B3 = c(24, 20, 30, 23, 21, 19),
                       tot1 = rep(0, 6),
                       tot2 = rep(0, 6),
                       tot3 = rep(0, 6),
                       totA = rep(0, 6),
                       totB = rep(0, 6)),
            start = 2019, frequency = 4)

# Get the balancing problem elements.
n_per <- nrow(my_ts)
p <- build_balancing_problem(my_ts, my_specs,
                             temporal_grp_periodicity = n_per)

# `A2`, `op2` and `b2` define 30 constraints (5 marginal totals X 6 periods)
# involving a total of 66 time series data points (11 series X 6 periods) of which
# 36 belong to the 6 component series and 30 belong to the 5 marginal totals.
dim(p$A2)

# Get the names of the marginal totals (series with a nonzero alter. coef), in the order
# in which the corresponding constraints appear in the specs (constraints specification
# order).
tmp <- p$coefs_df$col[p$coefs_df$con.flag]
tot_names <- tmp[tmp %in% p$ser_names[p$alter$nondated_id_vec[p$alter$nondated_coefs != 0]]]
```

```
# Define logical flags identifying the marginal total columns:
# - `tot_col_logi1`: for single-period elements (of length 11 = number of series)
# - `tot_col_logi2`: for multi-period elements (of length 66 = number of data points),
#                    in "column-major order" (the `A2` matrix element construction order)
tot_col_logi1 <- p$ser_names %in% tot_names
tot_col_logi2 <- rep(tot_col_logi1, each = n_per)

# Order of the marginal totals to be derived based on
# ... the input data columns ("mts" object `my_ts`)
p$ser_names[tot_col_logi1]
# ... the constraints specification (data frame `my_specs`)
tot_names


# Calculate the 5 marginal totals for all 6 periods
# Note: the following calculation allows for general linear equality constraints, i.e.,
#       a) nonzero right-hand side (RHS) constraint values (`b2`) and
#       b) nonzero constraint coefs other than 1 for the component series and -1 for
#          the derived series.
my_ts[, tot_names] <- {
  (
    # Constraints RHS.
    p$b2 -

    # Sums of the components ("weighted" by the constraint coefficients).
    p$A2[, !tot_col_logi2, drop = FALSE] %*% as.vector(p$values_ts[, !tot_col_logi1])
  ) /

  # Derived series constraint coefficients: `t()` allows for a "row-major order" search
  # in matrix `A2` (i.e., according to the constraints specification order).
  # Note: `diag(p$A2[, tot_col_logi2])` would work if `p$ser_names[tot_col_logi1]` and
  #       `tot_names` were identical (same totals order); however, the following search
  #       in "row-major order" will always work (and is necessary in the current case).
  t(p$A2[, tot_col_logi2])[t(p$A2[, tot_col_logi2]) != 0]
}
my_ts
```

`build_raking_problem` *Build the elements of raking problems.*

Description

This function is used internally by `tsraking()` to build the elements of the raking problem. It can also be useful to derive the cross-sectional (marginal) totals of the raking problem manually (outside of the `tsraking()` context).

Usage

```
build_raking_problem(
  data_df,
  metadata_df,
  data_df_name = deparse1(substitute(data_df)),
  metadata_df_name = deparse1(substitute(metadata_df)),
  alterability_df = NULL,
```

```
  alterSeries = 1,
  alterTotal1 = 0,
  alterTotal2 = 0
)
```

Arguments

`data_df` (mandatory)

Data frame (object of class "data.frame") that contains the time series data to be reconciled. It must minimally contain variables corresponding to the component series and cross-sectional control totals specified in the metadata data frame (argument `metadata_df`). If more than one observation (period) is provided, the sum of the provided component series values will also be preserved as part of implicit temporal constraints.

`metadata_df` (mandatory)

Data frame (object of class "data.frame") that describes the cross-sectional aggregation constraints (additivity rules) for the raking problem. Two character variables must be included in the metadata data frame: `series` and `total1`. Two variables are optional: `total2` (character) and `alterAnnual` (numeric). The values of variable `series` represent the variable names of the component series in the input time series data frame (argument `data_df`). Similarly, the values of variables `total1` and `total2` represent the variable names of the 1st and 2nd dimension cross-sectional control totals in the input time series data frame. Variable `alterAnnual` contains the alterability coefficient for the temporal constraint associated to each component series. When specified, the latter will override the default alterability coefficient specified with argument `alterAnnual`.

`data_df_name` (optional)

String containing the value of argument `data_df`.

Default value is `data_df_name = deparse1(substitute(data_df))`.

`metadata_df_name`

(optional)

String containing the value of argument `metadata_df`.

Default value is `metadata_df_name = deparse1(substitute(metadata_df))`.

`alterability_df`

(optional)

Data frame (object of class "data.frame"), or `NULL`, that contains the alterability coefficients variables. They must correspond to a component series or a cross-sectional control total, that is, a variable with the same name must exist in the input time series data frame (argument `data_df`). The values of these alterability coefficients will override the default alterability coefficients specified with arguments `alterSeries`, `alterTotal1` and `alterTotal2`. When the input time series data frame contains several observations and the alterability coefficients data frame contains only one, the alterability coefficients are used (repeated) for all observations of the input time series data frame. Alternatively, the alterability coefficients data frame may contain as many observations as the input time series data frame.

Default value is `alterability_df = NULL` (default alterability coefficients).

`alterSeries` (optional)

Nonnegative real number specifying the default alterability coefficient for the component series values. It will apply to component series for which alterability

coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterSeries = 1.0` (nonbinding component series values).

`alterTotal1` (optional)

Nonnegative real number specifying the default alterability coefficient for the 1st dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterTotal1 = 0.0` (binding 1st dimension cross-sectional control totals)

`alterTotal2` (optional)

Nonnegative real number specifying the default alterability coefficient for the 2nd dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterTotal2 = 0.0` (binding 2nd dimension cross-sectional control totals).

Details

See `tsraking()` for a detailed description of *time series raking* problems.

The returned raking problem elements do not include the implicit component series temporal totals when applicable (i.e., elements `g` and `G` only contain the cross-sectional totals info).

When the input data contains multiple periods (temporal total preservation scenario), raking problem elements `x`, `c_x`, `g`, `c_g` and `G` are constructed *column by column* (in "column-major order"), corresponding to the default behaviour of R for converting objects of class "matrix" into vectors.

Value

A list with the elements of the raking problem (excluding the implicit temporal totals):

- `x` : vector of component series initial values
- `c_x` : vector of component series alterability coefficients
- `comp_cols`: vector of component series (column) names
- `g` : vector of cross-sectional total initial values
- `c_g` : vector of cross-sectional total alterability coefficients
- `tot_cols` : vector of cross-sectional total (column) names
- `G` : cross-sectional total aggregation matrix (`g = G %*% x`)

See Also

`tsraking()` `build_balancing_problem()`

Examples

```
# Derive the 5 marginal totals of a 2 x 3 two-dimensional data cube using `tsraking()`
# metadata.

my_metadata <- data.frame(series = c("A1", "A2", "A3",
                                     "B1", "B2", "B3"),
```

```
                          total1 = c(rep("totA", 3),
                                     rep("totB", 3)),
                          total2 = rep(c("tot1", "tot2", "tot3"), 2))
my_metadata

# 6 periods of data with marginal totals set to `NA` (they MUST exist in the input data
# but can be `NA`).
my_data <- data.frame(A1 = c(12, 10, 12,  9, 15,  7),
                      B1 = c(20, 21, 15, 17, 19, 18),
                      A2 = c(14,  9,  8,  9, 11, 10),
                      B2 = c(20, 29, 20, 24, 21, 17),
                      A3 = c(13, 15, 17, 14, 16, 12),
                      B3 = c(24, 20, 30, 23, 21, 19),
                      tot1 = rep(NA, 6),
                      tot2 = rep(NA, 6),
                      tot3 = rep(NA, 6),
                      totA = rep(NA, 6),
                      totB = rep(NA, 6))

# Get the raking problem elements.
p <- build_raking_problem(my_data, my_metadata)
str(p)

# Calculate the 5 marginal totals for all 6 periods.
my_data[p$tot_cols] <- p$G %*% p$x
my_data
```

`gs.build_proc_grps` *Build reconciliation processing groups*

Description

This function builds the processing groups data frame for reconciliation problems. It is used internally by `tsraking_driver()` and `tsbalancing()`.

Usage

```
gs.build_proc_grps(
  ts_yr_vec,
  ts_per_vec,
  n_per,
  ts_freq,
  temporal_grp_periodicity,
  temporal_grp_start
)
```

Arguments

`ts_yr_vec`	(mandatory) Vector of the time series year (time unit) values (see `gs.time2year()`).
`ts_per_vec`	(mandatory) Vector of the time series period (cycle) values (see `gs.time2per()`).

`n_per` (mandatory)
Time series length (number of periods).

`ts_freq` (mandatory)
Time series frequency (see `stats::frequency()`).

`temporal_grp_periodicity` (mandatory)
Number of periods in temporal groups.

`temporal_grp_start` (mandatory)
First period of temporal groups.

Value

A data frame with the following variables (columns):

- `grp` : integer vector identifying the processing group (`1:<number-of-groups>`).
- `beg_per` : integer vector identifying the first period of the processing group.
- `end_per` : integer vector identifying the last period of the processing group.
- `complete_grp`: logical vector indicating if the processing group corresponds to a complete temporal group.

Processing groups

The set of periods of a given reconciliation (raking or balancing) problem is called a *processing group* and either corresponds to:

- a **single period** with period-by-period processing or, when preserving temporal totals, for the individual periods of an incomplete temporal group (e.g., an incomplete year)
- or the **set of periods of a complete temporal group** (e.g., a complete year) when preserving temporal totals.

The total number of processing groups (total number of reconciliation problems) depends on the set of periods in the input time series object (argument `in_ts`) and on the value of arguments `temporal_grp_periodicity` and `temporal_grp_start`.

Common scenarios include `temporal_grp_periodicity = 1` (default) for period-by period processing without temporal total preservation and `temporal_grp_periodicity = frequency(in_ts)` for the preservation of annual totals (calendar years by default). Argument `temporal_grp_start` allows the specification of other types of (*non-calendar*) years. E.g., fiscal years starting on April correspond to `temporal_grp_start = 4` with monthly data and `temporal_grp_start = 2` with quarterly data. Preserving quarterly totals with monthly data would correspond to `temporal_grp_periodicity = 3`.

By default, temporal groups covering more than a year (i.e., corresponding to `temporal_grp_periodicity > frequency(in_ts)` start on a year that is a multiple of `ceiling(temporal_grp_periodicity / frequency(in_ts))`. E.g., biennial groups corresponding to `temporal_grp_periodicity = 2 * frequency(in_ts)` start on an *even year* by default. This behaviour can be changed with argument `temporal_grp_start`. E.g., the preservation of biennial totals starting on an *odd year* instead of an *even year* (default) corresponds to `temporal_grp_start = frequency(in_ts) + 1` (along with `temporal_grp_periodicity = 2 * frequency(in_ts)`).

See the `gs.build_proc_grps()` **Examples** for common processing group scenarios.

See Also

tsraking_driver() tsbalancing() time_values_conv

Examples

```
#######
# Preliminary setup

# Dummy monthly and quarterly time series (2.5 years long)
mth_ts <- ts(rep(NA, 30), start = c(2019, 1), frequency = 12)
mth_ts
qtr_ts <- ts(rep(NA, 10), start = c(2019, 1), frequency = 4)
qtr_ts

# Summarized time series info
ts_info <- function(ts, sep = "-") {
  list(y = gs.time2year(ts),      # years
       p = gs.time2per(ts),       # periods
       n = length(ts),            # length
       f = frequency(ts),         # frequency
       l = gs.time2str(ts, sep))  # labels
}
mth_info <- ts_info(mth_ts)
qtr_info <- ts_info(qtr_ts, sep = "q")

# Function to add a description label for the processing group
add_desc <- function(grp_df, lab_vec, word) {
  grp_df$description <- ifelse(grp_df$complete_grp,
                       paste0("--- ", grp_df$end_per - grp_df$beg_per + 1, " ", word, "s: ",
                                     lab_vec[grp_df$beg_per], " to ",
                                     lab_vec[grp_df$end_per], " --- "),
                            paste0("--- 1 ", word, ": ", lab_vec[grp_df$beg_per], " ---"))
  grp_df
}



#######
# Common processing group scenarios for monthly data


# 0- Month-by-month processing (every single month is a processing group)
mth_grps0 <- gs.build_proc_grps(mth_info$y, mth_info$p, mth_info$n, mth_info$f,
                                temporal_grp_periodicity = 1,
                                temporal_grp_start = 1)
tmp <- add_desc(mth_grps0, mth_info$l, "month")
head(tmp)
tail(tmp)


# Temporal groups corresponding to ...

# 1- calendar years
mth_grps1 <- gs.build_proc_grps(mth_info$y, mth_info$p, mth_info$n, mth_info$f,
                                temporal_grp_periodicity = 12,
```

```
                                temporal_grp_start = 1)
add_desc(mth_grps1, mth_info$l, "month")

# 2- fiscal years starting on April
mth_grps2 <- gs.build_proc_grps(mth_info$y, mth_info$p, mth_info$n, mth_info$f,
                                temporal_grp_periodicity = 12,
                                temporal_grp_start = 4)
add_desc(mth_grps2, mth_info$l, "month")

# 3- regular quarters (starting on Jan, Apr, Jul and Oct)
mth_grps3 <- gs.build_proc_grps(mth_info$y, mth_info$p, mth_info$n, mth_info$f,
                                temporal_grp_periodicity = 3,
                                temporal_grp_start = 1)
add_desc(mth_grps3, mth_info$l, "month")

# 4- quarters shifted by one month (starting on Feb, May, Aug and Nov)
mth_grps4 <- gs.build_proc_grps(mth_info$y, mth_info$p, mth_info$n, mth_info$f,
                                temporal_grp_periodicity = 3,
                                temporal_grp_start = 2)
add_desc(mth_grps4, mth_info$l, "month")




#######
# Common processing group scenarios for quarterly data


# 0- Quarter-by-quarter processing (every single quarter is a processing group)
qtr_grps0 <- gs.build_proc_grps(qtr_info$y, qtr_info$p, qtr_info$n, qtr_info$f,
                                temporal_grp_periodicity = 1,
                                temporal_grp_start = 1)
add_desc(qtr_grps0, qtr_info$l, "quarter")


# Temporal groups corresponding to ...

# 1- calendar years
qtr_grps1 <- gs.build_proc_grps(qtr_info$y, qtr_info$p, qtr_info$n, qtr_info$f,
                                temporal_grp_periodicity = 4,
                                temporal_grp_start = 1)
add_desc(qtr_grps1, qtr_info$l, "quarter")

# 2- fiscal years starting on April (2nd quarter)
qtr_grps2 <- gs.build_proc_grps(qtr_info$y, qtr_info$p, qtr_info$n, qtr_info$f,
                                temporal_grp_periodicity = 4,
                                temporal_grp_start = 2)
add_desc(qtr_grps2, qtr_info$l, "quarter")
```

`gs.gInv_MP` *Moore-Penrose inverse*

Description

This function calculates the Moore-Penrose (pseudo) inverse of a square or rectangular matrix using Singular Value Decomposition (SVD). It is used internally by `tsraking()` and `benchmarking()`.

Usage

```
gs.gInv_MP(X, tol = NA)
```

Arguments

X	(mandatory) Matrix to invert.
tol	(optional) Real number that specifies the tolerance for identifying zero singular values. When tol = NA (default), the tolerance is calculated as the product of the size (dimension) of the matrix, the norm of the matrix (largest singular value) and the *machine epsilon* (.Machine$double.eps). **Default value** is tol = NA.

Details

The default tolerance (argument tol = NA) is coherent with the tolerance used by the MATLAB and GNU Octave software in their general inverse functions. In our testing, this default tolerance also produced solutions (results) comparable to G-Series 2.0 in SAS®.

Value

The Moore-Penrose (pseudo) inverse of matrix X.

See Also

tsraking() benchmarking()

Examples

```
# Invertible matrix
X1 <- matrix(c(3, 2, 8,
               6, 3, 2,
               5, 2, 4), nrow = 3, byrow = TRUE)
Y1 <- gs.gInv_MP(X1)
all.equal(Y1, solve(X1))
X1 %*% Y1

# Rectangular matrix
X2 <- X1[-1, ]
try(solve(X2))
X2 %*% gs.gInv_MP(X2)

# Non-invertible square matrix
X3 <- matrix(c(3, 0, 0,
               0, 0, 0,
               0, 0, 4), nrow = 3, byrow = TRUE)
try(solve(X3))
X3 %*% gs.gInv_MP(X3)
```

`osqp_settings_sequence` *OSQP settings sequence data frame*

Description

Data frame containing a sequence of OSQP settings for `tsbalancing()` specified with argument `osqp_settings_df`. The package includes two predefined OSQP settings sequence data frames:

- `default_osqp_sequence`: fast and effective (default `osqp_settings_df` argument value);
- `alternate_osqp_sequence`: geared towards precision at the expense of execution time.

See `vignette("osqp-settings-sequence-dataframe")` for the actual contents of these data frames.

Usage

```
# Default sequence:
# tsbalancing(..., osqp_settings_df = default_osqp_sequence)

# Alternative (slower) sequence:
# tsbalancing(..., osqp_settings_df = alternate_osqp_sequence)

# Custom-made sequence (use with caution!):
# tsbalancing(..., osqp_settings_df = <my-osqp-sequence-dataframe>)

# Single solving attempt with the default OSQP settings (not recommended!):
# tsbalancing(..., osqp_settings_df = NULL)
```

Format

A data frame with at least one row and at least one column, the *most common* columns being:

max_iter Maximum number of iterations (integer)

sigma Alternating direction method of multipliers (ADMM) sigma step (double)

eps_abs Absolute tolerance (double)

eps_rel Relative tolerance (double)

eps_prim_inf Primal infeasibility tolerance (double)

eps_dual_inf Dual infeasibility tolerance (double)

polish Perform solution polishing (logical)

scaling Number of scaling iterations (integer)

prior_scaling Scale problem data prior to solving with OSQP (logical)

require_polished Require a polished solution to stop the sequence (logical)

[*any-other-OSQP-setting*] Value of the corresponding OSQP setting

Details

With the exception of `prior_scaling` and `require_polished`, all columns of the data frame must correspond to a OSQP setting. Default OSQP values are used for any setting not specified in this data frame. Visit https://osqp.org/docs/interfaces/solver_settings.html for all available OSQP settings. Note that the OSQP `verbose` setting is actually controlled through `tsbalancing()` arguments `quiet` and `display_level` (i.e., column `verbose` in a *OSQP settings sequence data frame* would be ignored).

Each row of a *OSQP settings sequence data frame* represents one attempt at solving a balancing problem with the corresponding OSQP settings. The solving sequence stops as soon as a valid solution is obtained (a solution for which all constraint discrepancies are smaller or equal to the tolerance specified with `tsbalancing()` argument `validation_tol`) unless column `require_polished` = `TRUE`, in which case a polished solution from OSQP (`status_polish` = 1) would also be required to stop the sequence. Constraint discrepancies correspond to $\max(0, l - Ax, Ax - u)$ with constraints defined as $l \leq Ax \leq u$. In the event where a satisfactory solution cannot be obtained after having gone through the entire sequence, `tsbalancing()` returns the solution that generated the smallest total constraint discrepancies among valid solutions, if any, or among all solutions, otherwise. Note that running the entire solving sequence can be *enforced* by specifying `tsbalancing()` argument `full_sequence` = `TRUE`. Rows with column `prior_scaling` = `TRUE` have the problem data scaled prior to solving with OSQP, using the average of the free (nonbinding) problem values as the scaling factor.

In addition to specifying a custom-made *OSQP settings sequence data frame* with argument `osqp_settings_df`, one can also specify `osqp_settings_df` = `NULL` which would result in a single solving attempt with default OSQP values for all settings along with `prior_scaling` = `FALSE` and `require_polished` = `FALSE`. Note that it is recommended, however, to first try data frames `default_osqp_sequence` and `alternate_osqp_sequence`, along with `full_sequence` = `TRUE` if necessary, before considering other alternatives.

Vignette *OSQP Settings Sequence Data Frame* (`vignette("osqp-settings-sequence-dataframe")`) contains additional information.

`plot_benchAdj` *Plot benchmarking adjustments*

Description

Plot benchmarking adjustments for a single series in the current (active) graphics device. Up to three types of adjustments can be overlayed in the same plot:

- Adjustments generated by function `benchmarking()`
- Adjustments generated by function `stock_benchmarking()`
- Cubic spline associated to adjustments generated by function `stock_benchmarking()`

These plots can be useful to assess the quality of the benchmarking results and compare the adjustments generated by both benchmarking functions (`benchmarking()` and `stock_benchmarking()`) for stock series.

Usage

```
plot_benchAdj(
  PB_graphTable = NULL,
  SB_graphTable = NULL,
  SB_splineKnots = NULL,
  legendPos = "bottomright"
)
```

Arguments

PB_graphTable	(optional) Data frame (object of class "data.frame") corresponding to the benchmarking() (PB for "Proc Benchmarking" approach) function output graphTable data frame. Specify NULL not to include the benchmarking() adjustments in the plot. **Default value** is PB_graphTable = NULL.
SB_graphTable	(optional) Data frame (object of class "data.frame") corresponding to the stock_benchmarking() (SB) function output graphTable data frame. Specify NULL not to include the stock_benchmarking() adjustments in the plot. **Default value** is SB_graphTable = NULL.
SB_splineKnots	(optional) Data frame (object of class "data.frame") corresponding to the stock_benchmarking() (SB) function output splineKnots data frame. Specify NULL not to include the stock_benchmarking() cubic spline in the plot. **Default value** is SB_splineKnots = NULL.
legendPos	(optional) String (keyword) specifying the location of the legend in the plot. See the description of argument x in the documentation of graphics::legend() for the list of valid keywords. Specify NULL not to include a legend in the plot. **Default value** is legendPos = "bottomright".

Details

graphTable data frame (arguments PB_graphTable and SB_graphTable) variables used in the plot:

- t for the x-axis values (t)
- benchmarkedSubAnnualRatio for the *Stock Bench. (SB)* and *Proc Bench. (PB)* lines
- bias for the *Bias* line (when $\rho < 1$)

splineKnots data frame (argument SB_splineKnots) variables used in the plot:

- x for the x-axis values (t)
- y for the *Cubic spline* line and the *Extra knot* and *Original knot* points
- extraKnot for the type of knot (*Extra knot* vs. *Original knot*)

See section **Value** of benchmarking() and stock_benchmarking() for more details on these data frames.

Value

This function returns nothing (invisible(NULL)).

See Also

plot_graphTable() bench_graphs benchmarking() stock_benchmarking()

Examples

```
#######
# Preliminary setup

# Quarterly stocks (same annual pattern repeated for 7 years)
qtr_ts <- ts(rep(c(85, 95, 125, 95), 7), start = c(2013, 1), frequency = 4)

# End-of-year stocks
ann_ts <- ts(c(135, 125, 155, 145, 165), start = 2013, frequency = 1)

# Proportional benchmarking
# ... with `benchmarking()` ("Proc Benchmarking" approach)
out_PB <- benchmarking(
  ts_to_tsDF(qtr_ts),
  ts_to_bmkDF(ann_ts, discrete_flag = TRUE, alignment = "e", ind_frequency = 4),
  rho = 0.729, lambda = 1, biasOption = 3,
  quiet = TRUE)
# ... with `stock_benchmarking()`
out_SB <- stock_benchmarking(
  ts_to_tsDF(qtr_ts),
  ts_to_bmkDF(ann_ts, discrete_flag = TRUE, alignment = "e", ind_frequency = 4),
  rho = 0.729, lambda = 1, biasOption = 3,
  quiet = TRUE)


#######
# Plot the benchmarking adjustments

# `benchmarking()` adjustments (`out_PB`), without a legend
plot_benchAdj(PB_graphTable = out_PB$graphTable,
              legendPos = NULL)

# Add the `stock_benchmarking()` (`out_SB`) adjustments, with a legend this time
plot_benchAdj(PB_graphTable = out_PB$graphTable,
              SB_graphTable = out_SB$graphTable)

# Add the `stock_benchmarking()` cubic spline actually used to generate the adjustments
# (incl. the extra knots at both ends), with the legend located in the top-left corner
plot_benchAdj(PB_graphTable = out_PB$graphTable,
              SB_graphTable = out_SB$graphTable,
              SB_splineKnots = out_SB$splineKnots,
              legendPos = "topleft")


#######
# Simulate multiple series benchmarking (3 stock series)

qtr_mts <- ts.union(ser1 = qtr_ts, ser2 = qtr_ts * 100, ser3 = qtr_ts * 10)
ann_mts <- ts.union(ser1 = ann_ts, ser2 = ann_ts * 100, ser3 = ann_ts * 10)

# Using argument `allCols = TRUE` (identify stocks with column `varSeries`)
out_SB2 <- stock_benchmarking(
```

```
  ts_to_tsDF(qtr_mts),
  ts_to_bmkDF(ann_mts, discrete_flag = TRUE, alignment = "e", ind_frequency = 4),
  rho = 0.729, lambda = 1, biasOption = 3,
  allCols = TRUE,
  quiet = TRUE)

# Adjustments for 2nd stock (ser2)
plot_benchAdj(
  SB_graphTable = out_SB2$graphTable[out_SB2$graphTable$varSeries == "ser2", ])

# Using argument `by = "series"` (identify stocks with column `series`)
out_SB3 <- stock_benchmarking(
  stack_tsDF(ts_to_tsDF(qtr_mts)),
  stack_bmkDF(ts_to_bmkDF(
    ann_mts, discrete_flag = TRUE, alignment = "e", ind_frequency = 4)),
  rho = 0.729, lambda = 1, biasOption = 3,
  by = "series",
  quiet = TRUE)

# Cubic spline for 3nd stock (ser3)
plot_benchAdj(
  SB_splineKnots = out_SB3$splineKnots[out_SB3$splineKnots$series == "ser3", ])
```

`plot_graphTable` *Generate benchmarking graphics in a PDF file*

Description

Create a PDF file (US Letter paper size format in landscape view) containing benchmarking graphics for the set of series contained in the specified benchmarking function (`benchmarking()` or `stock_benchmarking()`) output `graphTable` data frame. Four types of benchmarking graphics can be generated for each series:

- **Original Scale Plot** (argument `ori_plot_flag`) - overlay graph of:
  - Indicator series
  - Average indicator series
  - Bias corrected indicator series (when $\rho < 1$)
  - Benchmarked series
  - Average benchmark
- **Adjustment Scale Plot** (argument `adj_plot_flag`) - overlay graph of:
  - Benchmarking adjustments
  - Average benchmarking adjustments
  - Bias line (when $\rho < 1$)
- **Growth Rates Plot** (argument `GR_plot_flag`) - bar chart of the indicator and benchmarked series growth rates.
- **Growth Rates Table** (argument `GR_table_flag`) - table of the indicator and benchmarked series growth rates.

These graphics can be useful to assess the quality of the benchmarking results. Any of the four types of benchmarking graphics can be enabled or disabled with the corresponding flag. The first three types of graphics (the plots) are generated by default while the fourth (growth rates table) is not.

Usage

```
plot_graphTable(
  graphTable,
  pdf_file,
  ori_plot_flag = TRUE,
  adj_plot_flag = TRUE,
  GR_plot_flag = TRUE,
  GR_table_flag = FALSE,
  add_bookmarks = TRUE
)
```

Arguments

`graphTable`	(mandatory) Data frame (object of class "data.frame") corresponding to the benchmarking function output`graphTable` data frame.
`pdf_file`	(mandatory) Name (and path) of the PDF file that will contain the benchmarking graphics. The name should include the ".pdf" file extension. The PDF file is created in the R session working directory (as returned by `getwd()`) if a path is not specified. Specifying `NULL` would cancel the creation of a PDF file.
`ori_plot_flag`, `adj_plot_flag`, `GR_plot_flag`, `GR_table_flag`	(optional) Logical arguments indicating whether or not the corresponding type of benchmarking graphic should be generated. All three plots are generated by default but not the growth rates tables. **Default values** are `ori_plot_flag = TRUE`, `adj_plot_flag = TRUE`, `GR_plot_flag = TRUE` and `GR_table_flag = FALSE`.
`add_bookmarks`	Logical argument indicating whether or not bookmarks should be added to the PDF file. See **Bookmarks** in section **Details** for more information. **Default value** is `add_bookmarks = TRUE`.

Details

List of the `graphTable` data frame variables corresponding to each element of the four types of benchmarking graphics:

- Original Scale Plot (argument `ori_plot_flag`)
  - `subAnnual` for the *Indicator Series* line
  - `avgSubAnnual` for the *Avg. Indicator Series* segments
  - `subAnnualCorrected` for the *Bias Corr. Indicator Series* line (when $\rho < 1$)
  - `benchmarked` for the *Benchmarked Series* line
  - `avgBenchmark` for the *Average Benchmark* segments
- Adjustment Scale Plot (argument `adj_plot_flag`)
  - `benchmarkedSubAnnualRatio` for the *BI Ratios (Benchmarked Series / Indicator Series)* line $^{(*)}$
  - `avgBenchmarkSubAnnualRatio` for the *Average BI Ratios* segments $^{(*)}$
  - `bias` for the *Bias* line (when $\rho < 1$)
- Growth Rates Plot (argument `GR_plot_flag`)

  - growthRateSubAnnual for the *Growth R. in Indicator Series* bars (*)
  - growthRateBenchmarked for the *Growth R. in Benchmarked Series* bars (*)

- Growth Rates Table (argument GR_table_flag)
  - year for the *Year* column
  - period for the *Period* column
  - subAnnual for the *Indicator Series* column
  - benchmarked for the *Benchmarked Series* column
  - growthRateSubAnnual for the *Growth Rate in Indicator Series* column (*)
  - growthRateBenchmarked for the *Growth Rate in Benchmarked Series* column (*)

(*) *BI ratios* and *growth rates* actually correspond to *differences* when $\lambda = 0$ (additive benchmarking).

The function uses the extra columns of the graphTable data frame (columns not listed in the **Value** section of benchmarking() and stock_benchmarking()), if any, to build BY-groups. See section **Benchmarking Multiple Series** of benchmarking() for more details.

Performance:

The two types of growth rates graphics, i.e., the bar chart (GR_plot_flag) and table (GR_table_flag), often requires the generation of several pages in the PDF file, especially for long monthly series with several years of data. This creation of extra pages slows down the execution of plot_graphTable(). This is why only the bar chart is generated by default (GR_plot_flag = TRUE and GR_table_flag = FALSE). Deactivating both types of growth rates graphics (GR_plot_flag = FALSE and GR_table_flag = FALSE) or reducing the size of the input graphTable data frame for very long series (e.g., keeping only recent years) could therefore improve execution time. Also note that the impact of benchmarking on the growth rates can be deduced from the adjustment scale plot (adj_plot_flag) by examining the extent of vertical movement (downward or upward) of the benchmarking adjustments between adjacent periods: the greater the vertical movement, the greater the impact on corresponding growth rate. Execution time of plot_graphTable() could therefore be reduced, if needed, by only generating the first two types of graphics while focusing on the adjustment scale plot to assess period-to-period movement preservation, i.e., the impact of benchmarking on the initial growth rates.

ggplot2 themes:

The plots are generated with the ggplot2 package which comes with a convenient set of complete themes for the general look and feel of the plots (with theme_grey() as the default theme). Use function theme_set() to change the theme applied to the plots generated by plot_graphTable() (see the **Examples**).

Bookmarks:

Bookmarks are added to the PDF file with xmpdf::set_bookmarks() when argument add_bookmarks = TRUE (default), which requires a command-line tool such as Ghostscript or PDFtk. See section **Installation** in vignette("xmpdf", package = "xmpdf") for details.

Important: bookmarks will be successfully added to the PDF file **if and only if** xmpdf::supports_set_bookmarks() returns TRUE **and** the execution of xmpdf::set_bookmarks() is successful. If Ghostscript is installed on your machine but xmpdf::supports_set_bookmarks() still returns FALSE, try specifying the path of the Ghostscript executable in environment variable R_GSCMD (e.g., Sys.setenv(R_GSCMD = "C:/Program Files/.../bin/gswin64c.exe") on Windows). On the other hand, if xmpdf::supports_set_bookmarks()} returns TRUE but you are experiencing (irresolvable) issues with xmpdf::set_bookmarks() (e.g., error related to the Ghostscript executable), bookmarks creation can be disabled by specifying add_bookmarks = FALSE.

Value

In addition to creating a PDF file containing the benchmarking graphics (except when `pdf_file = NULL`), this function also invisibly returns a list with the following elements:

- `graphTable`: Character string (character vector of length one) that contains the complete name and path of the PDF file if it was successfully created and `invisible(NA_character_)` otherwise or if `pdf_file = NULL` was specified.
- `graph_list`: List of the generated benchmarking graphics (one per series) with the following elements:
  - `name`: Character string describing the series (matches the bookmark name in the PDF file).
  - `page`: Integer representing the sequence number of the first graphic for the series in the entire sequence of graphics for all series (matches the page number in the PDF file).
  - `ggplot_list`: List of ggplot objects (one per graphic or page in the PDF file) corresponding to the generated benchmarking graphics for the series. See section **Value** in bench_graphs for details.

Note that the returned ggplot objects can be displayed *manually* with `print()`, in which case some updates to the ggplot2 theme defaults are recommended in order to produce graphics with a similar look and feel as those generated in the PDF file (see section **Value** in bench_graphs for details). Also keep in mind that these graphics are optimized for the US Letter paper size format in landscape view (as displayed in the PDF file), i.e., 11in wide (27.9cm, 1056px with 96 DPI) and 8.5in tall (21.6cm, 816px with 96 DPI).

See Also

bench_graphs `plot_benchAdj()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Set the working directory (for the PDF files)
iniwd <- getwd()
setwd(tempdir())

# Quarterly car and van sales (indicator series)
qtr_ind <- ts_to_tsDF(
  ts(matrix(c(# Car sales
              1851, 2436, 3115, 2205, 1987, 2635, 3435, 2361, 2183, 2822,
              3664, 2550, 2342, 3001, 3779, 2538, 2363, 3090, 3807, 2631,
              2601, 3063, 3961, 2774, 2476, 3083, 3864, 2773, 2489, 3082,
              # Van sales
              1900, 2200, 3000, 2000, 1900, 2500, 3800, 2500, 2100, 3100,
              3650, 2950, 3300, 4000, 3290, 2600, 2010, 3600, 3500, 2100,
              2050, 3500, 4290, 2800, 2770, 3080, 3100, 2800, 3100, 2860),
            ncol = 2),
     start = c(2011, 1),
     frequency = 4,
     names = c("car_sales", "van_sales")))

# Annual car and van sales (benchmarks)
ann_bmk <- ts_to_bmkDF(
  ts(matrix(c(# Car sales
              10324, 10200, 10582, 11097, 11582, 11092,
              # Van sales
```

```
              12000, 10400, 11550, 11400, 14500, 16000),
            ncol = 2),
      start = 2011,
      frequency = 1,
      names = c("car_sales", "van_sales")),
  ind_frequency = 4)

# Proportional benchmarking without bias correction
out_bench <- benchmarking(qtr_ind, ann_bmk,
                          rho = 0.729, lambda = 1, biasOption = 1,
                          allCols = TRUE,
                          quiet = TRUE)


# Default set of graphics (the first 3 types of plots)
plot_graphTable(out_bench$graphTable, "bench_graphs.pdf")

# Temporarily use ggplot2 `theme_bw()` for the plots
library(ggplot2)
ini_theme <- theme_get()
theme_set(theme_bw())
plot_graphTable(out_bench$graphTable, "bench_graphs_bw.pdf")
theme_set(ini_theme)

# Generate all 4 types of graphics (including the growth rates table)
plot_graphTable(out_bench$graphTable, "bench_graphs_with_GRTable.pdf",
                GR_table_flag = TRUE)

# Reduce execution time by disabling both types of growth rates graphics
plot_graphTable(out_bench$graphTable, "bench_graphs_no_GR.pdf",
                GR_plot_flag = FALSE)


# Reset the working directory to its initial location
setwd(iniwd)
```

`rkMeta_to_blSpecs` *Convert reconciliation metadata*

Description

Convert a `tsraking()` metadata data frame to a `tsbalancing()` problem specs data frame.

Usage

```
rkMeta_to_blSpecs(
  metadata_df,
  alterability_df = NULL,
  alterSeries = 1,
  alterTotal1 = 0,
  alterTotal2 = 0,
  alterability_df_only = FALSE
)
```

Arguments

`metadata_df` (mandatory)

Data frame (object of class "data.frame") that describes the cross-sectional aggregation constraints (additivity rules) for the raking problem. Two character variables must be included in the metadata data frame: `series` and `total1`. Two variables are optional: `total2` (character) and `alterAnnual` (numeric). The values of variable `series` represent the variable names of the component series in the input time series data frame (argument `data_df`). Similarly, the values of variables `total1` and `total2` represent the variable names of the 1st and 2nd dimension cross-sectional control totals in the input time series data frame. Variable `alterAnnual` contains the alterability coefficient for the temporal constraint associated to each component series. When specified, the latter will override the default alterability coefficient specified with argument `alterAnnual`.

`alterability_df`

(optional)

Data frame (object of class "data.frame"), or `NULL`, that contains the alterability coefficients variables. They must correspond to a component series or a cross-sectional control total, that is, a variable with the same name must exist in the input time series data frame (argument `data_df`). The values of these alterability coefficients will override the default alterability coefficients specified with arguments `alterSeries`, `alterTotal1` and `alterTotal2`. When the input time series data frame contains several observations and the alterability coefficients data frame contains only one, the alterability coefficients are used (repeated) for all observations of the input time series data frame. Alternatively, the alterability coefficients data frame may contain as many observations as the input time series data frame.

Default value is `alterability_df = NULL` (default alterability coefficients).

`alterSeries` (optional)

Nonnegative real number specifying the default alterability coefficient for the component series values. It will apply to component series for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterSeries = 1.0` (nonbinding component series values).

`alterTotal1` (optional)

Nonnegative real number specifying the default alterability coefficient for the 1st dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterTotal1 = 0.0` (binding 1st dimension cross-sectional control totals)

`alterTotal2` (optional)

Nonnegative real number specifying the default alterability coefficient for the 2nd dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterTotal2 = 0.0` (binding 2nd dimension cross-sectional control totals).

`alterability_df_only`

(optional)

Logical argument specifying whether or not only the set of alterability ceofficients found in the alterability file (argument `alterability_df`) should be included in the returned `tsbalancing()` problem specs data frame. When `alterability_df_only` = `FALSE` (the default), the alterability coefficients specified with arguments `alterSeries`, `alterTotal1` and `alterTotal2` are combined with those found in `alterability_df` (the latter coefficients overwriting the former) and the returned data frame therefore contains alterability coefficients for all component and cross-sectional control total series. This argument does not affect the set of temporal total alterability coefficients (associated to `tsraking()` argument `alterAnnual`) that are included in the returned `tsbalancing()` problem specs data frame. The latter always strictly contains those specified in `metadata_df` with a non-missing (non-`NA`) value for column `alterAnnual`.

Default value is `alterability_df_only` = `FALSE`.

Details

The preceding description of argument `alterability_df` comes from `tsraking()`. This function (`rkMeta_to_blSpecs()`) slightly changes the specification of alterability coefficients with argument `alterability_df` by allowing either

- a single observation, specifying the set of alterability coefficients to use for all periods,
- or one or several observations with an additional column named `timeVal` allowing the specification of both period-specific alterability coefficients (`timeVal` is not `NA`) and generic coefficients to use for all other periods (`timeVal` is `NA`). Values for column `timeVal` correspond to the *time values* of a "ts" object as returned by `stats::time()`, conceptually corresponding to $year + (period - 1)/frequency$.

Another difference with `tsraking()` is that missing (`NA`) values are allowed in the alterability coefficients data frame (argument `alterability_df`) and result in using the generic coefficients (observations for which `timeVal` is `NA`) or the default coefficients (arguments `alterSeries`, `alterTotal1` and `alterTotal2`).

Note that apart from discarding alterability coefficients for series not listed in the `tsraking()` metadata data frame (argument `metadata_df`), this function does not validate the values specified in the alterability coefficients data frame (argument `alterability_df`) nor the ones specified with column `alterAnnual` in the `tsraking()` metadata data frame (argument `metadata_df`). The function transfers them *as is* in the returned `tsbalancing()` problem specs data frame.

Value

A `tsbalancing()` problem specs data frame (argument `problem_specs_df`).

See Also

`tsraking()` `tsbalancing()`

Examples

```
# `tsraking()` metadata for a 2-dimensional raking problem (2 x 2 table)
my_metadata <- data.frame(series = c("A1", "A2", "B1", "B2"),
                          total1 = c("totA", "totA", "totB", "totB"),
                          total2 = c("tot1", "tot2", "tot1", "tot2"))
my_metadata
```

```
# Convert to `tsbalancing()` specifications

# Include the default `tsraking()` alterability coefficients
rkMeta_to_blSpecs(my_metadata)

# Almost binding 1st marginal totals (small alter. coef for columns `totA` and `totB`)
tail(rkMeta_to_blSpecs(my_metadata, alterTotal1 = 1e-6))

# Do not include alterability coefficients (aggregation constraints only)
rkMeta_to_blSpecs(my_metadata, alterability_df_only = TRUE)

# With an alterability coefficients file (argument `alterability_df`)
my_alter = data.frame(B2 = 0.5)
tail(rkMeta_to_blSpecs(my_metadata, alterability_df = my_alter))

# Only include the alterability coefficients from `alterability_df`
# (i.e., for column `B2` only)
tail(rkMeta_to_blSpecs(my_metadata, alterability_df = my_alter,
                       alterability_df_only = TRUE))
```

`stack_bmkDF` *Stack benchmarks data*

Description

Convert a multivariate benchmarks data frame (see `ts_to_bmkDF()`) for the benchmarking functions (`benchmarking()` and `stock_benchmarking()`) into a stacked (tall) data frame with six variables (columns):

- one (1) for the benchmark name (e.g., series name)
- four (4) for the benchmark coverage
- one (1) for the benchmark value

Missing (`NA`) benchmark values are not included in the output stacked data frame by default. Specify argument `keep_NA = TRUE` in order to keep them.

This function is useful when intending to use the `by` argument (*BY-group* processing mode) of the benchmarking functions in order to benchmark multiple series in a single function call.

Usage

```
stack_bmkDF(
  bmk_df,
  ser_cName = "series",
  startYr_cName = "startYear",
  startPer_cName = "startPeriod",
  endYr_cName = "endYear",
  endPer_cName = "endPeriod",
  val_cName = "value",
  keep_NA = FALSE
)
```

Arguments

`bmk_df` (mandatory)

Data frame (object of class "data.frame") that contains the multivariate benchmarks to be stacked.

`ser_cName` (optional)

String specifying the name of the character variable (column) in the output stacked data frame that will contain the benchmark names (name of the benchmark variables in the input multivariate benchmarks data frame). This variable can then be used as the BY-group variable (argument `by`) with the benchmarking functions.

Default value is `ser_cName = "series"`.

`startYr_cName`, `startPer_cName`, `endYr_cName`, `endPer_cName` (optional)

Strings specifying the name of the numeric variables (columns) in the input multivariate benchmarks data frame that define the benchmark coverage, i.e., the starting and ending year and period (cycle) identifiers. These variables are *transferred* to the output stacked data frame with the same variable names.

Default values are `startYr_cName = "startYear"`, `startPer_cName = "startPeriod"` `endYr_cName = "endYear"` and `endPer_cName = "endPeriod"`.

`val_cName` (optional)

String specifying the name of the numeric variable (column) in the output stacked data frame that will contain the benchmark values.

Default value is `val_cName = "value"`.

`keep_NA` (optional)

Logical argument specifying whether missing (`NA`) benchmark values in the input multivariate benchmarks data frame should be kept in the output stacked data frame.

Default value is `keep_NA = FALSE`.

Value

The function returns a data frame with six variables:

- Benchmark (series) name, type character (see argument `ser_cName`)
- Benchmark coverage starting year, type numeric (see argument `startYr_cName`)
- Benchmark coverage starting period, type numeric (see argument `startPer_cName`)
- Benchmark coverage ending year, type numeric (see argument `endtYr_cName`)
- Benchmark coverage ending period, type numeric (see argument `endPer_cName`)
- Benchmark value, type numeric (see argument `val_cName`)

Note: the function returns a "data.frame" object than can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

See Also

`stack_tsDF()` `ts_to_bmkDF()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Create an annual benchmarks data frame for 2 quarterly indicator series
# (with missing benchmark values for the last 2 years)
my_benchmarks <- ts_to_bmkDF(ts(data.frame(ser1 = c(1:3 *  10, NA, NA),
                                           ser2 = c(1:3 * 100, NA, NA)),
                                start = c(2019, 1), frequency = 1),
                             ind_frequency = 4)
my_benchmarks


# Stack the benchmarks ...

# discarding `NA` values in the output stacked data frame (default behavior)
stack_bmkDF(my_benchmarks)

# keep `NA` values in the output stacked data frame
stack_bmkDF(my_benchmarks, keep_NA = TRUE)

# using custom variable (column) names
stack_bmkDF(my_benchmarks, ser_cName = "bmk_name", val_cName = "bmk_val")
```

---

stack_tsDF *Stack time series data*

Description

Convert a multivariate time series data frame (see `ts_to_tsDF()`) for the benchmarking functions (`benchmarking()` and `stock_benchmarking()`) into a stacked (tall) data frame with four variables (columns):

- one (1) for the series name
- two (2) for the data point identification (year and period)
- one (1) for the data point value

Missing (`NA`) series values are not included in the output stacked data frame by default. Specify argument `keep_NA = TRUE` in order to keep them.

This function is useful when intending to use the `by` argument (*BY-group* processing mode) of the benchmarking functions in order to benchmark multiple series in a single function call.

Usage

```
stack_tsDF(
  ts_df,
  ser_cName = "series",
  yr_cName = "year",
  per_cName = "period",
  val_cName = "value",
  keep_NA = FALSE
)
```

Arguments

`ts_df` (mandatory)

Data frame (object of class "data.frame") that contains the multivariate time series data to be stacked.

`ser_cName` (optional)

String specifying the name of the character variable (column) in the output stacked data frame that will contain the series names (name of the time series variables in the input multivariate time series data frame). This variable can then be used as the BY-group variable (argument `by`) with the benchmarking functions.

Default value is `ser_cName = "series"`.

`yr_cName, per_cName`

(optional)

Strings specifying the name of the numeric variables (columns) in the input multivariate time series data frame that contain the data point year and period (cycle) identifiers. These variables are *transferred* to the output stacked data frame with the same variable names.

Default values are `yr_cName = "year"` and `per_cName = "period"`.

`val_cName` (optional)

String specifying the name of the numeric variable (column) in the output stacked data frame that will contain the data point values.

Default value is `val_cName = "value"`.

`keep_NA` (optional)

Logical argument specifying whether `NA` time series values in the input multivariate time series data frame should be kept in the output stacked data frame.

Default value is `keep_NA = FALSE`.

Value

The function returns a data frame with four variables:

- Series name, type character (see argument `ser_cName`)
- Data point year, type numeric (see argument `yr_cName`)
- Data point period, type numeric (see argument `per_cName`)
- Data point value, type numeric (see argument `val_cName`)

Note: the function returns a "data.frame" object than can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

See Also

`unstack_tsDF()` `stack_bmkDF()` `ts_to_tsDF()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Create a data frame with 2 quarterly indicators series
# (with missing values for the last 2 quarters)
my_indicators <- ts_to_tsDF(ts(data.frame(ser1 = c(1:5 *  10, NA, NA),
                                          ser2 = c(1:5 * 100, NA, NA)),
```

```
                                start = c(2019, 1), frequency = 4))
my_indicators


# Stack the indicator series ...

# discarding `NA` values in the output stacked data frame (default behavior)
stack_tsDF(my_indicators)

# keeping `NA` values in the output stacked data frame
stack_tsDF(my_indicators, keep_NA = TRUE)

# using custom variable (column) names
stack_tsDF(my_indicators, ser_cName = "ser_name", val_cName = "ser_val")
```

`stock_benchmarking` *Restore temporal constraints for stock series*

Description

Function specifically aimed at benchmarking stock series where the benchmarks are anchor points covering a single period of the indicator series. Benchmarks covering more than one period of the indicator series cannot be used with this function. Function `benchmarking()` should be used instead to benchmark non-stock series (flows).

Several stock series can be benchmarked in a single function call.

Note that functions `stock_benchmarking()` and `benchmarking()` mainly share the same arguments and return the same type of object. Differences are listed below:

- Argument `verbose` is not defined for `stock_benchmarking()`.
- Extra arguments defined for `stock_benchmarking()`:
  - `low_freq_periodicity`
  - `n_low_freq_proj`
  - `proj_knots_rho_bd`
- The list returned by `stock_benchmarking()` contains an extra (fourth) data frame:
  - `splineKnots`

See section **Details** for more information on the similarities and differences of functions `stock_benchmarking()` and `benchmarking()`.

A direct equivalent of `stock_benchmarking()` *does not exist in SAS® G-Series 2.0.*

Usage

```
stock_benchmarking(
  series_df,
  benchmarks_df,
  rho,
  lambda,
  biasOption,
  bias = NA,
  low_freq_periodicity = NA,
```

```
  n_low_freq_proj = 1,
  proj_knots_rho_bd = 0.995,
  tolV = 0.001,
  tolP = NA,
  warnNegResult = TRUE,
  tolN = -0.001,
  var = "value",
  with = NULL,
  by = NULL,
  constant = 0,
  negInput_option = 0,
  allCols = FALSE,
  quiet = FALSE
)
```

Arguments

`series_df` (mandatory)

Data frame (object of class "data.frame") that contains the indicator time series data to be benchmarked. In addition to the series data variable(s), specified with argument `var`, the data frame must also contain two numeric variables, `year` and `period`, identifying the periods of the indicator time series.

`benchmarks_df` (mandatory)

Data frame (object of class "data.frame") that contains the benchmarks. In addition to the benchmarks data variable(s), specified with argument `with`, the data frame must also contain four numeric variables, `startYear`, `startPeriod`, `endYear` and `endPeriod`, identifying the indicator time series periods covered by each benchmark.

`rho` (mandatory)

Real number in the $[0, 1]$ interval that specifies the value of the autoregressive parameter ρ. See section **Details** for more information on the effect of parameter ρ.

`lambda` (mandatory)

Real number, with suggested values in the $[-3, 3]$ interval, that specifies the value of the adjustment model parameter λ. Typical values are `lambda = 0.0` for an additive model and `lambda = 1.0` for a proportional model.

`biasOption` (mandatory)

Specification of the bias estimation option:

- `1`: Do not estimate the bias. The bias used to correct the indicator series will be the value specified with argument `bias`.
- `2`: Estimate the bias, display the result, but do not use it. The bias used to correct the indicator series will be the value specified with argument `bias`.
- `3`: Estimate the bias, display the result and use the estimated bias to correct the indicator series. Any value specified with argument `bias` will be ignored.

Argument `biasOption` is ignored when `rho = 1.0`. See section **Details** for more information on the bias.

`bias` (optional)

Real number, or `NA`, specifying the value of the user-defined bias to be used for the correction of the indicator series prior to benchmarking. The bias is added to

the indicator series with an additive model (argument `lambda = 0.0`) while it is multiplied otherwise (argument `lambda != 0.0`). No bias correction is applied when `bias = NA`, which is equivalent to specifying `bias = 0.0` when `lambda = 0.0` and `bias = 1.0` otherwise. Argument `bias` is ignored when `biasOption = 3` or `rho = 1.0`. See section **Details** for more information on the bias.

Default value is `bias = NA` (no user-defined bias).

`low_freq_periodicity`

(optional)

Positive integer representing the number of periods defining the *low* (e.g., benchmarks) frequency for adding the extra spline knots (before the first benchmark and after the last benchmark). For example, `low_freq_periodicity = 3` with monthly indicators would define quarterly knots. Annual knots are added when `low_freq_periodicity = NA`.

Default value is `low_freq_periodicity = NA` (annual knots).

`n_low_freq_proj`

(optional)

Nonnegative integer representing the number of low frequency knots (as defined with argument `low_freq_periodicity`) to add at both ends (before the first benchmark and after the last benchmark) before starting to add *high* (indicator series) frequency knots.

Default value is `n_low_freq_proj = 1`.

`proj_knots_rho_bd`

(optional)

Bound that applies to the value specified with argument `rho` and determines the type of extra knots to be added at both ends (before the first benchmark and after the last benchmark). When `rho > proj_knots_rho_bd`, *high* (indicator series) frequency knots are used right away. Otherwise, when `rho <= proj_knots_rho_bd`, *low* frequency knots (see arguments `low_freq_periodicity` and `n_low_freq_proj`) are first projected on either side. Note that for quarterly stocks, the cube of the specified `proj_knots_rho_bd` value is actually used. Therefore, the value for argument `proj_knots_rho_bd` should correspond to monthly stock indicators; it is internally adjusted for quarterly stocks. This argument aims at reaching a compromise for the set periods outside (before or after) the provided benchmarks (anchor points), i.e., Denton-type (straight line) adjustments as `rho` approaches 1 (when `rho > proj_knots_rho_bd`) and a natural looking (not overly contorted) spline otherwise (when `rho <= proj_knots_rho_bd`). Section **Details** contains more information on this subject and some illustrative cases are provided in section **Examples**.

Default value is `proj_knots_rho_bd = 0.995` (0.995^3 for quarterly stock indicators).

`tolV`, `tolP` (optional)

Nonnegative real number, or `NA`, specifying the tolerance, in absolute value or percentage, to be used for the validation of the output binding benchmarks (alterability coefficient of 0.0). This validation compares the input binding benchmark values with the equivalent values calculated from the benchmarked series (output) data. Arguments `tolV` and `tolP` cannot be both specified (one must be specified while the other must be `NA`).

Example: to set a tolerance of 10 *units*, specify `tolV = 10, tolP = NA`; to set a tolerance of 1%, specify `tolV = NA, tolP = 0.01`.

Default values are `tolV = 0.001` and `tolP = NA`.

warnNegResult (optional)

Logical argument specifying whether a warning message is generated when a negative value created by the function in the benchmarked (output) series is smaller than the threshold specified by argument tolN.

Default value is warnNegResult = TRUE.

tolN (optional)

Negative real number specifying the threshold for the identification of negative values. A value is considered negative when it is smaller than this threshold.

Default value is tolN = -0.001.

var (optional)

String vector (minimum length of 1) specifying the variable name(s) in the indicator series data frame (argument series_df) containing the values and (optionally) the user-defined alterability coefficients of the series to be benchmarked. These variables must be numeric.

The syntax is var = c("series1 </ alt_ser1>", "series2 </ alt_ser2>", ...). Default alterability coefficients of 1.0 are used when a user-defined alterability coefficients variable is not specified alongside an indicator series variable. See section **Details** for more information on alterability coefficients.

Example: var = "value / alter" would benchmark indicator series data frame variable value with the alterability coefficients contained in variable alter while var = c("value / alter", "value2") would additionally benchmark variable value2 with default alterability coefficients of 1.0.

Default value is var = "value" (benchmark variable value using default alterability coefficients of 1.0).

with (optional)

String vector (same length as argument var), or NULL, specifying the variable name(s) in the benchmarks data frame (argument benchmarks_df) containing the values and (optionally) the user-defined alterability coefficients of the benchmarks. These variables must be numeric. Specifying with = NULL results in using benchmark variable(s) with the same names(s) as those specified with argument var without user-defined benchmark alterability coefficients (i.e., default alterability coefficients of 0.0 corresponding to binding benchmarks).

The syntax is with = NULL or with = c("bmk1 </ alt_bmk1>", "bmk2 </ alt_bmk2>", ...). Default alterability coefficients of 0.0 (binding benchmarks) are used when a user-defined alterability coefficients variable is not specified alongside a benchmark variable. See section **Details** for more information on alterability coefficients.

Example: with = "val_bmk" would use benchmarks data frame variable val_bmk with default benchmark alterability coefficients of 0.0 to benchmark the indicator series while with = c("val_bmk", "val_bmk2 / alt_bmk2") would additionally benchmark a second indicator series using benchmark variable val_bmk2 with the benchmark alterability coefficients contained in variable alt_bmk2.

Default value is with = NULL (same benchmark variable(s) as argument var using default benchmark alterability coefficients of 0.0).

by (optional)

String vector (minimum length of 1), or NULL, specifying the variable name(s) in the input data frames (arguments series_df and benchmarks_df) to be used to form groups (for *BY-group* processing) and allow the benchmarking of multiple series in a single function call. BY-group variables can be numeric or character

(factors or not), must be present in both input data frames and will appear in all three output data frames (see section **Value**). BY-group processing is not implemented when by = NULL. See "Benchmarking Multiple Series" in section **Details** for more information.

Default value is by = NULL (no BY-group processing).

constant (optional)

Real number that specifies a value to be temporarily added to both the indicator series and the benchmarks before solving proportional benchmarking problems (lambda != 0.0). The temporary constant is removed from the final output benchmarked series. E.g., specifying a (small) constant would allow proportional benchmarking with rho = 1 (e.g., proportional Denton benchmarking) on indicator series that include values of 0. Otherwise, proportional benchmarking with values of 0 in the indicator series is only possible when rho < 1. Specifying a constant with additive benchmarking (lambda = 0.0) has no impact on the resulting benchmarked data. The data variables in the graphTable output data frame include the constant, corresponding to the benchmarking problem that was actually solved.

Default value is constant = 0 (no temporary additive constant).

negInput_option

(optional)

Handling of negative values in the input data for proportional benchmarking (lambda != 0.0):

- 0: Do not allow negative values with proportional benchmarking. An error message is displayed in the presence of negative values in the input indicator series or benchmarks and missing (NA) values are returned for the benchmarked series. This corresponds to the G-Series 2.0 behaviour.
- 1: Allow negative values with proportional benchmarking but display a warning message.
- 2: Allow negative values with proportional benchmarking without displaying any message.

Default value is negInput_option = 0 (do not allow negative values with proportional benchmarking).

allCols (optional)

Logical argument specifying whether all variables in the indicator series data frame (argument series_df), other than year and period, determine the set of series to benchmark. Values specified with arguments var and with are ignored when allCols = TRUE, which automatically implies default alterability coefficients, and variables with the same names as the indicator series must exist in the benchmarks data frame (argument benchmarks_df).

Default value is allCols = FALSE.

quiet (optional)

Logical argument specifying whether or not to display only essential information such as warning messages, error messages and variable (series) or BY-group information when multiple series are benchmarked in a single call to the function. We advise against *wrapping* your benchmarking() call with suppressMessages() to further suppress the display of variable (series) or BY-group information when processing multiple series as this would make troubleshooting difficult in case of issues with individual series. Note that specifying quiet = TRUE would also *nullify* argument verbose.

Default value is quiet = FALSE.

Details

Comparison with benchmarking():

With stock series, benchmarking() is known to produce breaks in the benchmarking adjustments at periods corresponding to the benchmark stocks (anchor points). stock_benchmarking() addresses this issue by working directly on the benchmarking adjustments. Smooth adjustments for stocks are ensured by estimating a *slope=0* cubic spline (a spline that is flat at the end knots) going through knots corresponding to the difference (when argument lambda = 0.0) or ratio (otherwise) between the benchmarks (anchor points) and the corresponding indicator series values. These knots are sometimes referred to as *BI* (**B**enchmark-to-**I**ndicator) *differences* or *BI ratios*. Interpolations from the estimated cubic spline then provide the adjustments for the periods between benchmarks.

Arguments rho, lambda, biasOptionand bias play a similar role as in benchmarking(). However, note that for stock_benchmarking(), argument rho only affects the results for periods outside of, or around the, first and last benchmarks and lambda only takes two values in practice: lambda = 0.0 for additive adjustments (spline interpolations where the knots are *BI differences*) or lambda = 1.0 for multiplicative adjustments (spline interpolations where the knots are *BI ratios*). Any nonzero value for lambda would return the same result as lambda = 1.0. Alterability coefficients also play a similar role as in benchmarking() and have the same default values, i.e., 1.0 for the indicator series (nonbinding values) and 0.0 for the benchmarks (binding benchmarks). However, similar to argument lambda, alterability coefficients in this function only take two values in practice: 0.0 for binding values or 1.0 for nonbinding values. Any nonzero alterability coefficient would return the same result as a coefficient of 1.0. Another difference with benchmarking() is that user-defined alterability coefficients are allowed even when rho = 1 with stock_benchmarking(). Finally, specifying a nonbinding benchmark with stock_benchmarking() is equivalent to ignoring the benchmark entirely, as if the benchmark was not included in the input benchmarks file. Compared to benchmarking(), this generally translates into nonbinding benchmarks having a larger impact on the resulting benchmarked stocks.

Solution around the first and last benchmarks (benchmarking *timeliness issue*):

A *slope=0* spline is chosen because it conceptually corresponds to the (popular) *Denton benchmarking* approach (rho = 1). In order to provide a solution before the first benchmark and after the last benchmark that is similar to benchmarking() when rho < 1, i.e., adjustments converging to the bias at a speed dictated by argument rho, extra knots are added at both ends before estimating the spline. By default, one extra low frequency (as defined with argument low_freq_periodicity) knot is added on each side (beginning and end), i.e. one extra knot is added before the first benchmark and after the last benchmark. Then, high (indicator series) frequency knots are added to cover the indicator series span to which is added an extra year worth of high frequency knots. The value of all those extra knots is based on arguments rho, biasOption and bias. This produces natural looking, smooth adjustments for periods outside of or around the first and last benchmarks that gradually converge to the bias, similarly to benchmarking(). The number of extra low frequency knots to be added can be modified with argument n_low_freq_proj. Using high frequency knots right away (n_low_freq_proj = 0) would produce the same projected adjustments as benchmarking(). However, note that this tends to produce an unnatural looking (overly contorted) spline around the first and last benchmarks that could be substantially revised once the next benchmark is available. Using the default n_low_freq_proj = 1 generally works better. However, when rho is *close to 1* (see argument proj_knots_rho_bd), high frequency knots are immediately added on each side in order to ensure Denton-type (straight line) projected adjustments for periods outside of the first and last benchmarks. Finally, a *slope=0* cubic spline is fitted through the (original and extra) knots. Note that in practice, the *slope=0* spline is actually approximated by replicating the value of the end

knots 100 times within the following period (at a frequency corresponding to 100 times the indicator series frequency).

A *natural spline* at the original end knots (first and last benchmarks) can be approximated by specifying a large value for argument `low_freq_periodicity`. The larger the value of `low_freq_periodicity`, the more the cubic spline at the end knots will behave like a *natural spline* (2nd derivative equal to 0 at the end knots, i.e., a spline that keeps a constant slope at the end knots as opposed to being flat like a *slope=0* spline).

In summary, the projected adjustments are controlled with arguments `rho`, `bias` (and `biasOption`), `n_low_freq_proj`, `proj_knots_rho_bd` and `low_freq_periodicity`:

- Default values for these arguments produce `benchmarking()` function-like projected adjustments (reasonably slow convergence to the bias).
- Smaller values of `rho` would generate faster convergence to the bias.
- Specifying a user-defined bias with argument `bias` when `rho < 1` is another way to influence the shape of the projected adjustments.
- Specifying `rho = 1` produce Denton-like projected adjustments (repeated first/last adjustments without convergence to the bias).
- Specifying a large value for `low_freq_periodicity` generates projected adjustments that behave more like a natural spline, i.e., adjustments that continue in the same direction at the first/last benchmark. The larger the value of `low_freq_periodicity`, the more the projected adjustments keep on going in the same direction before *turning around*.

Note on revisions to the benchmarking adjustments:

`benchmarking()` adjustments would not be revised if all future benchmarks were to fall exactly on the projected ones (based on the bias and value of `rho`) and the bias was fixed. The same could be achieved with `stock_benchmarking()` if *enough* low (e.g., benchmarks) frequency knots were projected. The problem with this approach, however, is that the projected adjustments may not look natural as the spline may oscillate more than desired around the projected knots. This is clearly noticeable as `rho` approaches 1 and the spline oscillates around the horizontally aligned projected knots instead of being aligned in a perfectly straight line. The default implementation of the spline around the first and last benchmarks described previously aims at reaching a *best compromise* solution:

- a natural looking spline around the end knots avoiding oscillations and excessive contortions;
- small revisions to the spline if the next benchmark is close to the projected one when `rho` is *far enough* from 1 (`rho <= proj_knots_rho_bd`);
- projected adjustments that are in a straight line (free of oscillations) as `rho` approaches 1 (`rho > proj_knots_rho_bd`).

Subsections *Benchmarking Multiple Series*, *Arguments* `constant` *and* `negInput_option` and *Treatment of Missing (*`NA`*) Values* at the end of the `benchmarking()` **Details** section are also relevant for `stock_benchmarking()`. Consult them as necessary.

Finally, note that the cubic spline associated to the `stock_benchmarking()` adjustments can be conveniently plotted with `plot_benchAdj()`. The latter is used in the **Examples** to illustrate some of the topics discussed above.

Value

The function returns is a list of four data frames:

- `series`: data frame containing the benchmarked data (primary function output). BY-group variables specified with argument `by` would be included in the data frame but not alterability coefficient variables specified with argument `var`.

- benchmarks: copy of the input benchmarks data frame (excluding invalid benchmarks when applicable). BY-group variables specified with argument by would be included in the data frame but not alterability coefficient variables specified with argument with.
- graphTable: data frame containing supplementary data useful to produce analytical tables and graphs (see function plot_graphTable()). It contains the following variables in addition to the BY-group variables specified with argument by:
  - varSeries: Name of the indicator series variable
  - varBenchmarks: Name of the benchmark variable
  - altSeries: Name of the user-defined indicator series alterability coefficients variable
  - altSeriesValue: Indicator series alterability coefficients
  - altbenchmarks: Name of the user-defined benchmark alterability coefficients variable
  - altBenchmarksValue: Benchmark alterability coefficients
  - t: Indicator series period identifier (1 to T)
  - m: Benchmark coverage periods identifier (1 to M)
  - year: Data point calendar year
  - period: Data point period (cycle) value (1 to periodicity)
  - rho: Autoregressive parameter ρ (argument rho)
  - lambda: Adjustment model parameter λ (argument lambda)
  - bias: Bias adjustment (default, user-defined or estimated bias according to arguments biasOption and bias)
  - periodicity: The maximum number of periods in a year (e.g. 4 for a quarterly indicator series)
  - date: Character string combining the values of variables year and period
  - subAnnual: Indicator series values
  - benchmarked: Benchmarked series values
  - avgBenchmark: Benchmark values divided by the number of coverage periods
  - avgSubAnnual: Indicator series values (variable subAnnual) averaged over the benchmark coverage period
  - subAnnualCorrected: Bias corrected indicator series values
  - benchmarkedSubAnnualRatio: Difference ($\lambda = 0$) or ratio ($\lambda \neq 0$) of the values of variables benchmarked and subAnnual
  - avgBenchmarkSubAnnualRatio: Difference ($\lambda = 0$) or ratio ($\lambda \neq 0$) of the values of variables avgBenchmark and avgSubAnnual
  - growthRateSubAnnual: Period to period difference ($\lambda = 0$) or relative difference ($\lambda \neq 0$) of the indicator series values (variable subAnnual)
  - growthRateBenchmarked: Period to period difference ($\lambda = 0$) or relative difference ($\lambda \neq 0$) of the benchmarked series values (variable benchmarked)
- splineKnots: set of x and y coordinates (knots) used to estimate the natural cubic spline with function stats::spline(). In addition to the original set of knots corresponding to binding benchmarks (anchor points), extra knots are also added at the beginning and end in order to deal with the *benchmarking timeliness issue* and approximate a *slope=0* spline at both ends (see section **Details**). It contains the following variables in addition to the BY-group variables specified with argument by:
  - varSeries: Name of the indicator series variable
  - varBenchmarks: Name of the benchmark variable
  - x: Cubic spline x coordinate
  - y: Cubic spline y coordinate

- extraKnot: Logical value identifying the extra knots added at the beginning and end

Rows for which extraKnot == FALSE correspond to rows in the graphTable output data frame for which m is not missing (not NA), with x = t and y = benchmarkedSubAnnualRatio.

Notes:

- The output benchmarks data frame always contains the original benchmarks provided in the input benchmarks data frame. Modified nonbinding benchmarks, when applicable, can be recovered (calculated) from the output series data frame.
- The function returns a NULL object if an error occurs before data processing could start. Otherwise, if execution gets far enough so that data processing could start, then an incomplete object would be returned in case of errors (e.g., output series data frame with NA values for the benchmarked data).
- The function returns "data.frame" objects that can be explicitly coerced to other types of objects with the appropriate as*() function (e.g., tibble::as_tibble() would coerce any of them to a tibble).

References

Statistics Canada (2012). "Chapter 5: Benchmarking Stock". **Theory and Application of Benchmarking (Course code 0436)**. Statistics Canada, Ottawa, Canada.

See Also

benchmarking() plot_graphTable() bench_graphs plot_benchAdj()

Examples

```
# Quarterly stock series (same pattern repeated every year)
my_series <- ts_to_tsDF(ts(rep(c(85, 95, 125, 95), 7),
                           start = c(2013, 1),
                           frequency = 4))
head(my_series)

# Annual benchmarks (end-of-year stocks)
my_benchmarks <- ts_to_bmkDF(ts(c(135, 125, 155, 145, 165),
                                start = 2013,
                                frequency = 1),
                             discrete_flag = TRUE,
                             alignment = "e",
                             ind_frequency = 4)
my_benchmarks

# Benchmark using...
#   - recommended `rho` value for quarterly series (`rho = 0.729`)
#   - proportional model (`lambda = 1`)
#   - bias-corrected indicator series with the estimated bias (`biasOption = 3`)

# ... with `benchmarking()` ("Proc Benchmarking" approach)
out_PB <- benchmarking(my_series,
                       my_benchmarks,
                       rho = 0.729,
                       lambda = 1,
                       biasOption = 3)
```

```
# ... with `stock_benchmarking()` ("Stock Benchmarking" approach)
out_SB <- stock_benchmarking(my_series,
                             my_benchmarks,
                             rho = 0.729,
                             lambda = 1,
                             biasOption = 3)

# Compare the benchmarking adjustments of both approaches
plot_benchAdj(PB_graphTable = out_PB$graphTable,
              SB_graphTable = out_SB$graphTable)


# Have you noticed how smoother the `stock_benchmarking()` adjustments are compared
# to the `benchmarking()` ones?

# The gain in the quality of the resulting benchmarked stocks might not necessarily
# be obvious in this example
plot(out_SB$graphTable$t, out_SB$graphTable$benchmarked,
     type = "b", col = "red", xlab = "t", ylab = "Benchmarked Stock")
lines(out_PB$graphTable$t, out_PB$graphTable$benchmarked,
      type = "b", col = "blue")
legend(x = "topleft", bty = "n", inset = 0.05, lty = 1, pch = 1,
       col = c("red", "blue"), legend = c("out_SB", "out_PB"))
title("Benchmarked Stock")


# What about cases where a flat indicator is used, which may happen in practice
# in absence of a good indicator of the quarterly (sub-annual) movement?
my_series2 <- my_series
my_series2$value <- 1  # flat indicator
head(my_series2)
out_PB2 <- benchmarking(my_series2,
                        my_benchmarks,
                        rho = 0.729,
                        lambda = 1,
                        biasOption = 3,
                        quiet = TRUE)  # don't show the function header

out_SB2 <- stock_benchmarking(my_series2,
                              my_benchmarks,
                              rho = 0.729,
                              lambda = 1,
                              biasOption = 3,
                              quiet = TRUE)  # don't show the function header

plot(out_SB2$graphTable$t, out_SB2$graphTable$benchmarked,
     type = "b", col = "red", xlab = "t", ylab = "Benchmarked Stock")
lines(out_PB2$graphTable$t, out_PB2$graphTable$benchmarked,
      type = "b", col = "blue")
legend(x = "bottomright", bty = "n", inset = 0.05, lty = 1, pch = 1,
       col = c("red", "blue"), legend = c("out_SB2", "out_PB2"))
title("Benchmarked Stock - Flat Indicator")


# The awkwardness of the benchmarked stocks produced by `benchmarking()` suddenly
# becomes obvious. That's because the benchmarked series corresponds to the
# benchmarking adjustments when using a flat indicator (e.g., a series on 1's
```

```
# with proportional benchmarking):
plot_benchAdj(PB_graphTable = out_PB2$graphTable,
              SB_graphTable = out_SB2$graphTable)


# The shortcomings of the "Proc Benchmarking" approach (function `benchmarking()`)
# with stocks is also quite noticeable in this case when looking at the resulting
# quarterly growth rates, which are conveniently produced by `plot_graphTable()`.
# Pay particular attention to the transition in the growth rates from Q4 to Q1
# every year in the generated PDF graphs.
plot_graphTable(out_PB2$graphTable, file.path(tempdir(), "PB_stock_flat_ind.pdf"))
plot_graphTable(out_SB2$graphTable, file.path(tempdir(), "SB_stock_flat_ind.pdf"))


# Illustrate approximating a natural cubic spline at the original end knots (first and
# last benchmarks) by specifying a large `low_freq_periodicity` value.
out_SB3 <- stock_benchmarking(my_series,
                              my_benchmarks,
                              rho = 0.729,
                              lambda = 1,
                              biasOption = 3,

                              # Large value to approximate a natural cubic spline
                              low_freq_periodicity = 100,

                              quiet = TRUE)

plot_benchAdj(SB_graphTable = out_SB3$graphTable,
              SB_splineKnots = out_SB3$splineKnots,
              legendPos = "topleft")


# Illustrate "oscillations" of the cubic spline beyond the original end knots with
# Denton-type benchmarking (`rho ~ 1`) caused by using low frequency (annual) extra knots.
out_SB4 <- stock_benchmarking(my_series,
                              my_benchmarks,
                              rho = 0.999,
                              lambda = 1,
                              biasOption = 3,

                              # Use 3 annual extra knots first
                              n_low_freq_proj = 3,
                              proj_knots_rho_bd = 1,

                              quiet = TRUE)

plot_benchAdj(SB_graphTable = out_SB4$graphTable,
              SB_splineKnots = out_SB4$splineKnots)

# No "oscillations" with the default `proj_knots_rho_bd` value because high frequency
# (quarterly) extra knots are used right away (`n_low_freq_proj` is ignored) since
# `rho = 0.999` exceeds the default `proj_knots_rho_bd` value (0.995^3 for quarterly data).
# These projected adjustments are more in line with Denton-type adjustments (straight line).
out_SB4b <- stock_benchmarking(my_series,
                               my_benchmarks,
                               rho = 0.999,
                               lambda = 1,
```

```
                              biasOption = 3,
                              quiet = TRUE)

plot_benchAdj(SB_graphTable = out_SB4b$graphTable,
              SB_splineKnots = out_SB4b$splineKnots)


# Illustrate "contortions" of the cubic spline around the original end knots caused
# by using high frequency extra knots right away (`n_low_freq_proj = 0`), i.e., using
# the same projected adjustments as those that would be obtained with `benchmarking()`.
#
# To exacerbate the phenomenon, we'll use monthly data (11 periods between each annual
# benchmark compared to only 3 for quarterly data, i.e., a less constrained spline)
# and a rather small value for `rho` (0.5 < 0.9 = recommended value for monthly data)
# for a faster convergence to the bias of the projected adjustments.

yr_vec <- unique(my_series$year)
my_series3 <- data.frame(year = rep(yr_vec, each = 12),
                         period = rep(1:12, length(yr_vec)),
                         value = rep(1, 12 * length(yr_vec)))  # flat indicator
my_benchmarks2 <- my_benchmarks
my_benchmarks2[c("startPeriod", "endPeriod")] <- 12

out_SB5 <- stock_benchmarking(my_series3,
                              my_benchmarks2,
                              rho = 0.5,
                              lambda = 1,
                              biasOption = 3,

                              # Use monthly extra knots right away
                              n_low_freq_proj = 0,

                              quiet = TRUE)

plot_benchAdj(SB_graphTable = out_SB5$graphTable,
              SB_splineKnots = out_SB5$splineKnots)

# No excessive "contortions" around the original end knots with the default
# `n_low_freq_proj = 1`, i.e., use 1 low frequency (annual) extra knot first.
out_SB5b <- stock_benchmarking(my_series3,
                               my_benchmarks2,
                               rho = 0.5,
                               lambda = 1,
                               biasOption = 3,
                               quiet = TRUE)

plot_benchAdj(SB_graphTable = out_SB5b$graphTable,
              SB_splineKnots = out_SB5b$splineKnots)

# To even better highlight the potential excessive "contortions" of the cubic spline
# when enforcing the `benchmarking()` projected adjustment (i.e., low frequency extra
# knots right away with `n_low_freq_proj = 0`), let's plot the previous two sets of
# adjustments on the same plot (the blue line corresponds to the `n_low_freq_proj = 0`
# case, i.e., the `benchmarking()` projected adjustments while the red line corresponds
# to the default `stock_benchmarking()` adjustments, i.e., `n_low_freq_proj = 1`).
plot_benchAdj(PB_graphTable = out_SB5$graphTable,
              SB_graphTable = out_SB5b$graphTable,
```

```
                legend = NULL)
```

`time_values_conv` *Time values conversion functions*

Description

Time values conversion functions used internally by other gseries functions.

Usage

```
gs.time2year(ts)

gs.time2per(ts)

gs.time2str(ts, sep = "-")
```

Arguments

`ts`	(mandatory) Time series (object of class "ts" or "mts").
`sep`	(optional) String (character constant) specifying the separator to use between the year and period values. **Default value** is `sep = "-"`.

Value

`gs.time2year()` returns an integer vector of the "nearest" year (time unit) values. This function is the equivalent of `stats::cycle()` for time unit values.

`gs.time2per()` returns an integer vector of the period (cycle) values (see `stats::cycle()`).

`gs.time2str()` returns a character vector corresponding to `gs.time2year(ts)` if `stats::frequency(ts) == 1` or `gs.time2year(ts)` and `gs.time2per(ts)` separated with `sep` otherwise.

See Also

`ts_to_tsDF()` `ts_to_bmkDF()` `gs.build_proc_grps()`

Examples

```
# Dummy monthly time series
mth_ts <- ts(rep(NA, 15), start = c(2019, 1), frequency = 12)
mth_ts
gs.time2year(mth_ts)
gs.time2per(mth_ts)
gs.time2str(mth_ts)
gs.time2str(mth_ts, sep = "m")

# Dummy quarterly time series
qtr_ts <- ts(rep(NA, 5), start = c(2019, 1), frequency = 4)
qtr_ts
```

```
gs.time2year(qtr_ts)
gs.time2per(qtr_ts)
gs.time2str(qtr_ts)
gs.time2str(qtr_ts, sep = "q")
```

`tsbalancing` *Restore cross-sectional (contemporaneous) linear constraints*

Description

*Replication of the G-Series 2.0 SAS® **GSeriesTSBalancing** macro. See the G-Series 2.0 documentation for details (Statistics Canada 2016).*

This function balances (reconciles) a system of time series according to a set of linear constraints. The balancing solution is obtained by solving one or several quadratic minimization problems (see section **Details**) with the OSQP solver (Stellato et al. 2020). Given the feasibility of the balancing problem(s), the resulting time series data respect the specified constraints for every time period. Linear equality and inequality constraints are allowed. Optionally, the preservation of temporal totals may also be specified.

Usage

```
tsbalancing(
  in_ts,
  problem_specs_df,
  temporal_grp_periodicity = 1,
  temporal_grp_start = 1,
  osqp_settings_df = default_osqp_sequence,
  display_level = 1,
  alter_pos = 1,
  alter_neg = 1,
  alter_mix = 1,
  alter_temporal = 0,
  lower_bound = -Inf,
  upper_bound = Inf,
  tolV = 0,
  tolV_temporal = 0,
  tolP_temporal = NA,

  # New in G-Series 3.0
  validation_tol = 0.001,
  trunc_to_zero_tol = validation_tol,
  full_sequence = FALSE,
  validation_only = FALSE,
  quiet = FALSE
)
```

Arguments

`in_ts`	(mandatory) Time series (object of class "ts" or "mts") that contains the time series data to be reconciled. They are the balancing problems' input data (initial solutions).

problem_specs_df

(mandatory)

Balancing problem specifications data frame. Using a sparse format inspired from the SAS/OR® LP procedure's *sparse data input format* (SAS Institute 2015), it contains only the relevant information such as the nonzero coefficients of the balancing constraints as well as the non-default alterability coefficients and lower/upper bounds (i.e., values that would take precedence over those defined with arguments alter_pos, alter_neg, alter_mix, alter_temporal, lower_bound and upper_bound).

The information is provided using four mandatory variables (type, col, row and coef) and one optional variable (timeVal). An observation (a row) in the problem specs data frame either defines a label for one of the seven types of the balancing problem elements with columns type and row (see *Label definition records* below) or specifies coefficients (numerical values) for those balancing problem elements with variables col, row, coef and timeVal (see *Information specification records* below).

- **Label definition records** (type is not missing (is not NA))
  - type (chr): reserved keyword identifying the type of problem element being defined:
    - EQ: equality ($=$) balancing constraint
    - LE: lower or equal ($\leq$) balancing constraint
    - GE: greater or equal ($\geq$) balancing constraint
    - lowerBd: period value lower bound
    - upperBd: period value upper bound
    - alter: period values alterability coefficient
    - alterTmp: temporal total alterability coefficient
  - row (chr): label to be associated to the problem element (type *keyword*)
  - *all other variables are irrelevant and should contain missing data (*NA *values)*

- **Information specification records** (type is missing (is NA))
  - type (chr): not applicable (NA)
  - col (chr): series name or reserved word _rhs_ to specify a balancing constraint right-hand side (RHS) value.
  - row (chr): problem element label.
  - coef (num): problem element value:
    - balancing constraint series coefficient or RHS value
    - series period value lower or upper bound
    - series period value or temporal total alterability coefficient
  - timeVal (num): optional time value to restrict the application of series bounds or alterability coefficients to a specific time period (or temporal group). It corresponds to the time value, as returned by stats::time(), of a given input time series (argument in_ts) period (observation) and is conceptually equivalent to $year + (period - 1)/frequency$.

Note that empty strings ("" or '') for character variables are interpreted as missing (NA) by the function. Variable row identifies the elements of the balancing problem and is the key variable that makes the link between both types of records. The same label (row) cannot be associated with more than one type

of problem element (`type`) and multiple labels (`row`) cannot be defined for the same given type of problem element (`type`), except for balancing constraints (values `"EQ"`, `"LE"` and `"GE"` of column `type`). User-friendly features of the problem specs data frame include:

- The order of the observations (rows) is not important.
- Character values (variables `type`, `row` and `col`) are not case sensitive (e.g., strings `"Constraint 1"` and `"CONSTRAINT 1"` for `row` would be considered as the same problem element label), except when `col` is used to specify a series name (a column of the input time series object) where **case sensitivity is enforced**.
- The variable names of the problem specs data frame are also not case sensitive (e.g., `type`, `Type` or `TYPE` are all valid) and `time_val` is an accepted variable name (instead of `timeVal`).

Finally, the following table lists valid aliases for the `type` *keywords* (type of problem element):

Keyword	Aliases
EQ	==, =
LE	<=, <
GE	>=, >
lowerBd	lowerBound, lowerBnd, + *same terms with '_', '.' or ' ' between words*
upperBd	upperBound, upperBnd, + *same terms with '_', '.' or ' ' between words*
alterTmp	alterTemporal, alterTemp, + *same terms with '_', '.' or ' ' between words*

Reviewing the **Examples** should help conceptualize the balancing problem specifications data frame.

`temporal_grp_periodicity`

(optional)

Positive integer defining the number of periods in temporal groups for which the totals should be preserved. E.g., specify `temporal_grp_periodicity = 3` with a monthly time series for quarterly total preservation and `temporal_grp_periodicity = 12` (or `temporal_grp_periodicity = frequency(in_ts)`) for annual total preservation. Specifying `temporal_grp_periodicity = 1` (*default*) corresponds to period-by-period processing without temporal total preservation.

Default value is `temporal_grp_periodicity = 1` (period-by-period processing without temporal total preservation).

`temporal_grp_start`

(optional)

Integer in the [1 .. `temporal_grp_periodicity`] interval specifying the starting period (cycle) for temporal total preservation. E.g., annual totals corresponding to fiscal years defined from April to March of the following year would be specified with `temporal_grp_start = 4` for a monthly time series (`frequency(in_ts) = 12`) and `temporal_grp_start = 2` for a quarterly time series (`frequency(in_ts) = 4`). This argument has no effect for period-by-period processing without temporal total preservation (`temporal_grp_periodicity = 1`).

Default value is `temporal_grp_start = 1`.

`osqp_settings_df`

(optional)

Data frame containing a sequence of OSQP settings for solving the balancing problems. The package includes two predefined OSQP settings sequence data frames:

- default_osqp_sequence: fast and effective (default);
- alternate_osqp_sequence: geared towards precision at the expense of execution time.

See `vignette("osqp-settings-sequence-dataframe")` for more details on this topic and to see the actual contents of these two data frames. Note that the concept of a *solving sequence* with different sets of solver settings is new in G-Series 3.0 (a single solving attempt was made in G-Series 2.0).

Default value is `osqp_settings_df = default_osqp_sequence`.

`display_level` (optional)

Integer in the [0 .. 3] interval specifying the level of information to display in the console (`stdout()`). Note that specifying argument `quiet = TRUE` would *nullify* argument `display_level` (none of the following information would be displayed).

Displayed information	0	1	2	3
Function header	✓	✓	✓	✓
Balancing problem elements		✓	✓	✓
Individual problem solving details			✓	✓
Individual problem results (values and constraints)				✓

Default value is `display_level = 1`.

`alter_pos` (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with **positive** coefficients in all balancing constraints in which they are involved (e.g., component series in aggregation table raking problems). Alterability coefficients provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `alter_pos = 1.0` (nonbinding values).

`alter_neg` (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with **negative** coefficients in all balancing constraints in which they are involved (e.g., marginal totals in aggregation table raking problems). Alterability coefficients provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `alter_neg = 1.0` (nonbinding values).

`alter_mix` (optional)

Nonnegative real number specifying the default alterability coefficient associated to the values of time series with a mix of **positive and negative** coefficients in the balancing constraints in which they are involved. Alterability coefficients provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `alter_mix = 1.0` (nonbinding values).

`alter_temporal` (optional)

Nonnegative real number specifying the default alterability coefficient associated to the time series temporal totals. Alterability coefficients provided in the

problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `alter_temporal = 0.0` (binding values).

`lower_bound` (optional)

Real number specifying the default lower bound for the time series values. Lower bounds provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `lower_bound = -Inf` (unbounded).

`upper_bound` (optional)

Real number specifying the default upper bound for the time series values. Upper bounds provided in the problem specification data frame (argument `problem_specs_df`) override this value.

Default value is `upper_bound = Inf` (unbounded).

`tolV` (optional)

Nonnegative real number specifying the tolerance, in absolute value, for the balancing constraints right-hand side (RHS) values:

- `EQ` constraints: $A\mathbf{x} = \mathbf{b}$ become $\mathbf{b} - \epsilon \leq A\mathbf{x} \leq \mathbf{b} + \epsilon$
- `LE` constraints: $A\mathbf{x} \leq \mathbf{b}$ become $A\mathbf{x} \leq \mathbf{b} + \epsilon$
- `GE` constraints: $A\mathbf{x} \geq \mathbf{b}$ become $A\mathbf{x} \geq \mathbf{b} - \epsilon$

where ϵ is the tolerance specified with `tolV`. This argument does not apply to the *period value (lower and upper) bounds* specified with arguments `lower_bound` and `upper_bound` or in the problem specs data frame (argument `prob_specs_df`). I.e., `tolV` does not affect the time series values lower and upper bounds, unless they are specified as *balancing constraints* instead (with `GE` and `LE` constraints in the problem specs data frame).

Default value is `tolV = 0.0` (no tolerance).

`tolV_temporal, tolP_temporal`

(optional)

Nonnegative real number, or `NA`, specifying the tolerance, in percentage (`tolP_temporal`) or absolute value (`tolV_temporal`), for the implicit temporal aggregation constraints associated to **binding temporal totals** ($\sum_t x_{i,t} = \sum_t y_{i,t}$), which become:

$$\sum_t y_{i,t} - \epsilon_{\text{abs}} \leq \sum_t x_{i,t} \leq \sum_t y_{i,t} + \epsilon_{\text{abs}}$$

or

$$\sum_t y_{i,t}\,(1 - \epsilon_{\text{rel}}) \leq \sum_t x_{i,t} \leq \sum_t y_{i,t}\,(1 + \epsilon_{\text{rel}})$$

where ϵ_{abs} and ϵ_{rel} are the absolute and percentage tolerances specified respectively with `tolV_temporal` and `tolP_temporal`. Both arguments cannot be specified together (one must be specified while the other must be `NA`).

Example: to set a tolerance of 10 *units*, specify `tolV_temporal = 10, tolP_temporal = NA`; to set a tolerance of 1%, specify`tolV_temporal = NA, tolP_temporal = 0.01`.

Default values are `tolV_temporal = 0.0` and `tolP_temporal = NA` (no tolerance).

`validation_tol` (optional)

Nonnegative real number specifying the tolerance for the validation of the balancing results. The function verifies if the final (reconciled) time series values meet the constraints, allowing for discrepancies up to the value specified with

this argument. A warning is issued as soon as one constraint is not met (discrepancy greater than `validation_tol`).

With constraints defined as $\mathbf{l} \leq A\mathbf{x} \leq \mathbf{u}$, where $\mathbf{l} = \mathbf{u}$ for `EQ` constraints, $\mathbf{l} = -\infty$ for `LE` constraints and $\mathbf{u} = \infty$ for `GE` constraints, **constraint discrepancies** correspond to $\max(0, \mathbf{l} - A\mathbf{x}, A\mathbf{x} - \mathbf{u})$, where constraint bounds $\mathbf{l}$ and $\mathbf{u}$ include the tolerances, when applicable, specified with arguments `tolV`, `tolV_temporal` and `tolP_temporal`.

Default value is `validation_tol = 0.001`.

`trunc_to_zero_tol`

(optional)

Nonnegative real number specifying the tolerance, in absolute value, for replacing by zero (small) values in the output (reconciled) time series data (output object `out_ts`). Specify `trunc_to_zero_tol = 0` to disable this *truncation to zero* process on the reconciled data. Otherwise, specify `trunc_to_zero_tol > 0` to replace with 0.0 any value in the $[-\epsilon, \epsilon]$ interval, where ϵ is the tolerance specified with `trunc_to_zero_tol`.

Note that the final constraint discrepancies (see argument `validation_tol`) are calculated on the *zero truncated* reconciled time series values, therefore ensuring accurate validation of the actual reconciled data returned by the function.

Default value is `trunc_to_zero_tol = validation_tol`.

`full_sequence` (optional)

Logical argument specifying whether all the steps of the *OSQP settings sequence data frame* should be performed or not. See argument `osqp_settings_df` and `vignette("osqp-settings-sequence-dataframe")` for more details on this topic.

Default value is `full_sequence = FALSE`.

`validation_only`

(optional)

Logical argument specifying whether the function should only perform input data validation or not. When `validation_only = TRUE`, the specified *balancing constraints* and *period value (lower and upper) bounds* constraints are validated against the input time series data, allowing for discrepancies up to the value specified with argument `validation_tol`. Otherwise, when `validation_only = FALSE` (default), the input data are first reconciled and the resulting (output) data are then validated.

Default value is `validation_only = FALSE`.

`quiet` (optional)

Logical argument specifying whether or not to display only essential information such as warnings, errors and the period (or set of periods) being reconciled. You could further suppress, if desired, the display of the *balancing period(s)* information by *wrapping* your `tsbalancing()` call with `suppressMessages()`. In that case, the `proc_grp_df` output data frame can be used to identify (unsuccessful) balancing problems associated with warning messages (if any). Note that specifying `quiet = TRUE` would also *nullify* argument `display_level`.

Default value is `quiet = FALSE`.

Details

This function solves one balancing problem per processing group (see section **Processing groups** for details). Each of these balancing problems is a quadratic minimization problem of the following

form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & (\mathbf{y}-\mathbf{x})^{\mathrm{T}} W (\mathbf{y}-\mathbf{x}) \\ \text{subject to} \quad & \mathbf{l} \leq A\mathbf{x} \leq \mathbf{u} \end{aligned}$$

where

- $\mathbf{y}$ is the vector of the initial problem values, i.e., the initial time series period values and, when applicable, temporal totals;
- $\mathbf{x}$ is the final (reconciled) version of vector $\mathbf{y}$;
- matrix $W = \operatorname{diag}(\mathbf{w})$ with vector $\mathbf{w}$ elements $w_i = \begin{cases} 0 & \text{if } |c_i y_i| = 0 \\ \frac{1}{|c_i y_i|} & \text{otherwise} \end{cases}$, where c_i is the alterability coefficient of problem value y_i and cases corresponding to $|c_i y_i| = 0$ are fixed problem values (binding period values or temporal totals);
- matrix A and vectors $\mathbf{l}$ and $\mathbf{u}$ specify the *balancing constraints*, the *implicit temporal total aggregation constraints* (when applicable), the *period value (upper and lower) bounds* as well as $x_i = y_i$ *constraints for fixed* y_i *values* ($|c_i y_i| = 0$).

In practice, the objective function of the problem solved by OSQP excludes constant term $\mathbf{y}^{\mathrm{T}} W \mathbf{y}$, therefore corresponding to $\mathbf{x}^{\mathrm{T}} W \mathbf{x} - 2(\mathbf{w}\mathbf{y})^{\mathrm{T}} \mathbf{x}$, and the fixed y_i values ($|c_i y_i| = 0$) are removed from the problem, adjusting the constraints accordingly, i.e.:

- rows corresponding to the $x_i = y_i$ *constraints for fixed* y_i *values* are removed from A, $\mathbf{l}$ and $\mathbf{u}$;
- columns corresponding to fixed y_i values are removed from A while appropriately adjusting $\mathbf{l}$ and $\mathbf{u}$.

Alterability Coefficients:
Alterability coefficients are nonnegative numbers that change the relative cost of modifying an initial problem value. By changing the actual objective function to minimize, they allow the generation of a wide range of solutions. Since they appear in the denominator of the objective function (matrix W), the larger the alterability coefficient the less costly it is to modify a problem value (period value or temporal total) and, conversely, the smaller the alterability coefficient the more costly it becomes. This results in problem values with larger alterability coefficients proportionally changing more than the ones with smaller alterability coefficients. Alterability coefficients of 0.0 define fixed (binding) problem values while alterability coefficients greater than 0.0 define free (nonbinding) values. The default alterability coefficients are 0.0 for temporal totals (argument `alter_temporal`) and 1.0 for period values (arguments `alter_pos`, `alter_neg`, `alter_mix`). In the common case of aggregation table raking problems, the period values of the marginal totals (time series with a coefficient of -1 in the balancing constraints) are usually binding (specified with `alter_neg = 0`) while the period values of the component series (time series with a coefficient of 1 in the balancing constraints) are usually nonbinding (specified with `alter_pos > 0`, e.g., `alter_pos = 1`). *Almost binding* problem values (e.g., marginal totals or temporal totals) can be obtained in practice by specifying very small (almost 0.0) alterability coefficients relative to those of the other (nonbinding) problem values.
Temporal total preservation refers to the fact that temporal totals, when applicable, are usually kept "as close as possible" to their initial value. *Pure preservation* is achieved by default with binding temporal totals while the change is minimized with nonbinding temporal totals (in accordance with the set of alterability coefficients).

Validation and troubleshooting:
Successful balancing problems (problems with a valid solution) have `sol_status_val > 0` or, equivalently, `n_unmet_con = 0` or `max_discr <= validation_tol` in the output `proc_grp_df` data frame. Troubleshooting unsuccessful balancing problems is not necessarily straightforward. Following are some suggestions:

- Investigate the failed constraints (`unmet_flag = TRUE` or, equivalently, `discr_out > validation_tol` in the output `prob_con_df` data frame) to make sure that they do not cause an empty solution space (infeasible problem).
- Change the OSQP solving sequence. E.g., try:
  1. argument `full_sequence = TRUE`
  2. argument `osqp_settings_df = alternate_osqp_sequence`
  3. arguments `osqp_settings_df = alternate_osqp_sequence` and `full_sequence = TRUE`

  See `vignette("osqp-settings-sequence-dataframe")` for more details on this topic.
- Increase (review) the `validation_tol` value. Although this may sound like *cheating*, the default `validation_tol` value (1×10^{-3}) may actually be too small for balancing problems that involve very large values (e.g., in billions) or, conversely, too large with very small problem values (e.g, < 1.0). Multiplying the average scale of the problem data by the *machine tolerance* (`.Machine$double.eps`) gives an approximation of the average size of the discrepancies that `tsbalancing()` should be able to handle (distinguish from 0) and should probably constitute an **absolute lower bound** for argument `validation_tol`. In practice, a reasonable `validation_tol` value would likely be 1×10^3 to 1×10^6 times larger than this *lower bound*.
- Address constraints redundancy. Multi-dimensional aggregation table raking problems are over-specified (involve redundant constraints) when all totals of all dimensions of the *data cube* are binding (fixed) and a constraint is defined for all of them. Redundancy also occurs for the implicit temporal aggregation constraints in single- or multi-dimensional aggregation table raking problems with binding (fixed) temporal totals. Over-specification is generally not an issue for `tsbalancing()` if the input data are not contradictory with regards to the redundant constraints, i.e., if there are no inconsistencies (discrepancies) associated to the redundant constraints in the input data or if they are *negligible* (reasonably small relative to the scale of the problem data). Otherwise, this may lead to unsuccessful balancing problems with `tsbalancing()`. Possible solutions would then include:
  1. Resolve (or reduce) the discrepancies associated to the redundant constraints in the input data.
  2. Select one marginal total in every dimension, but one, of the data cube and remove the corresponding balancing constraints from the problem. *This cannot be done for the implicit temporal aggregation constraints*.
  3. Select one marginal total in every dimension, but one, of the data cube and make them nonbinding (alterability coefficient of, say, 1.0).
  4. Do the same as (3) for the temporal totals of one of the inner-cube component series (make them nonbinding).
  5. Make all marginal totals of every dimension, but one, of the data cube *amlost binding*, i.e., specify very small alterability coefficients (say 1×10^{-6}) compared to those of the inner-cube component series.
  6. Do the same as (5) for the temporal totals of all inner-cube component series (very small alterability coefficients, e.g., with argument `alter_temporal`).
  7. Use `tsraking()` (if applicable), which handles these inconsistencies by using the Moore-Penrose inverse (uniform distribution among all binding totals).

  Solutions (2) to (7) above should only be considered if the discrepancies associated to the redundant constraints in the input data are *reasonably small* as they would be distributed among the omitted or nonbinding totals with `tsbalancing()` and all binding totals with `tsraking()`. Otherwise, one should first investigate solution (1) above.
- Relax the bounds of the problem constraints, e.g.:
  - argument `tolV` for the balancing constraints;

  - arguments tolV_temporal and tolP_temporal for the implicit temporal aggregation constraints;
  - arguments lower_bound and upper_bound.

Value

The function returns is a list of seven objects:

- out_ts: modified version of the input time series object (class "ts" or "mts"; see argument in_ts) with the resulting reconciled time series values (primary function output). It can be explicitly coerced to another type of object with the appropriate as*() function (e.g., tsibble::as_tsibble() would coerce it to a tsibble).
- proc_grp_df: processing group summary data frame, useful to identify problems that have succeeded or failed. It contains one observation (row) for each balancing problem with the following columns:
  - proc_grp (num): processing group id.
  - proc_grp_type (chr): processing group type. Possible values are:
    * "period";
    * "temporal group".
  - proc_grp_label (chr): string describing the processing group in the following format:
    * "<year>-<period>" (single periods)
    * "<start year>-<start period> - <end year>-<end period>" (temporal groups)
  - sol_status_val, sol_status (num, chr): solution status numerical (integer) value and description string:
    * 1: "valid initial solution";
    * -1: "invalid initial solution";
    * 2: "valid polished osqp solution";
    * -2: "invalid polished osqp solution";
    * 3: "valid unpolished osqp solution";
    * -3: "invalid unpolished osqp solution";
    * -4: "unsolvable fixed problem" (invalid initial solution).
  - n_unmet_con (num): number of unmet constraints (sum(prob_conf_df$unmet_flag)).
  - max_discr (num): maximum constraint discrepancy (max(prob_conf_df$discr_out)).
  - validation_tol (num): specified tolerance for validation purposes (argument validation_tol).
  - sol_type (chr): returned solution type. Possible values are:
    * "initial" (initial solution, i.e., input data values);
    * "osqp" (OSQP solution).
  - osqp_attempts (num): number of attempts made with OSQP (depth achieved in the solving sequence).
  - osqp_seqno (num): step # of the solving sequence corresponding to the returned solution. NA when sol_type = "initial".
  - osqp_status (chr): OSQP status description string (osqp_sol_info_df$status). NA when sol_type = "initial".
  - osqp_polished (logi): TRUE if the returned OSQP solution is polished (osqp_sol_info_df $status_polish = 1), FALSE otherwise. NA when sol_type = "initial".
  - total_solve_time (num): total time, in seconds, of the solving sequence.

Column proc_grp constitutes a *unique key* (distinct rows) for the data frame. Successful balancing problems (problems with a valid solution) correspond to rows with sol_status_val > 0 or, equivalently, to n_unmet_con = 0 or to max_discr <= validation_tol. The *initial solution* (sol_type = "initial") is returned only if **a)** there are no initial constraint discrepancies, **b)** the problem is fixed (all values are binding) or **c)** it beats the OSQP solution (smaller total constraint discrepancies). The OSQP solving sequence is described in vignette("osqp-settings-sequence-dataframe").

- periods_df: time periods data frame, useful to match periods to processing groups. It contains one observation (row) for each period of the input time series object (argument in_ts) with the following columns:
  - proc_grp (num): processing group id.
  - t (num): time id (1:nrow(in_ts)).
  - time_val (num): time value (stats::time(in_ts)). It conceptually corresponds to $year + (period - 1)/frequency$.

  Columns t and time_val both constitute a *unique key* (distinct rows) for the data frame.
- prob_val_df: problem values data frame, useful to analyze change diagnostics, i.e., initial vs final (reconciled) values. It contains one observation (row) for each value involved in each balancing problem, with the following columns:
  - proc_grp (num): processing group id.
  - val_type (chr): problem value type. Possible values are:
    - "period value";
    - "temporal total".
  - name (chr): time series (variable) name.
  - t (num): time id (1:nrow(in_ts)); id of the first period of the temporal group for a *temporal total*.
  - time_val (num): time value (stats::time(in_ts)); value of the first period of the temporal group for a *temporal total*. It conceptually corresponds to $year + (period - 1)/frequency$.
  - lower_bd, upper_bd (num): period value bounds; always -Inf and Inf for a *temporal total*.
  - alter (num): alterability coefficient.
  - value_in, value_out (num): initial and final (reconciled) values.
  - dif (num): value_out - value_in.
  - rdif (num): dif / value_in; NA if value_in = 0.

  Columns val_type + name + t and val_type + name + time_val both constitute a *unique key* (distinct rows) for the data frame. Binding (fixed) problem values correspond to rows with alter = 0 or value_in = 0. Conversely, nonbinding (free) problem values correspond to rows with alter != 0 and value_in != 0.
- prob_con_df: problem constraints data frame, useful for troubleshooting problems that failed (identify unmet constraints). It contains one observation (row) for each constraint involved in each balancing problem, with the following columns:
  - proc_grp (num): processing group id.
  - con_type (chr): problem constraint type. Possible values are:
    - "balancing constraint";
    - "temporal aggregation constraint";
    - "period value bounds".

While *balancing constraints* are specicied by the user, the other two types of constraints (*temporal aggregation constraints* and *period value bounds*) are automatically added to the problem by the function (when applicable).

  - `name` (chr): constraint label or time series (variable) name.
  - `t` (num): time id (`1:nrow(in_ts)`); id of the first period of the temporal group for a *temporal aggregation constraint*.
  - `time_val` (num): time value (`stats::time(in_ts)`); value of the first period of the temporal group for a *temporal aggregation constraint*. It conceptually corresponds to $year + (period - 1)/frequency$.
  - `l`, `u`, `Ax_in`, `Ax_out` (num): initial and final constraint elements ($\mathbf{l} \leq A\mathbf{x} \leq \mathbf{u}$).
  - `discr_in`, `discr_out` (num): initial and final constraint discrepancies ($\max(0, \mathbf{l} - A\mathbf{x}, A\mathbf{x} - \mathbf{u})$).
  - `validation_tol` (num): specified tolerance for validation purposes (argument `validation_tol`).
  - `unmet_flag` (logi): `TRUE` if the constraint is not met (`discr_out > validation_tol`), `FALSE` otherwise.

Columns `con_type + name + t` and `con_type + name + time_val` both constitute a *unique key* (distinct rows) for the data frame. Constraint bounds $\mathbf{l} = \mathbf{u}$ for `EQ` constraints, $\mathbf{l} = -\infty$ for `LE` constraints, $\mathbf{u} = \infty$ for `GE` constraints, and include the tolerances, when applicable, specified with arguments `tolV`, `tolV_temporal` and `tolP_temporal`.

- `osqp_settings_df`: OSQP settings data frame. It contains one observation (row) for each problem (processing group) solved with OSQP (`proc_grp_df$sol_type = "osqp"`), with the following columns:
  - `proc_grp` (num): processing group id.
  - one column corresponding to each element of the list returned by the `osqp::GetParams()` method applied to a *OSQP solver object* (class "osqp_model" object as returned by `osqp::osqp()`), e.g.:
    * Maximum iterations (`max_iter`);
    * Primal and dual infeasibility tolerances (`eps_prim_inf` and `eps_dual_inf`);
    * Solution polishing flag (`polish`);
    * Number of scaling iterations (`scaling`);
    * etc.
  - extra settings specific to `tsbalancing()`:
    * `prior_scaling` (logi): `TRUE` if the problem data were scaled (using the average of the free (nonbinding) problem values as the scaling factor) prior to solving with OSQP, `FALSE` otherwise.
    * `require_polished` (logi): `TRUE` if a polished solution from OSQP (`osqp_sol_info_df $status_polish = 1`) was required for this step in order to end the solving sequence, `FALSE` otherwise. See `vignette("osqp-settings-sequence-dataframe")` for more details on the solving sequence used by `tsbalancing()`.

  Column `proc_grp` constitutes a *unique key* (distinct rows) for the data frame. Visit `https://osqp.org/docs/interfaces/solver_settings.html` for all available OSQP settings. Problems (processing groups) for which the initial solution was returned (`proc_grp_df$sol_type = "initial"`) are not included in this data frame.
- `osqp_sol_info_df`: OSQP solution information data frame. It contains one observation (row) for each problem (processing group) solved with OSQP (`proc_grp_df$sol_type = "osqp"`), with the following columns:
  - `proc_grp` (num): processing group id.
  - one column corresponding to each element of the `info` list of a *OSQP solver object* (class "osqp_model" object as returned by `osqp::osqp()`) after having been solved with the `osqp::Solve()` method, e.g.:

    - Solution status (`status` and `status_val`);
    - Polishing status (`status_polish`);
    - Number of iterations (`iter`);
    - Objective function value (`obj_val`);
    - Primal and dual residuals (`pri_res` and `dua_res`);
    - Solve time (`solve_time`);
    - etc.
  - extra information specific to `tsbalancing()`:
    - `prior_scaling_factor` (num): value of the scaling factor when `osqp_settings_df$prior_scaling = TRUE` (`prior_scaling_factor = 1.0` otherwise).
    - `obj_val_ori_prob` (num): original balancing problem's objective function value, which is the OSQP objective function value (`obj_val`) on the original scale (when `osqp_settings_df$prior_scaling = TRUE`) plus the constant term of the original balancing problem's objective function, i.e., `obj_val_ori_prob = obj_val * prior_scaling_factor + <constant term>`, where `<constant term>` corresponds to $\mathbf{y}^{\mathrm{T}} W \mathbf{y}$. See section **Details** for the definition of vector $\mathbf{y}$, matrix W and, more generally speaking, the complete expression of the balancing problem's objective function.

  Column `proc_grp` constitutes a *unique key* (distinct rows) for the data frame. Visit https://osqp.org for more information on OSQP. Problems (processing groups) for which the initial solution was returned (`proc_grp_df$sol_type = "initial"`) are not included in this data frame.

Note that the "data.frame" objects returned by the function can be explicitly coerced to other types of objects with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce any of them to a tibble).

Processing groups

The set of periods of a given reconciliation (raking or balancing) problem is called a *processing group* and either corresponds to:

- a **single period** with period-by-period processing or, when preserving temporal totals, for the individual periods of an incomplete temporal group (e.g., an incomplete year)
- or the **set of periods of a complete temporal group** (e.g., a complete year) when preserving temporal totals.

The total number of processing groups (total number of reconciliation problems) depends on the set of periods in the input time series object (argument `in_ts`) and on the value of arguments `temporal_grp_periodicity` and `temporal_grp_start`.

Common scenarios include `temporal_grp_periodicity = 1` (default) for period-by period processing without temporal total preservation and `temporal_grp_periodicity = frequency(in_ts)` for the preservation of annual totals (calendar years by default). Argument `temporal_grp_start` allows the specification of other types of (*non-calendar*) years. E.g., fiscal years starting on April correspond to `temporal_grp_start = 4` with monthly data and `temporal_grp_start = 2` with quarterly data. Preserving quarterly totals with monthly data would correspond to `temporal_grp_periodicity = 3`.

By default, temporal groups covering more than a year (i.e., corresponding to `temporal_grp_periodicity > frequency(in_ts)` start on a year that is a multiple of `ceiling(temporal_grp_periodicity / frequency(in_ts))`. E.g., biennial groups corresponding to `temporal_grp_periodicity = 2 * frequency(in_ts)` start on an *even year* by default. This behaviour can be changed with argument `temporal_grp_start`. E.g., the preservation of biennial totals starting on an *odd*

year instead of an *even year* (default) corresponds to `temporal_grp_start = frequency(in_ts) + 1` (along with `temporal_grp_periodicity = 2 * frequency(in_ts)`).

See the `gs.build_proc_grps()` **Examples** for common processing group scenarios.

Comparing `tsraking()` and `tsbalancing()`

- `tsraking()` is limited to one- and two-dimensional aggregation table raking problems (with temporal total preservation if required) while `tsbalancing()` handles more general balancing problems (e.g., higher dimensional raking problems, nonnegative solutions, general linear equality and inequality constraints as opposed to aggregation rules only, etc.).
- `tsraking()` returns the generalized least squared solution of the Dagum and Cholette regression-based raking model (Dagum and Cholette 2006) while `tsbalancing()` solves the corresponding quadratic minimization problem using a numerical solver. In most cases, *convergence to the minimum* is achieved and the `tsbalancing()` solution matches the (exact) `tsraking()` least square solution. It may not be the case, however, if convergence could not be achieved after a reasonable number of iterations. Having said that, only in very rare occasions will the `tsbalancing()` solution *significantly* differ from the `tsraking()` solution.
- `tsbalancing()` is usually faster than `tsraking()`, especially for large raking problems, but is generally more sensitive to the presence of (small) inconsistencies in the input data associated to the redundant constraints of fully specified (over-specified) raking problems. `tsraking()` handles these inconsistencies by using the Moore-Penrose inverse (uniform distribution among all binding totals).
- `tsbalancing()` accommodates the specification of sparse problems in their reduced form. This is not true in the case of `tsraking()` where aggregation rules must always be fully specified since a *complete data cube* without missing data is expected as input (every single *inner-cube* component series must contribute to all dimensions of the cube, i.e., to every single *outer-cube* marginal total series).
- Both tools handle negative values in the input data differently by default. While the solutions of raking problems obtained from `tsbalancing()` and `tsraking()` are identical when all input data points are positive, they will differ if some data points are negative (unless argument `Vmat_option = 2` is specified with `tsraking()`).
- While both `tsbalancing()` and `tsraking()` allow the preservation of temporal totals, time management is not incorporated in `tsraking()`. For example, the construction of the processing groups (sets of periods of each raking problem) is left to the user with `tsraking()` and separate calls must be submitted for each processing group (each raking problem). That's where helper function `tsraking_driver()` comes in handy with `tsraking()`.
- `tsbalancing()` returns the same set of series as the input time series object while `tsraking()` returns the set of series involved in the raking problem plus those specified with argument `id` (which could correspond to a subset of the input series).

References

Dagum, E. B. and P. Cholette (2006). **Benchmarking, Temporal Distribution and Reconciliation Methods of Time Series**. Springer-Verlag, New York, Lecture Notes in Statistics, Vol. 186.

Ferland, M., S. Fortier and J. Bérubé (2016). "A Mathematical Optimization Approach to Balancing Time Series: Statistics Canada's GSeriesTSBalancing". In **JSM Proceedings, Business and Economic Statistics Section**. Alexandria, VA: American Statistical Association. 2292-2306.

Ferland, M. (2018). "Time Series Balancing Quadratic Problem — Hessian matrix and vector of linear objective function coefficients". **Internal document**. Statistics Canada, Ottawa, Canada.

Quenneville, B. and S. Fortier (2012). "Restoring Accounting Constraints in Time Series – Methods and Software for a Statistical Agency". **Economic Time Series: Modeling and Seasonality**. Chapman & Hall, New York.

SAS Institute Inc. (2015). "The LP Procedure Sparse Data Input Format". **SAS/OR® 14.1 User's Guide: Mathematical Programming Legacy Procedures**. https://support.sas.com/documentation/cdl/en/ormplpug/68158/HTML/default/viewer.htm#ormplpug_lp_details03.htm

Statistics Canada (2016). "The **GSeriesTSBalancing** Macro". **G-Series 2.0 User Guide**. Statistics Canada, Ottawa, Canada.

Statistics Canada (2018). **Theory and Application of Reconciliation (Course code 0437)**. Statistics Canada, Ottawa, Canada.

Stellato, B., G. Banjac, P. Goulart et al. (2020). "OSQP: an operator splitting solver for quadratic programs". **Math. Prog. Comp. 12**, 637–672 (2020). https://doi.org/10.1007/s12532-020-00179-2

See Also

tsraking() tsraking_driver() rkMeta_to_blSpecs() gs.build_proc_grps() build_balancing_problem() aliases

Examples

```
###########
# Example 1: In this first example, the objective is to balance a following simple
#            accounting table (`Profits = Revenues - Expenses`) for 5 quarters
#            without modifying `Profits` where `Revenues >= 0` and `Expenses >= 0`.

# Problem specifications
my_specs1 <- data.frame(type = c("EQ", rep(NA, 3),
                                 "alter", NA,
                                 "lowerBd", NA, NA),
                        col = c(NA, "Revenues", "Expenses", "Profits",
                                NA, "Profits",
                                NA, "Revenues", "Expenses"),
                        row = c(rep("Accounting Rule", 4),
                                rep("Alterability Coefficient", 2),
                                rep("Lower Bound", 3)),
                        coef = c(NA, 1, -1, -1,
                                 NA, 0,
                                 NA, 0, 0))
my_specs1

# Problem data
my_series1 <- ts(matrix(c( 15,  10,  10,
                            4,   8,  -1,
                          250, 250,   5,
                            8,  12,   0,
                            0,  45, -55),
                        ncol = 3,
                        byrow = TRUE,
                        dimnames = list(NULL, c("Revenues", "Expenses", "Profits"))),
                 start = c(2022, 1),
                 frequency = 4)

# Reconcile the data
```

```
out_balanced1 <- tsbalancing(in_ts = my_series1,
                             problem_specs_df = my_specs1,
                             display_level = 3)

# Initial data
my_series1

# Reconciled data
out_balanced1$out_ts

# Check for invalid solutions
any(out_balanced1$proc_grp_df$sol_status_val < 0)

# Display the maximum output constraint discrepancies
out_balanced1$proc_grp_df[, c("proc_grp_label", "max_discr")]


# The solution returned by `tsbalancing()` corresponds to equal proportional changes
# (pro-rating) and is related to the default alterability coefficients of 1. Equal
# absolute changes could be obtained instead by specifying alterability coefficients
# equal to the inverse of the initial values.
#
# Let's do this for the processing group 2022Q2 (`timeVal = 2022.25`), with the default
# displayed level of information (`display_level = 1`).

my_specs1b <- rbind(cbind(my_specs1,
                          data.frame(timeVal = rep(NA_real_, nrow(my_specs1)))),
                    data.frame(type = rep(NA, 2),
                               col = c("Revenues", "Expenses"),
                               row = rep("Alterability Coefficient", 2),
                               coef = c(0.25, 0.125),
                               timeVal = rep(2022.25, 2)))
my_specs1b

out_balanced1b <- tsbalancing(in_ts = my_series1,
                              problem_specs_df = my_specs1b)

# Display the initial 2022Q2 values and both solutions
cbind(data.frame(Status = c("initial", "pro-rating", "equal change")),
      rbind(as.data.frame(my_series1[2, , drop = FALSE]),
            as.data.frame(out_balanced1$out_ts[2, , drop = FALSE]),
            as.data.frame(out_balanced1b$out_ts[2, , drop = FALSE])),
      data.frame(Accounting_discr = c(my_series1[2, 1] - my_series1[2, 2] -
                                        my_series1[2, 3],
                                      out_balanced1$out_ts[2, 1] -
                                        out_balanced1$out_ts[2, 2] -
                                        out_balanced1$out_ts[2, 3],
                                      out_balanced1b$out_ts[2, 1] -
                                        out_balanced1b$out_ts[2, 2] -
                                        out_balanced1b$out_ts[2, 3]),
                 RelChg_Rev = c(NA,
                                out_balanced1$out_ts[2, 1] / my_series1[2, 1] - 1,
                                out_balanced1b$out_ts[2, 1] / my_series1[2, 1] - 1),
                 RelChg_Exp = c(NA,
                                out_balanced1$out_ts[2, 2] / my_series1[2, 2] - 1,
                                out_balanced1b$out_ts[2, 2] / my_series1[2, 2] - 1),
                 AbsChg_Rev = c(NA,
```

```
                                out_balanced1$out_ts[2, 1] - my_series1[2, 1],
                                out_balanced1b$out_ts[2, 1] - my_series1[2, 1]),
                  AbsChg_Exp = c(NA,
                                out_balanced1$out_ts[2, 2] - my_series1[2, 2],
                                out_balanced1b$out_ts[2, 2] - my_series1[2, 2])))


############
# Example 2: In this second example, consider the simulated data on quarterly
#            vehicle sales by region (West, Centre and East), along with a national
#            total for the three regions, and by type of vehicles (cars, trucks and
#           a total that may include other types of vehicles). The input data correspond
#            to directly seasonally adjusted data that have been benchmarked to the
#            annual totals of the corresponding unadjusted time series data as part
#            of the seasonal adjustment process (e.g., with the FORCE spec in the
#            X-13ARIMA-SEATS software).
#
#            The objective is to reconcile the regional sales to the national sales
#            without modifying the latter while ensuring that the sum of the sales of
#            cars and trucks do not exceed 95% of the sales for all types of vehicles
#            in any quarter. For illustrative purposes, we assume that the sales of
#            trucks in the Centre region for the 2nd quarter of 2022 cannot be modified.

# Problem specifications
my_specs2 <- data.frame(

  type = c("EQ", rep(NA, 4),
           "EQ", rep(NA, 4),
           "EQ", rep(NA, 4),
           "LE", rep(NA, 3),
           "LE", rep(NA, 3),
           "LE", rep(NA, 3),
           "alter", rep(NA, 4)),

  col = c(NA, "West_AllTypes", "Centre_AllTypes", "East_AllTypes", "National_AllTypes",
          NA, "West_Cars", "Centre_Cars", "East_Cars", "National_Cars",
          NA, "West_Trucks", "Centre_Trucks", "East_Trucks", "National_Trucks",
          NA, "West_Cars", "West_Trucks", "West_AllTypes",
          NA, "Centre_Cars", "Centre_Trucks", "Centre_AllTypes",
          NA, "East_Cars", "East_Trucks", "East_AllTypes",
          NA, "National_AllTypes", "National_Cars", "National_Trucks", "Centre_Trucks"),

  row = c(rep("National Total - All Types", 5),
          rep("National Total - Cars", 5),
          rep("National Total - Trucks", 5),
          rep("West Region Sum", 4),
          rep("Center Region Sum", 4),
          rep("East Region Sum", 4),
          rep("Alterability Coefficient", 5)),

  coef = c(NA, 1, 1, 1, -1,
           NA, 1, 1, 1, -1,
           NA, 1, 1, 1, -1,
           NA, 1, 1, -.95,
           NA, 1, 1, -.95,
           NA, 1, 1, -.95,
           NA, 0, 0, 0, 0),
```

```
  time_val = c(rep(NA, 31), 2022.25))

# Beginning and end of the specifications data frame
head(my_specs2, n = 10)
tail(my_specs2)

# Problem data
my_series2 <- ts(
  matrix(c(43, 49, 47, 136, 20, 18, 12, 53, 20, 22, 26, 61,
           40, 45, 42, 114, 16, 16, 19, 44, 21, 26, 21, 59,
           35, 47, 40, 133, 14, 15, 16, 50, 19, 25, 19, 71,
           44, 44, 45, 138, 19, 20, 14, 52, 21, 18, 27, 74,
           46, 48, 55, 135, 16, 15, 19, 51, 27, 25, 28, 54),
         ncol = 12,
         byrow = TRUE,
         dimnames = list(NULL,
                         c("West_AllTypes", "Centre_AllTypes", "East_AllTypes",
                           "National_AllTypes", "West_Cars", "Centre_Cars",
                           "East_Cars", "National_Cars", "West_Trucks",
                           "Centre_Trucks", "East_Trucks", "National_Trucks"))),
  start = c(2022, 1),
  frequency = 4)

# Reconcile without displaying the function header and enforce nonnegative data
out_balanced2 <- tsbalancing(
  in_ts                    = my_series2,
  problem_specs_df         = my_specs2,
  temporal_grp_periodicity = frequency(my_series2),
  lower_bound              = 0,
  quiet                    = TRUE)

# Initial data
my_series2

# Reconciled data
out_balanced2$out_ts

# Check for invalid solutions
any(out_balanced2$proc_grp_df$sol_status_val < 0)

# Display the maximum output constraint discrepancies
out_balanced2$proc_grp_df[, c("proc_grp_label", "max_discr")]


###########
# Example 3: Reproduce the `tsraking_driver()` 2nd example with `tsbalancing()`
#            (1-dimensional raking problem with annual total preservation).

# `tsraking()` metadata
my_metadata3 <- data.frame(series = c("cars_alb", "cars_sask", "cars_man"),
                           total1 = rep("cars_tot", 3))
my_metadata3

# `tsbalancing()` problem specifications
my_specs3 <- rkMeta_to_blSpecs(my_metadata3)
my_specs3
```

```
# Problem data
my_series3 <- ts(matrix(c(14, 18, 14, 58,
                          17, 14, 16, 44,
                          14, 19, 18, 58,
                          20, 18, 12, 53,
                          16, 16, 19, 44,
                          14, 15, 16, 50,
                          19, 20, 14, 52,
                          16, 15, 19, 51),
                        ncol = 4,
                        byrow = TRUE,
                        dimnames = list(NULL, c("cars_alb", "cars_sask",
                                                "cars_man", "cars_tot"))),
                 start = c(2019, 2),
                 frequency = 4)

# Reconcile the data with `tsraking()` (through `tsraking_driver()`)
out_raked3 <- tsraking_driver(in_ts = my_series3,
                              metadata_df = my_metadata3,
                              temporal_grp_periodicity = frequency(my_series3),
                              quiet = TRUE)

# Reconcile the data with `tsbalancing()`
out_balanced3 <- tsbalancing(in_ts = my_series3,
                             problem_specs_df = my_specs3,
                             temporal_grp_periodicity = frequency(my_series3),
                             quiet = TRUE)

# Initial data
my_series3

# Both sets of reconciled data
out_raked3
out_balanced3$out_ts

# Check for invalid `tsbalancing()` solutions
any(out_balanced3$proc_grp_df$sol_status_val < 0)

# Display the maximum output constraint discrepancies from the `tsbalancing()` solutions
out_balanced3$proc_grp_df[, c("proc_grp_label", "max_discr")]

# Confirm that both solutions (`tsraking() and `tsbalancing()`) are the same
all.equal(out_raked3, out_balanced3$out_ts)
```

`tsDF_to_ts` *Reciprocal function of* `ts_to_tsDF()`

Description

Convert a (non-stacked) time series data frame (`benchmarking()` and `stock_benchmarking()` data format) into a "ts" (or "mts") object.

This function is useful to convert the benchmarked data frame returned by a call to `benchmarking()` or `stock_benchmarking()` into a "ts" object, where one or several series were benchmarked in *non*

BY-group processing mode. Stacked time series data frames associated to executions in *BY-group* mode must first be *unstacked* with `unstack_tsDF()`.

Usage

```
tsDF_to_ts(
  ts_df,
  frequency,
  yr_cName = "year",
  per_cName = "period"
)
```

Arguments

`ts_df`	(mandatory) Data frame (object of class "data.frame") to be converted.
`frequency`	(mandatory) Integer specifying the frequency of the time series to be converted. The frequency of a time series corresponds to the maximum number of periods in a year (12 for a monthly data, 4 for a quarterly data, 1 for annual data).
`yr_cName, per_cName`	(optional) Strings specifying the name of the numeric variables (columns) in the input data frame that contain the data point year and period identifiers. **Default values** are `yr_cName = "year"` and `per_cName = "period"`.

Value

The function returns a time series object (class "ts" or "mts"), which can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tsibble::as_tsibble()` would coerce it to a tsibble).

See Also

`ts_to_tsDF()` `unstack_tsDF()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Initial quarterly time series (indicator series to be benchmarked)
qtr_ts <- ts(c(1.9, 2.4, 3.1, 2.2, 2.0, 2.6, 3.4, 2.4, 2.3),
             start = c(2015, 1), frequency = 4)

# Annual time series (benchmarks)
ann_ts <- ts(c(10.3, 10.2), start = 2015, frequency = 1)


# Proportional benchmarking
out_bench <- benchmarking(ts_to_tsDF(qtr_ts),
                          ts_to_bmkDF(ann_ts, ind_frequency = 4),
                          rho = 0.729, lambda = 1, biasOption = 3,
                          quiet = TRUE)

# Initial and final (benchmarked) quarterly time series ("ts" objects)
qtr_ts
```

```
tsDF_to_ts(out_bench$series, frequency = 4)


# Proportional end-of-year stock benchmarking - multiple (3) series processed
# with argument `by` (in BY-group mode)
qtr_mts <- ts.union(ser1 = qtr_ts,     ser2 = qtr_ts * 100, ser3 = qtr_ts * 10)
ann_mts <- ts.union(ser1 = ann_ts / 4, ser2 = ann_ts * 25,  ser3 = ann_ts * 2.5)
out_bench2 <- stock_benchmarking(stack_tsDF(ts_to_tsDF(qtr_mts)),
                                 stack_bmkDF(ts_to_bmkDF(
                                   ann_mts, ind_frequency = 4,
                                   discrete_flag = TRUE, alignment = "e")),
                                 rho = 0.729, lambda = 1, biasOption = 3,
                                 by = "series",
                                 quiet = TRUE)

# Initial and final (benchmarked) quarterly time series ("mts" objects)
qtr_mts
tsDF_to_ts(unstack_tsDF(out_bench2$series), frequency = 4)
```

`tsraking` *Restore cross-sectional (contemporaneous) aggregation constraints*

Description

Replication of the G-Series 2.0 SAS® TSRAKING procedure (PROC TSRAKING). See the G-Series 2.0 documentation for details (Statistics Canada 2016).

This function will restore cross-sectional aggregation constraints in a system of time series. The aggregation constraints may come from a 1 or 2-dimensional table. Optionally, temporal constraints can also be preserved.

`tsraking()` is usually called in practice through `tsraking_driver()` in order to reconcile all periods of the time series system in a single function call.

Usage

```
tsraking(
  data_df,
  metadata_df,
  alterability_df = NULL,
  alterSeries = 1,
  alterTotal1 = 0,
  alterTotal2 = 0,
  alterAnnual = 0,
  tolV = 0.001,
  tolP = NA,
  warnNegResult = TRUE,
  tolN = -0.001,
  id = NULL,
  verbose = FALSE,

  # New in G-Series 3.0
  Vmat_option = 1,
  warnNegInput = TRUE,
```

```
  quiet = FALSE
)
```

Arguments

`data_df`	(mandatory) Data frame (object of class "data.frame") that contains the time series data to be reconciled. It must minimally contain variables corresponding to the component series and cross-sectional control totals specified in the metadata data frame (argument `metadata_df`). If more than one observation (period) is provided, the sum of the provided component series values will also be preserved as part of implicit temporal constraints.
`metadata_df`	(mandatory) Data frame (object of class "data.frame") that describes the cross-sectional aggregation constraints (additivity rules) for the raking problem. Two character variables must be included in the metadata data frame: `series` and `total1`. Two variables are optional: `total2` (character) and `alterAnnual` (numeric). The values of variable `series` represent the variable names of the component series in the input time series data frame (argument `data_df`). Similarly, the values of variables `total1` and `total2` represent the variable names of the 1st and 2nd dimension cross-sectional control totals in the input time series data frame. Variable `alterAnnual` contains the alterability coefficient for the temporal constraint associated to each component series. When specified, the latter will override the default alterability coefficient specified with argument `alterAnnual`.
`alterability_df`	(optional) Data frame (object of class "data.frame"), or `NULL`, that contains the alterability coefficients variables. They must correspond to a component series or a cross-sectional control total, that is, a variable with the same name must exist in the input time series data frame (argument `data_df`). The values of these alterability coefficients will override the default alterability coefficients specified with arguments `alterSeries`, `alterTotal1` and `alterTotal2`. When the input time series data frame contains several observations and the alterability coefficients data frame contains only one, the alterability coefficients are used (repeated) for all observations of the input time series data frame. Alternatively, the alterability coefficients data frame may contain as many observations as the input time series data frame. **Default value** is `alterability_df = NULL` (default alterability coefficients).
`alterSeries`	(optional) Nonnegative real number specifying the default alterability coefficient for the component series values. It will apply to component series for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`). **Default value** is `alterSeries = 1.0` (nonbinding component series values).
`alterTotal1`	(optional) Nonnegative real number specifying the default alterability coefficient for the 1st dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`). **Default value** is `alterTotal1 = 0.0` (binding 1st dimension cross-sectional control totals)

`alterTotal2` (optional)

Nonnegative real number specifying the default alterability coefficient for the 2nd dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).

Default value is `alterTotal2 = 0.0` (binding 2nd dimension cross-sectional control totals).

`alterAnnual` (optional)

Nonnegative real number specifying the default alterability coefficient for the component series temporal constraints (e.g., annual totals). It will apply to component series for which alterability coefficients have not already been specified in the metadata data frame (argument `metadata_df`).

Default value is `alterAnnual = 0.0` (binding temporal control totals).

`tolV`, `tolP` (optional)

Nonnegative real number, or `NA`, specifying the tolerance, in absolute value or percentage, to be used when performing the ultimate test in the case of binding totals (alterability coefficient of 0.0 for temporal or cross-sectional control totals). The test compares the input binding control totals with the ones calculated from the reconciled (output) component series. Arguments `tolV` and `tolP` cannot be both specified together (one must be specified while the other must be `NA`).

Example: to set a tolerance of 10 *units*, specify `tolV = 10, tolP = NA`; to set a tolerance of 1%, specify `tolV = NA, tolP = 0.01`.

Default values are `tolV = 0.001` and `tolP = NA`.

`warnNegResult` (optional)

Logical argument specifying whether a warning message is generated when a negative value created by the function in the reconciled (output) series is smaller than the threshold specified by argument `tolN`.

Default value is `warnNegResult = TRUE`.

`tolN` (optional)

Negative real number specifying the threshold for the identification of negative values. A value is considered negative when it is smaller than this threshold.

Default value is `tolN = -0.001`.

`id` (optional)

String vector (minimum length of 1), or `NULL`, specifying the name of additional variables to be transferred from the input time series data frame (argument `data_df`) to the output time series data frame, the object returned by the function (see section **Value**). By default, the output series data frame only contains the variables listed in the metadata data frame (argument `metadata_df`).

Default value is `id = NULL`.

`verbose` (optional)

Logical argument specifying whether information on intermediate steps with execution time (real time, not CPU time) should be displayed. Note that specifying argument `quiet = TRUE` would *nullify* argument `verbose`.

Default value is `verbose = FALSE`.

`Vmat_option` (optional)

Specification of the option for the variance matrices (V_e and V_ϵ; see section **Details**):

Value	Description
1	Use vectors x and g in the variance matrices.
2	Use vectors $\|x\|$ and $\|g\|$ in the variance matrices.

See Ferland (2016) and subsection **Arguments** `Vmat_option` **and** `warnNegInput` in section **Details** for more information.

Default value is `Vmat_option = 1`.

`warnNegInput` (optional)

Logical argument specifying whether a warning message is generated when a negative value smaller than the threshold specified by argument `tolN` is found in the input time series data frame (argument `data_df`).

Default value is `warnNegInput = TRUE`.

`quiet` (optional)

Logical argument specifying whether or not to display only essential information such as warnings and errors. Specifying `quiet = TRUE` would also *nullify* argument `verbose` and is equivalent to *wrapping* your `tsraking()` call with `suppressMessages()`.

Default value is `quiet = FALSE`.

Details

This function returns the generalized least squared solution of a specific, simple variant of the general regression-based raking model proposed by Dagum and Cholette (Dagum and Cholette 2006). The model, in matrix form, is:

$$\begin{bmatrix} x \\ g \end{bmatrix} = \begin{bmatrix} I \\ G \end{bmatrix} \theta + \begin{bmatrix} e \\ \varepsilon \end{bmatrix}$$

where

- x is the vector of the initial component series values.
- θ is the vector of the final (reconciled) component series values.
- $e \sim (0, V_e)$ is the vector of the measurement errors of x with covariance matrix $V_e = \text{diag}(c_x x)$, or $V_e = \text{diag}(|c_x x|)$ when argument `Vmat_option = 2`, where c_x is the vector of the alterability coefficients of x.
- g is the vector of the initial control totals, including the component series temporal totals (when applicable).
- $\varepsilon \sim (0, V_\varepsilon)$ is the vector of the measurement errors of g with covariance matrix $V_\varepsilon = \text{diag}(c_g g)$, or $V_\varepsilon = \text{diag}(|c_g g|)$ when argument `Vmat_option = 2`, where c_g is the vector of the alterability coefficients of g.
- G is the matrix of aggregation constraints, including the implicit temporal constraints (when applicable).

The generalized least squared solution is:

$$\hat{\theta} = x + V_e G^{\mathrm{T}} \left(G V_e G^{\mathrm{T}} + V_\varepsilon \right)^{+} (g - Gx)$$

where A^{+} designates the Moore-Penrose inverse of matrix A.

`tsraking()` solves a single raking problem, i.e., either a single period of the time series system, or a single temporal group (e.g., all periods of a given year) when temporal total preservation is

required. Several call to `tsraking()` are therefore necessary in order to reconcile all the periods of the time series system. `tsraking_driver()` can achieve this in a single call: it conveniently determines the required set of raking problems to be solved and internally generates the individual calls to `tsraking()`.

Alterability Coefficients:

Alterability coefficients c_x and c_g conceptually represent the measurement errors associated with the input component series values x and control totals g respectively. They are nonnegative real numbers which, in practice, specify the extent to which an initial value can be modified in relation to other values. Alterability coefficients of 0.0 define fixed (binding) values while alterability coefficients greater than 0.0 define free (nonbinding) values. Increasing the alterability coefficient of an intial value results in more changes for that value in the reconciled (output) data and, conversely, less changes when decreasing the alterability coefficient. The default alterability coefficients are 1.0 for the component series values and 0.0 for the cross-sectional control totals and, when applicable, the component series temporal totals. These default alterability coefficients result in a proportional allocation of the discrepancies to the component series. Setting the component series alterability coefficients to the inverse of the component series initial values would result in a uniform allocation of the discrepancies instead. *Almost binding* totals can be obtained in practice by specifying very small (almost 0.0) alterability coefficients relative to those of the (nonbinding) component series.

Temporal total preservation refers to the fact that temporal totals, when applicable, are usually kept "as close as possible" to their initial value. *Pure preservation* is achieved by default with binding temporal totals while the change is minimized with nonbinding temporal totals (in accordance with the set of alterability coefficients).

Arguments `Vmat_option` and `warnNegInput`:

These arguments allow for an alternative handling of negative values in the input data, similar to that of `tsbalancing()`. Their default values correspond to the G-Series 2.0 behaviour (SAS® PROC TSRAKING) for which equivalent options are not defined. The latter was developed with "nonnegative input data only" in mind, similar to SAS® PROC BENCHMARKING in G-Series 2.0 that did not allow negative values either with proportional benchmarking, which explains the "suspicious use of proportional raking" warning in presence of negative values with PROC TSRAKING in G-Series 2.0 and when `warnNegInput = TRUE` (default). However, (proportional) raking in the presence of negative values generally works well with `Vmat_option = 2` and produces reasonable, intuitive solutions. E.g., while the default `Vmat_option = 1` fails at solving constraint `A + B = C` with input data `A = 2`, `B = -2`, `C = 1` and the default alterability coefficients, `Vmat_option = 2` returns the (intuitive) solution `A = 2.5`, `B = -1.5`, `C = 1` (25% increase for `A` and `B`). See Ferland (2016) for more details.

Treatment of Missing (`NA`) Values:

Missing values in the input time series data frame (argument `data_df`) or alterability coefficients data frame (argument `alterability_df`) for any of the raking problem data (variables listed in the metadata data frame with argument `metadata_df`) will generate an error message and stop the function execution.

Value

The function returns a data frame containing the reconciled component series, reconciled cross-sectional control totals and variables specified with argument `id`. Note that the "data.frame" object can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

Comparing `tsraking()` and `tsbalancing()`

- `tsraking()` is limited to one- and two-dimensional aggregation table raking problems (with temporal total preservation if required) while `tsbalancing()` handles more general balancing problems (e.g., higher dimensional raking problems, nonnegative solutions, general linear equality and inequality constraints as opposed to aggregation rules only, etc.).
- `tsraking()` returns the generalized least squared solution of the Dagum and Cholette regression-based raking model (Dagum and Cholette 2006) while `tsbalancing()` solves the corresponding quadratic minimization problem using a numerical solver. In most cases, *convergence to the minimum* is achieved and the `tsbalancing()` solution matches the (exact) `tsraking()` least square solution. It may not be the case, however, if convergence could not be achieved after a reasonable number of iterations. Having said that, only in very rare occasions will the `tsbalancing()` solution *significantly* differ from the `tsraking()` solution.
- `tsbalancing()` is usually faster than `tsraking()`, especially for large raking problems, but is generally more sensitive to the presence of (small) inconsistencies in the input data associated to the redundant constraints of fully specified (over-specified) raking problems. `tsraking()` handles these inconsistencies by using the Moore-Penrose inverse (uniform distribution among all binding totals).
- `tsbalancing()` accommodates the specification of sparse problems in their reduced form. This is not true in the case of `tsraking()` where aggregation rules must always be fully specified since a *complete data cube* without missing data is expected as input (every single *inner-cube* component series must contribute to all dimensions of the cube, i.e., to every single *outer-cube* marginal total series).
- Both tools handle negative values in the input data differently by default. While the solutions of raking problems obtained from `tsbalancing()` and `tsraking()` are identical when all input data points are positive, they will differ if some data points are negative (unless argument `Vmat_option = 2` is specified with `tsraking()`).
- While both `tsbalancing()` and `tsraking()` allow the preservation of temporal totals, time management is not incorporated in `tsraking()`. For example, the construction of the processing groups (sets of periods of each raking problem) is left to the user with `tsraking()` and separate calls must be submitted for each processing group (each raking problem). That's where helper function `tsraking_driver()` comes in handy with `tsraking()`.
- `tsbalancing()` returns the same set of series as the input time series object while `tsraking()` returns the set of series involved in the raking problem plus those specified with argument `id` (which could correspond to a subset of the input series).

References

Bérubé, J. and S. Fortier (2009). "PROC TSRAKING: An in-house SAS® procedure for balancing time series". In **JSM Proceedings, Business and Economic Statistics Section**. Alexandria, VA: American Statistical Association.

Dagum, E. B. and P. Cholette (2006). **Benchmarking, Temporal Distribution and Reconciliation Methods of Time Series**. Springer-Verlag, New York, Lecture Notes in Statistics, Vol. 186.

Ferland, M. (2016). "Negative Values with PROC TSRAKING". **Internal document**. Statistics Canada, Ottawa, Canada.

Fortier, S. and B. Quenneville (2009). "Reconciliation and Balancing of Accounts and Time Series". In **JSM Proceedings, Business and Economic Statistics Section**. Alexandria, VA: American Statistical Association.

Quenneville, B. and S. Fortier (2012). "Restoring Accounting Constraints in Time Series – Methods and Software for a Statistical Agency". **Economic Time Series: Modeling and Seasonality**. Chapman & Hall, New York.

Statistics Canada (2016). "The TSRAKING Procedure". **G-Series 2.0 User Guide**. Statistics Canada, Ottawa, Canada.

Statistics Canada (2018). **Theory and Application of Reconciliation (Course code 0437)**. Statistics Canada, Ottawa, Canada.

See Also

tsraking_driver() tsbalancing() rkMeta_to_blSpecs() gs.gInv_MP() build_raking_problem() aliases

Examples

```
###########
# Example 1: Simple 1-dimensional raking problem where the values of `cars` and `vans`
#            must sum up to the value of `total`.

# Problem metadata
my_metadata1 <- data.frame(series = c("cars", "vans"),
                           total1 = c("total", "total"))
my_metadata1

# Problem data
my_series1 <- data.frame(cars = 25, vans = 5, total = 40)

# Reconcile the data
out_raked1 <- tsraking(my_series1, my_metadata1)

# Initial data
my_series1

# Reconciled data
out_raked1

# Check the output cross-sectional constraint
all.equal(rowSums(out_raked1[c("cars", "vans")]), out_raked1$total)

# Check the control total (fixed)
all.equal(my_series1$total, out_raked1$total)


###########
# Example 2: 2-dimensional raking problem similar to the 1st example but adding
#            regional sales for the 3 prairie provinces (Alb., Sask. and Man.)
#            and where the sales of vans in Sask. are non-alterable
#            (alterability coefficient = 0), with `quiet = TRUE` to avoid
#            displaying the function header.

# Problem metadata
my_metadata2 <- data.frame(series = c("cars_alb", "cars_sask", "cars_man",
                                      "vans_alb", "vans_sask", "vans_man"),
                           total1 = c(rep("cars_total", 3),
                                      rep("vans_total", 3)),
                           total2 = rep(c("alb_total", "sask_total", "man_total"), 2))
my_metadata2

# Problem data
```

```
my_series2 <- data.frame(cars_alb = 12, cars_sask = 14, cars_man = 13,
                         vans_alb = 20, vans_sask = 20, vans_man = 24,
                         alb_total = 30, sask_total = 31, man_total = 32,
                         cars_total = 40, vans_total = 53)

# Reconciled data
out_raked2 <- tsraking(my_series2, my_metadata2,
                       alterability_df = data.frame(vans_sask = 0),
                       quiet = TRUE)

# Initial data
my_series2

# Reconciled data
out_raked2

# Check the output cross-sectional constraints
all.equal(rowSums(out_raked2[c("cars_alb", "cars_sask", "cars_man")]), out_raked2$cars_total)
all.equal(rowSums(out_raked2[c("vans_alb", "vans_sask", "vans_man")]), out_raked2$vans_total)
all.equal(rowSums(out_raked2[c("cars_alb", "vans_alb")]), out_raked2$alb_total)
all.equal(rowSums(out_raked2[c("cars_sask", "vans_sask")]), out_raked2$sask_total)
all.equal(rowSums(out_raked2[c("cars_man", "vans_man")]), out_raked2$man_total)

# Check the control totals (fixed)
tot_cols <- union(unique(my_metadata2$total1), unique(my_metadata2$total2))
all.equal(my_series2[tot_cols], out_raked2[tot_cols])

# Check the value of vans in Saskatchewan (fixed at 20)
all.equal(my_series2$vans_sask, out_raked2$vans_sask)
```

`tsraking_driver` *Helper function for* `tsraking()`

Description

Helper function for the `tsraking()` function that conveniently determines the required set of raking problems to be solved and internally generates the individual calls to `tsraking()`. It is especially useful in the context of temporal total (e.g., annual total) preservation where each individual raking problem either involves a single period for incomplete temporal groups (e.g., incomplete years) or several periods for complete temporal groups (e.g., the set of periods of a complete year).

Usage

```
tsraking_driver(
  in_ts,
  ...,  # `tsraking()` arguments excluding `data_df`
  temporal_grp_periodicity = 1,
  temporal_grp_start = 1
)
```

Arguments

`in_ts` (mandatory)

Time series (object of class "ts" or "mts") that contains the time series data to be reconciled. They are the raking problems' input data (initial solutions).

... Arguments passed on to tsraking

`metadata_df` (mandatory)
: Data frame (object of class "data.frame") that describes the cross-sectional aggregation constraints (additivity rules) for the raking problem. Two character variables must be included in the metadata data frame: `series` and `total1`. Two variables are optional: `total2` (character) and `alterAnnual` (numeric). The values of variable `series` represent the variable names of the component series in the input time series data frame (argument `data_df`). Similarly, the values of variables `total1` and `total2` represent the variable names of the 1st and 2nd dimension cross-sectional control totals in the input time series data frame. Variable `alterAnnual` contains the alterability coefficient for the temporal constraint associated to each component series. When specified, the latter will override the default alterability coefficient specified with argument `alterAnnual`.

`alterability_df` (optional)
: Data frame (object of class "data.frame"), or `NULL`, that contains the alterability coefficients variables. They must correspond to a component series or a cross-sectional control total, that is, a variable with the same name must exist in the input time series data frame (argument `data_df`). The values of these alterability coefficients will override the default alterability coefficients specified with arguments `alterSeries`, `alterTotal1` and `alterTotal2`. When the input time series data frame contains several observations and the alterability coefficients data frame contains only one, the alterability coefficients are used (repeated) for all observations of the input time series data frame. Alternatively, the alterability coefficients data frame may contain as many observations as the input time series data frame.
Default value is `alterability_df = NULL` (default alterability coefficients).

`alterSeries` (optional)
: Nonnegative real number specifying the default alterability coefficient for the component series values. It will apply to component series for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).
Default value is `alterSeries = 1.0` (nonbinding component series values).

`alterTotal1` (optional)
: Nonnegative real number specifying the default alterability coefficient for the 1st dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).
Default value is `alterTotal1 = 0.0` (binding 1st dimension cross-sectional control totals)

`alterTotal2` (optional)
: Nonnegative real number specifying the default alterability coefficient for the 2nd dimension cross-sectional control totals. It will apply to cross-sectional control totals for which alterability coefficients have not already been specified in the alterability coefficients data frame (argument `alterability_df`).
Default value is `alterTotal2 = 0.0` (binding 2nd dimension cross-sectional control totals).

`alterAnnual` (optional)

Nonnegative real number specifying the default alterability coefficient for the component series temporal constraints (e.g., annual totals). It will apply to component series for which alterability coefficients have not already been specified in the metadata data frame (argument `metadata_df`).
Default value is `alterAnnual = 0.0` (binding temporal control totals).

`tolV,tolP` (optional)
Nonnegative real number, or `NA`, specifying the tolerance, in absolute value or percentage, to be used when performing the ultimate test in the case of binding totals (alterability coefficient of 0.0 for temporal or cross-sectional control totals). The test compares the input binding control totals with the ones calculated from the reconciled (output) component series. Arguments `tolV` and `tolP` cannot be both specified together (one must be specified while the other must be `NA`).
Example: to set a tolerance of 10 *units*, specify `tolV = 10, tolP = NA`; to set a tolerance of 1%, specify `tolV = NA, tolP = 0.01`.
Default values are `tolV = 0.001` and `tolP = NA`.

`warnNegResult` (optional)
Logical argument specifying whether a warning message is generated when a negative value created by the function in the reconciled (output) series is smaller than the threshold specified by argument `tolN`.
Default value is `warnNegResult = TRUE`.

`tolN` (optional)
Negative real number specifying the threshold for the identification of negative values. A value is considered negative when it is smaller than this threshold.
Default value is `tolN = -0.001`.

`id` (optional)
String vector (minimum length of 1), or `NULL`, specifying the name of additional variables to be transferred from the input time series data frame (argument `data_df`) to the output time series data frame, the object returned by the function (see section **Value**). By default, the output series data frame only contains the variables listed in the metadata data frame (argument `metadata_df`).
Default value is `id = NULL`.

`verbose` (optional)
Logical argument specifying whether information on intermediate steps with execution time (real time, not CPU time) should be displayed. Note that specifying argument `quiet = TRUE` would *nullify* argument `verbose`.
Default value is `verbose = FALSE`.

`Vmat_option` (optional)
Specification of the option for the variance matrices (V_e and V_e; see section **Details**):

Value	Description
1	Use vectors x and g in the variance matrices.
2	Use vectors $\|x\|$ and $\|g\|$ in the variance matrices.

See Ferland (2016) and subsection **Arguments** `Vmat_option` **and** `warnNegInput` in section **Details** for more information.
Default value is `Vmat_option = 1`.

warnNegInput (optional)
: Logical argument specifying whether a warning message is generated when a negative value smaller than the threshold specified by argument `tolN` is found in the input time series data frame (argument `data_df`).
Default value is `warnNegInput = TRUE`.

quiet (optional)
: Logical argument specifying whether or not to display only essential information such as warnings and errors. Specifying `quiet = TRUE` would also *nullify* argument `verbose` and is equivalent to *wrapping* your `tsraking()` call with `suppressMessages()`.
Default value is `quiet = FALSE`.

temporal_grp_periodicity
: (optional)

Positive integer defining the number of periods in temporal groups for which the totals should be preserved. E.g., specify `temporal_grp_periodicity = 3` with a monthly time series for quarterly total preservation and `temporal_grp_periodicity = 12` (or `temporal_grp_periodicity = frequency(in_ts)`) for annual total preservation. Specifying `temporal_grp_periodicity = 1` (*default*) corresponds to period-by-period processing without temporal total preservation.

Default value is `temporal_grp_periodicity = 1` (period-by-period processing without temporal total preservation).

temporal_grp_start
: (optional)

Integer in the [1 .. `temporal_grp_periodicity`] interval specifying the starting period (cycle) for temporal total preservation. E.g., annual totals corresponding to fiscal years defined from April to March of the following year would be specified with `temporal_grp_start = 4` for a monthly time series (`frequency(in_ts) = 12`) and `temporal_grp_start = 2` for a quarterly time series (`frequency(in_ts) = 4`). This argument has no effect for period-by-period processing without temporal total preservation (`temporal_grp_periodicity = 1`).

Default value is `temporal_grp_start = 1`.

Details

This function solves one raking problem with `tsraking()` per processing group (see section **Processing groups** for details). The mathematical expression of these raking problem can be found in the **Details** section of the `tsraking()` documentation.

The alterability coefficients data frame (argument `alterability_df`) specified with `tsraking_driver()` can either contain:

- A single observation: the specified coefficients will be used for all periods of input time series object (argument `in_ts`).
- A number of observations equal to `frequency(in_ts)`: the specified coefficients will be used for the corresponding *cycle* of the input time series object (argument `in_ts`) periods. Monthly data example: 1st observation for January, 2nd observation for February, etc.).
- A number of observations equal to `nrow(in_ts)`: the specified coefficients will be used for the corresponding periods of the input time series object (argument `in_ts`), i.e., 1st observation for the 1st period, 2nd observation for the 2nd period, etc.).

Specifying quiet = TRUE will suppress the tsraking() messages (e.g., function header) and only display essential information such as warnings, errors and the period (or set of periods) being reconciled. We advise against *wrapping* your tsraking_driver() function call with suppressMessages() to further suppress the display of the *raking period(s)* information as this would make troubleshooting difficult in case of issues with individual raking problems.

Although tsraking() could be called with *apply() to successively reconcile all the periods of the input time series (in_ts), using tsraking_driver() has a few advantages, namely:

- temporal total preservation (only period-by-period processing, without temporal total preservation, would be possible with *apply());
- more flexibility in the specification of user-defined alterability coefficients (e.g., period-specific values);
- display of the period being processed (reconciled) in the console, which is useful for troubleshooting individual raking problems;
- improved error handling, i.e., better management of warnings or errors if they were to occur only for some raking problems (periods);
- readily returns a "ts" ("mts") object.

Value

The function returns a time series object (class "ts" or "mts") containing the reconciled component series, reconciled cross-sectional control totals and other series specified with tsraking() argument id. It can be explicitly coerced to another type of object with the appropriate as*() function (e.g., tsibble::as_tsibble() would coerce it to a tsibble).

Note that a NULL object is returned if an error occurs before data processing could start. Otherwise, if execution gets far enough so that data processing could start, then an incomplete object (with NA values) would be returned in case of errors.

Processing groups

The set of periods of a given reconciliation (raking or balancing) problem is called a *processing group* and either corresponds to:

- a **single period** with period-by-period processing or, when preserving temporal totals, for the individual periods of an incomplete temporal group (e.g., an incomplete year)
- or the **set of periods of a complete temporal group** (e.g., a complete year) when preserving temporal totals.

The total number of processing groups (total number of reconciliation problems) depends on the set of periods in the input time series object (argument in_ts) and on the value of arguments temporal_grp_periodicity and temporal_grp_start.

Common scenarios include temporal_grp_periodicity = 1 (default) for period-by period processing without temporal total preservation and temporal_grp_periodicity = frequency(in_ts) for the preservation of annual totals (calendar years by default). Argument temporal_grp_start allows the specification of other types of (*non-calendar*) years. E.g., fiscal years starting on April correspond to temporal_grp_start = 4 with monthly data and temporal_grp_start = 2 with quarterly data. Preserving quarterly totals with monthly data would correspond to temporal_grp_periodicity = 3.

By default, temporal groups covering more than a year (i.e., corresponding to temporal_grp_periodicity > frequency(in_ts) start on a year that is a multiple of ceiling(temporal_grp_periodicity / frequency(in_ts)). E.g., biennial groups corresponding to temporal_grp_per

`iodicity = 2 * frequency(in_ts)` start on an *even year* by default. This behaviour can be changed with argument `temporal_grp_start`. E.g., the preservation of biennial totals starting on an *odd year* instead of an *even year* (default) corresponds to `temporal_grp_start = frequency(in_ts) + 1` (along with `temporal_grp_periodicity = 2 * frequency(in_ts)`).

See the `gs.build_proc_grps()` **Examples** for common processing group scenarios.

References

Statistics Canada (2018). "Chapter 6: Advanced topics", **Theory and Application of Reconciliation (Course code 0437)**, Statistics Canada, Ottawa, Canada.

See Also

`tsraking()` `tsbalancing()` `rkMeta_to_blSpecs()` `gs.build_proc_grps()`

Examples

```
# 1-dimensional raking problem where the quarterly sales of cars in the 3 prairie
# provinces (Alb., Sask. and Man.) for 8 quarters, from 2019 Q2 to 2021 Q1, must
# sum up to the total (`cars_tot`).

# Problem metadata
my_metadata <- data.frame(series = c("cars_alb", "cars_sask", "cars_man"),
                          total1 = rep("cars_tot", 3))
my_metadata

# Problem data
my_series <- ts(matrix(c(14, 18, 14, 58,
                         17, 14, 16, 44,
                         14, 19, 18, 58,
                         20, 18, 12, 53,
                         16, 16, 19, 44,
                         14, 15, 16, 50,
                         19, 20, 14, 52,
                         16, 15, 19, 51),
                       ncol = 4,
                       byrow = TRUE,
                       dimnames = list(NULL, c("cars_alb", "cars_sask",
                                               "cars_man", "cars_tot"))),
                start = c(2019, 2),
                frequency = 4)


###########
# Example 1: Period-by-period processing without temporal total preservation.

# Reconcile the data
out_raked1 <- tsraking_driver(my_series, my_metadata)

# Initial data
my_series

# Reconciled data
out_raked1

# Check the output cross-sectional constraint
```

```
all.equal(rowSums(out_raked1[, my_metadata$series]), as.vector(out_raked1[, "cars_tot"]))

# Check the control total (fixed)
all.equal(my_series[, "cars_tot"], out_raked1[, "cars_tot"])


###########
# Example 2: Annual total preservation for year 2020 (period-by-period processing
#            for incomplete years 2019 and 2021), with `quiet = TRUE` to avoid
#            displaying the function header for all processing groups.

# First, check that the 2020 annual total for the total series (`cars_tot`) and the
# sum of the component series (`cars_alb`, `cars_sask` and `cars_man`) matches.
# Otherwise, this "grand total" discrepancy would first have to be resolved before
# calling `tsraking_driver()`.
tot2020 <- aggregate.ts(window(my_series, start = c(2020, 1), end = c(2020, 4)))
all.equal(as.numeric(tot2020[, "cars_tot"]), sum(tot2020[, my_metadata$series]))

# Reconcile the data
out_raked2 <- tsraking_driver(in_ts = my_series,
                              metadata_df = my_metadata,
                              quiet = TRUE,
                              temporal_grp_periodicity = frequency(my_series))

# Initial data
my_series

# Reconciled data
out_raked2

# Check the output cross-sectional constraint
all.equal(rowSums(out_raked2[, my_metadata$series]), as.vector(out_raked2[, "cars_tot"]))

# Check the output temporal constraints (2020 annual totals for each series)
all.equal(tot2020,
          aggregate.ts(window(out_raked2, start = c(2020, 1), end = c(2020, 4))))

# Check the control total (fixed)
all.equal(my_series[, "cars_tot"], out_raked2[, "cars_tot"])


###########
# Example 3: Annual total preservation for fiscal years defined from April to March
#            (2019Q2-2020Q1 and 2020Q2-2021Q1).

# Calculate the fiscal year totals (as an annual "ts" object)
fiscalYr_tot <- ts(rbind(aggregate.ts(window(my_series,
                                             start = c(2019, 2),
                                             end = c(2020, 1))),
                         aggregate.ts(window(my_series,
                                             start = c(2020, 2),
                                             end = c(2021, 1)))),
                   start = 2019,
                   frequency = 1)

# Discrepancies in both fiscal year totals (total series vs. sum of the component series)
as.numeric(fiscalYr_tot[, "cars_tot"]) - rowSums(fiscalYr_tot[, my_metadata$series])
```

```
# 3a) Reconcile the fiscal year totals (rake the fiscal year totals of the component series
#     to those of the total series).
new_fiscalYr_tot <- tsraking_driver(in_ts = fiscalYr_tot,
                                    metadata_df = my_metadata,
                                    quiet = TRUE)

# Confirm that the previous discrepancies are now "gone" (are both zero)
as.numeric(new_fiscalYr_tot[, "cars_tot"]) - rowSums(new_fiscalYr_tot[, my_metadata$series])

# 3b) Benchmark the quarterly component series to these new (coherent) fiscal year totals.
out_bench <- benchmarking(series_df = ts_to_tsDF(my_series[, my_metadata$series]),
                          benchmarks_df = ts_to_bmkDF(
                            new_fiscalYr_tot[, my_metadata$series],
                            ind_frequency = frequency(my_series),

                            # Fiscal years starting on Q2 (April)
                            bmk_interval_start = 2),

                          rho = 0.729,
                          lambda = 1,
                          biasOption = 2,
                          allCols = TRUE,
                          quiet = TRUE)
my_new_ser <- tsDF_to_ts(cbind(out_bench$series, cars_tot = my_series[, "cars_tot"]),
                         frequency = frequency(my_series))

# 3c) Reconcile the quarterly data with preservation of fiscal year totals.
out_raked3 <- tsraking_driver(in_ts = my_new_ser,
                              metadata_df = my_metadata,
                              temporal_grp_periodicity = frequency(my_series),

                              # Fiscal years starting on Q2 (April)
                              temporal_grp_start = 2,

                              quiet = TRUE)

# Initial data
my_series

# With coherent fiscal year totals
my_new_ser

# Reconciled data
out_raked3

# Check the output cross-sectional constraint
all.equal(rowSums(out_raked3[, my_metadata$series]), as.vector(out_raked3[, "cars_tot"]))

# Check the output temporal constraints (both fiscal year totals for all series)
all.equal(rbind(aggregate.ts(window(my_new_ser, start = c(2019, 2), end = c(2020, 1))),
                aggregate.ts(window(my_new_ser, start = c(2020, 2), end = c(2021, 1)))),
          rbind(aggregate.ts(window(out_raked3, start = c(2019, 2), end = c(2020, 1))),
                aggregate.ts(window(out_raked3, start = c(2020, 2), end = c(2021, 1)))))

# Check the control total (fixed)
```

```
all.equal(my_series[, "cars_tot"], out_raked3[, "cars_tot"])
```

`ts_to_bmkDF` *Convert a "ts" object to a benchmarks data frame*

Description

Convert a "ts" (or "mts") object into a benchmarks data frame for the benchmarking functions with five or more variables (columns):

- four (4) for the benchmark coverage
- one (1) for each benchmark time series

For discrete benchmarks (anchor points covering a single period of the indicator series, e.g., end of year stocks), specify `discrete_flag = TRUE` and `alignment = "b"`, `"e"` or `"m"`.

Usage

```
ts_to_bmkDF(
  in_ts,
  ind_frequency,
  discrete_flag = FALSE,
  alignment = "b",
  bmk_interval_start = 1,
  startYr_cName = "startYear",
  startPer_cName = "startPeriod",
  endYr_cName = "endYear",
  endPer_cName = "endPeriod",
  val_cName = "value"
)
```

Arguments

`in_ts`	(mandatory) Time series (object of class "ts" or "mts") to be converted.
`ind_frequency`	(mandatory) Integer specifying the frequency of the indicator (high frequency) series for which the benchmarks (low frequency series) are related to. The frequency of a time series corresponds to the maximum number of periods in a year (e.g., 12 for a monthly data, 4 for a quarterly data, 1 for annual data).
`discrete_flag`	(optional) Logical argument specifying whether the benchmarks correspond to discrete values (anchor points covering a single period of the indicator series, e.g., end of year stocks) or not. `discrete_flag = FALSE` defines non-discrete benchmarks, i.e., benchmarks that cover several periods of the indicator series (e.g. annual benchmarks cover 4 quarters or 12 months, quarterly benchmarks cover 3 months, etc.). **Default value** is `discrete_flag = FALSE`.
`alignment`	(optional) Character identifying the alignment of discrete benchmarks (argument `discrete_flag = TRUE`) in the benchmark (low frequency series) interval coverage window:

- `alignment = "b"`: beginning of the benchmark interval window (first period)
- `alignment = "e"`: end of the benchmark interval window (last period)
- `alignment = "m"`: middle of the benchmark interval window (middle period)

This argument has no effect for non-discrete benchmarks (`discrete_flag = FALSE`).

Default value is `alignment = "b"`.

`bmk_interval_start`

(optional)

Integer in the [1 .. `ind_frequency`] interval specifying the period (cycle) of the indicator (high frequency) series at which the benchmark (low frequency series) interval window starts. E.g., annual benchmarks corresponding to fiscal years defined from April to March of the following year would be specified with `bmk_interval_start = 4` for a monthly indicator series (`ind_frequency = 12`) and `bmk_interval_start = 2` for a quarterly indicator series (`ind_frequency = 4`).

Default value is `bmk_interval_start = 1`.

`startYr_cName`, `startPer_cName`, `endYr_cName`, `endPer_cName`

(optional)

Strings specifying the name of the numeric variables (columns) in the output data frame that will define the benchmarks coverage, i.e., the starting and ending year and period (cycle) identifiers.

Default values are `startYr_cName = "startYear"`, `startPer_cName = "startPeriod"` `endYr_cName = "endYear"` and `endPer_cName = "endPeriod"`.

`val_cName`

(optional)

String specifying the name of the numeric variable (column) in the output data frame that will contain the benchmark values. This argument has no effect for "mts" objects (benchmark variable names are automatically inherited from the "mts" object).

Default value is `val_cName = "value"`.

Value

The function returns a data frame with five or more variables:

- Benchmark coverage starting year, type numeric (see argument `startYr_cName`)
- Benchmark coverage starting period (cycle), type numeric (see argument `startPer_cName`)
- Benchmark coverage ending year, type numeric (see argument `endtYr_cName`)
- Benchmark coverage ending period (cycle), type numeric (see argument `endPer_cName`)
- One ("ts" object) or many ("mts" object) benchmark data variable(s), type numeric (see argument `val_cName`)

Note: the function returns a "data.frame" object than can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

See Also

`ts_to_tsDF()` `stack_bmkDF()` `benchmarking()` `stock_benchmarking()` time_values_conv

Examples

```
# Annual and quarterly time series
my_ann_ts <- ts(1:5 * 100, start = 2019, frequency = 1)
my_ann_ts
my_qtr_ts <- ts(my_ann_ts, frequency = 4)
my_qtr_ts


# Annual benchmarks for a monthly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 12)

# Annual benchmarks for a quarterly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 4)

# Quarterly benchmarks for a monthly indicator series
ts_to_bmkDF(my_qtr_ts, ind_frequency = 12)

# Start of year stocks for a quarterly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 4,
            discrete_flag = TRUE)

# End of quarter stocks for a monthly indicator series
ts_to_bmkDF(my_qtr_ts, ind_frequency = 12,
            discrete_flag = TRUE, alignment = "e")

# April to March annual benchmarks for a ...
# ... monthly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 12,
            bmk_interval_start = 4)
# ... quarterly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 4,
            bmk_interval_start = 2)

# End-of-year (April to March) stocks for a ...
# ... monthly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 12,
            discrete_flag = TRUE, alignment = "e", bmk_interval_start = 4)
# ... quarterly indicator series
ts_to_bmkDF(my_ann_ts, ind_frequency = 4,
            discrete_flag = TRUE, alignment = "e", bmk_interval_start = 2)

# Custom name for the benchmark data variable (column)
ts_to_bmkDF(my_ann_ts, ind_frequency = 12,
            val_cName = "bmk_val")

# Multiple time series: argument `val_cName` ignored
# (the "mts" object column names are always used)
ts_to_bmkDF(ts.union(ser1 = my_ann_ts, ser2 = my_ann_ts / 10), ind_frequency = 12,
            val_cName = "useless_column_name")
```

`ts_to_tsDF` *Convert a "ts" object to a time series data frame*

Description

Convert a "ts" (or "mts") object into a time series data frame for the benchmarking functions with three or more variables (columns):

- two (2) for the data point identification (year and period)
- one (1) for each time series

Usage

```
ts_to_tsDF(
  in_ts,
  yr_cName = "year",
  per_cName = "period",
  val_cName = "value"
)
```

Arguments

`in_ts`	(mandatory) Time series (object of class "ts" or "mts") to be converted.
`yr_cName`, `per_cName`	(optional) Strings specifying the name of the numeric variables (columns) in the output data frame that will contain the data point year and period identifiers. **Default values** are `yr_cName = "year"` and `per_cName = "period"`.
`val_cName`	(optional) String specifying the name of the numeric variable (column) in the output data frame that will contain the data point value. This argument has no effect for "mts" objects (time series data variable names are automatically inherited from the "mts" object). **Default value** is `val_cName = "value"`.

Value

The function returns a data frame with three or more variables:

- Data point year, type numeric (see argument `startYr_cName`)
- Data point period, type numeric (see argument `startPer_cName`)
- One ("ts" object) or many ("mts" object) time series data variable(s), type numeric (see argument `val_cName`)

Note: the function returns a "data.frame" object than can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

See Also

`tsDF_to_ts()` `ts_to_bmkDF()` `stack_tsDF()` `benchmarking()` `stock_benchmarking()` time_values_conv

Examples

```
# Quarterly time series
my_ts <- ts(1:10 * 100, start = 2019, frequency = 4)
my_ts


# With the default variable (column) names
ts_to_tsDF(my_ts)

# Using a custom name for the time series data variable (column)
ts_to_tsDF(my_ts, val_cName = "ser_val")


# Multiple time series: argument `val_cName` ignored
# (the "mts" object column names are always used)
ts_to_tsDF(ts.union(ser1 = my_ts,
                    ser2 = my_ts / 10),
           val_cName = "useless_column_name")
```

---

unstack_tsDF *Reciprocal function of* stack_tsDF()

Description

Convert a stacked (tall) multivariate time series data frame (benchmarking() and stock_benchmarking() data format) into a non-stacked (wide) multivariate time series data frame.

This function, combined with tsDF_to_ts(), is useful to convert the benchmarked data frame returned by a call to benchmarking() or stock_benchmarking() back into a "mts" object, where multiple series were benchmarked in *BY-group* processing mode.

Usage

```
unstack_tsDF(
  ts_df,
  ser_cName = "series",
  yr_cName = "year",
  per_cName = "period",
  val_cName = "value"
)
```

Arguments

ts_df	(mandatory) Data frame (object of class "data.frame") that contains the multivariate time series data to be *unstacked*.
ser_cName	(optional) String specifying the name of the character variable (column) in the input time series data frame that contains the series identifier (the time series variable names in the output data frame). **Default value** is ser_cName = "series".

`yr_cName`, `per_cName`
: (optional)
Strings specifying the name of the numeric variables (columns) in the input time series data frame that contain the data point year and period identifiers. These variables are *transferred* to the output data frame with the same names.
Default values are `yr_cName = "year"` and `per_cName = "period"`.

`val_cName`
: (optional)
String specifying the name of the numeric variable (column) in the input time series data frame that contains the data point values.
Default value is `val_cName = "value"`.

Value

The function returns a data frame with three or more variables:

- Data point year, type numeric (see argument `yr_cName`)
- Data point period, type numeric (see argument `per_cName`)
- One time series data variable for each distinct value of the input data frame variable specified with argument `ser_cName`, type numeric (see arguments `ser_cName` and `val_cName`)

Note: the function returns a "data.frame" object than can be explicitly coerced to another type of object with the appropriate `as*()` function (e.g., `tibble::as_tibble()` would coerce it to a tibble).

See Also

`stack_tsDF()` `tsDF_to_ts()` `benchmarking()` `stock_benchmarking()`

Examples

```
# Proportional benchmarking for multiple (3) quarterly series processed with
# argument `by` (in BY-group mode)

ind_vec <- c(1.9, 2.4, 3.1, 2.2, 2.0, 2.6, 3.4, 2.4, 2.3)
ind_df <- ts_to_tsDF(ts(data.frame(ser1 = ind_vec,
                                   ser2 = ind_vec * 100,
                                   ser3 = ind_vec * 10),
                        start = c(2015, 1), frequency = 4))

bmk_vec <- c(10.3, 10.2)
bmk_df <- ts_to_bmkDF(ts(data.frame(ser1 = bmk_vec,
                                    ser2 = bmk_vec * 100,
                                    ser3 = bmk_vec * 10),
                         start = 2015, frequency = 1),
                      ind_frequency = 4)

out_bench <- benchmarking(stack_tsDF(ind_df),
                          stack_bmkDF(bmk_df),
                          rho = 0.729, lambda = 1, biasOption = 3,
                          by = "series",
                          quiet = TRUE)

# Initial and final (benchmarked) quarterly time series data frames
ind_df
unstack_tsDF(out_bench$series)
```

Index